Asynchronous MPC with $t < n/2$ Using Non-equivocation

Michael Backes[1] Fabian Bendun[1] Ashish Choudhury[2] Aniket Kate[3]

[1]Center for IT-Security, Privacy and Accountability (CISPA), Saarland University, Germany
[2] Department of Computer Science and Engineering, Jadavpur University, India
[3]MMCI, Saarland University, Germany
{backes,bendun}@cs.uni-saarland.de, achoudhury@cse.jdvu.ac.in,
aniket@mmci.uni-saarland.de

Abstract

Multiparty computation (MPC) among n parties can tolerate up to $t < n/2$ active corruptions in a *synchronous* communication setting; however, in an *asynchronous* communication setting, the resiliency bound decreases to only $t < n/3$ active corruptions. We improve the resiliency bound for asynchronous MPC (AMPC) to match synchronous MPC using *non-equivocation*.

Non-equivocation is a message authentication mechanism to restrict a corrupted sender from making conflicting statements to different (honest) parties. It can be implemented using an increment-only counter and a digital signature oracle, realizable with trusted hardware modules readily available in commodity computers and smartphone devices. A non-equivocation mechanism can also be transferable and allow a receiver to verifiably transfer the authenticated statement to other parties. In this work, using transferable non-equivocation, we present an AMPC protocol tolerating $t < n/2$ faults. From a practical point of view, our AMPC protocol requires fewer setup assumptions than the previous AMPC protocol with $t < n/2$ by Beerliová-Trubíniová, Hirt and Nielsen [PODC 2010]: unlike their AMPC protocol, it does not require *any* synchronous broadcast round at the beginning of the protocol and avoids the threshold homomorphic encryption setup assumption. Moreover, our AMPC protocol is also efficient and provides a gain of $\Theta(n)$ in the communication complexity per multiplication gate, over the AMPC protocol of Beerliová-Trubíniová et al. In the process, using non-equivocation, we also define the first asynchronous verifiable secret sharing (AVSS) scheme with $t < n/2$, which is of independent interest to threshold cryptography.

1 Introduction

Multi-party computation (MPC) is an important primitives in distributed systems. Informally, in a system of n mutually distrusting parties, an MPC protocol allows the parties to "securely" evaluate any agreed-on function f of their private inputs, in the presence of a centralized active adversary $\mathcal{A}$, controlling at most any t out of the n parties. In the synchronous communication model, where the message transfer delays are bounded by a known constant, the MPC problem has been studied extensively (e.g., [Yao82, GMW87, BOGW88, CCD88, RBO89, CDN01, BTH06, BTH08, BSFO12]). In practice, there is growing interest in generalizing MPC to an asynchronous communication model [BOCG93, Can96, CKAS02] that does not place any bound on the communication delays. The weaker restrictions on the adversary in the asynchronous model not only worsen the required resiliency conditions and communication complexities, but also make designing protocols a more challenging task; intuitively this is because in a completely asynchronous setting, it is not possible to distinguish between a slow (but honest) sender and a crashed sender. Due to this, at any "stage" of an asynchronous protocol, no party can afford to wait to hear from all the parties and so the

communication from t (potentially honest) parties may be ignored [Can96]. Due to their complexity, only a few asynchronous MPC (AMPC) protocols are available [BOCG93, BOKR94, SR00, HNP05, HNP08, BTH07, PCR09, CHP13].

In this work, we focus on an asynchronous model with a *computationally bounded* adversary $\mathcal{A}$, where the parties are connected by pairwise authenticated links. In this setting, AMPC protocols are possible if and only if $t < n/3$ [HNP05, HNP08]. This is in contrast to the synchronous world, where we can tolerate upto $t < n/2$ corruptions [HN06]. Interested in bridging this gap between the resilience of synchronous and asynchronous MPC protocols, Beerliová-Trubíniová, Hirt and Nielsen [BTHN10] observed that it is possible to design an AMPC protocol tolerating $t < n/2$ corruptions in a "partial" synchronous network. More specifically, assuming one *synchronous broadcast* round at the beginning of the protocol, where each party can synchronously broadcast to every other party, they designed an AMPC protocol tolerating $t < n/2$ corruptions. Due to the availability of the synchronous broadcast round, their protocol could also ensure "input provision", i.e. the inputs of all the (honest) parties are considered for the computation, which otherwise is impossible to achieve in an asynchronous protocol [Can96]. Nevertheless, their requirement of one synchronous broadcast round per MPC instance may not always be realizable: deterministic broadcast protocols [DS83] require $\Theta(t)$ rounds of communication over the pairwise channels or randomized broadcast protocols [Tou84, FM85] require $\mathcal{O}(1)$ (with a large constant) expected rounds of communication. It was left as an open problem in [BTHN10] to see whether one can design an AMPC protocol with $t < n/2$ under other simplified assumptions.

In distributed computing research, a similar problem with asynchronous protocols has recently been addressed by introducing a small trusted hardware assumption [CMSK07, LDLM09, CVL10, CJKR12, JMS12, KBC$^+$12]. In particular, it was shown that, the resilience of asynchronous distributed computing tasks such as reliable broadcast, Byzantine agreement, and state machine replication (SMR) can be improved using a small *trusted hardware module* at each party. The hardware module utilized is just a trusted, increment-only local counter and a signature oracle, which can be realized with pervasively available trusted hardware-enabled devices. Using such trusted hardware with each party, one can design asynchronous reliable broadcast tolerating up to $t < n$ active faults [CJKR12, CVL10], and asynchronous Byzantine agreement (ABA) and SMR protocols tolerating up to $t < n/2$ [CMSK07, LDLM09, KBC$^+$12] active faults, all of which otherwise require $t < n/3$ [Tou84].

At a conceptual level, such a trusted module makes it impossible for a corrupted party to perform *equivocation*, which essentially means making conflicting statements to different (honest) parties. The use of signatures (or transferable authentication) complements *non-equivocation* (i.e., making equivocation impossible) by making it transferable as required in the asynchronous environment with unknown delays. Clement et al. [CJKR12] generalized the results [CMSK07, LDLM09, CVL10] and proved that non-equivocation with signatures (i.e., *transferable* non-equivocation) allows treating active (or Byzantine) faults as crash failure for many distributed computing primitives. In particular, they present a generic transformation that enables any crash-fault tolerant distributed protocol to tolerate the same number of Byzantine faults using transferable non-equivocation. Nevertheless, their generic transformation considers only the basic distributed computing requirements of safety and liveness. It does not apply to cryptographic tasks such as AMPC where *confidentiality (or privacy)* of inputs is also required. This presents an interesting challenge to assess the utility of transferable non-equivocation for the secure distributed computing task of AMPC.

1.1 Contribution and Comparison

We study the power of transferable non-equivocation in the context of AMPC and demonstrate how to improve the resilience of AMPC from $t < n/3$ to $t < n/2$, without any synchrony assumption. In particular, we present a general MPC protocol in a completely asynchronous communication model with $n \geq 2t + 1$. Our protocol, called NeqAMPC , improves upon the previous AMPC protocol [BTHN10] with $n \geq 2t + 1$ in the following ways:

(a) Simplified assumptions. The NeqAMPC protocol needs a transferable non-equivocation mechanism, but unlike [BTHN10] neither makes a synchronous broadcast round assumption nor requires a threshold homomorphic encryption setup. Given the feasibility of realizing transferable non-equivocation over prevelent computing devices, we argue that transferable non-equivocation is a more practical assumption than the synchronous broadcast round assumption.

(b) Efficiency. For a security parameter κ, our AMPC protocol requires an amortized communication complexity of $\mathcal{O}(n^3\kappa)$ bits per multiplication gate, which improves upon the AMPC protocol of [BTHN10] by a factor of $\Theta(n)$.

To reduce the setup assumptions for the NeqAMPC protocol, we avoid the traditional threshold additive homomorphic encryption based circuit evaluation approach as used in [HNP05, HNP08, BTHN10]. Instead, we employ a secret-sharing based circuit evaluation approach [BOGW88, CCD88, RBO89], where privacy of the computation is maintained via secret sharing. Nevertheless, as detailed in our protocol overview (§2), secret-sharing based AMPC with $n = 2t + 1$ and $\mathcal{O}(n^3\kappa)$ communication complexity (per multiplication) presents several interesting challenges. As a result the NeqAMPC protocol is significantly different than those in the literature [HNP05, HNP08, BTH07, BTHN10].

In the process, we also present the first computationally secure asynchronous verifiable secret sharing (AVSS) [Can96, CKAS02, BKP11, BDK13] scheme for $n \geq 2t+1$ with $\mathcal{O}(n^2\kappa)$ communication complexity (using transferable non-equivocation), which otherwise requires $t < n/3$ [Can96]. Our AVSS scheme has an additional useful feature—it is the first publicly verifiable [Sta96] AVSS scheme, as it allows any third party to publicly verify the "consistency" of the shares. With its efficiency and public verifiability, our AVSS scheme may be of independent interest to other cryptographic protocols.

Comparison with Existing Work. The best known computationally secure AMPC protocols are reported in [HNP08, BTHN10]. The protocol in [HNP08] considers a *fully* asynchronous setting with $t < n/3$, whereas [BTHN10] assumes one synchronous broadcast round and can tolerate up to $t < n/2$ corruptions. Both the protocols require a threshold additive homomorphic encryption instantiation, and incur an (amortized) communication complexity of $\mathcal{O}(n^2\kappa)$ and $\mathcal{O}(n^4\kappa)$ bits per multiplication gate respectively.[1]

We do not employ a threshold encryption setup, but rather prefer a more standard public key encryption setup with the addition of transferable non-equivocation. Our NeqAMPC protocol with $t < n/2$ performs circuit evaluation by secret-sharing the inputs and incurs a communication complexity of $\mathcal{O}(n^3\kappa)$ bits per multiplication gate. Nevertheless, by modifying our protocol and employing a threshold encryption setup (coupled with transferable non-equivocation), we can tolerate $t < n/2$ faults with communication complexity $\mathcal{O}(n^2\kappa)$ bits per multiplication gate. However, we prefer the secret-sharing based AMPC, as we aim to reduce the assumptions relied upon.

We note that unlike [BTHN10], our AMPC protocol could not enforce *input provision*: the input from t potentially honest parties may be ignored for computation. As discussed earlier, this is inherent to asynchronous systems and presents a trade-off between our protocol and that of [BTHN10] based on what is more important: input provision or avoiding the synchrony assumption. Finally, we note that using a transferable non-equivocation mechanism, one can realize asynchronous reliable broadcast (see §3.3) with $t < n$ and consequently get rid of the synchronous broadcast round required in [BTHN10]. Nevertheless, the resultant protocol will still require the threshold homomorphic encryption setup and $\mathcal{O}(n^4\kappa)$ communication complexity per multiplication, and it will no longer support input provision.

[1] Beerliová-Trubíniová et al. [BTHN10] focused on designing a protocol with $t < n/2$, and the communication complexity of $\mathcal{O}(n^4\kappa)$ of their protocol (measured by us in Appendix E) can possibly be improved.

2 Overview of Our NeqAMPC Protocol

Without loss of generality, we assume $n = 2t + 1$; thus, $t = \Theta(n)$. We assume that the function f to be computed is expressed as an arithmetic circuit over the field $\mathbb{Z}_p$, where $p > n$ is a κ bit prime and κ is the security parameter. The circuit consists of two-input addition (linear) and multiplication (non-linear) gates, apart from random gates. The AMPC protocol consists of two phases: an input phase and a computation phase. During the input phase, the parties share their inputs, while during the computation phase, the parties jointly evaluate f on the shared inputs and publicly reconstruct the output. Linear gates can be evaluated locally if the underlying secret-sharing scheme is linear; thus, we use the polynomial-based (Shamir) secret-sharing scheme with threshold t [Sha79]. We denote a sharing of a value s by $[s]$. It follows that locally adding the shares of $[x]$ and $[y]$ provides the shares for $[x + y]$.

Multiplication gates cannot be evaluated locally since multiplying the individual shares results in the underlying sharing polynomial having degree $2t$ instead of t. Therefore we evaluate multiplication gates using the standard Beaver's circuit-randomization technique [Bea91]. This technique requires three "pre-processed" secret-shared values, say $([u], [v], [w])$, unknown to the adversary $\mathcal{A}$, such that $w = u \cdot v$. Given such a shared *multiplication triple*, and shared inputs of a multiplication gate, say $[x]$ and $[y]$, the multiplication gate is securely evaluated using the equation $[x \cdot y] = (x - u) \cdot (y - v) + [v] \cdot (x - u) + [u] \cdot (y - v) + [u \cdot v]$. In particular, the parties compute the sharings of $(x - u)$ and $(y - v)$, and publicly reconstruct the same. Once $(x-u)$ and $(y-v)$ are public, the parties can compute their shares of $x \cdot y$, using the above equation and employing linearity of the secret sharing. As u and v are random and unknown to $\mathcal{A}$, the public knowledge of $(x - u)$ and $(y - v)$ does not violate the privacy of x and y.

2.1 Pre-processing Phase

Although our above AMPC protocol idea is the same as the existing information-theoretically secure MPC and AMPC protocols [DN07, BTH07, BTH08, CHP13], our major challenge lies in generating the required shared multiplication triples with $n = 2t+1$ parties; the existing protocols [DN07, BTH07, BTH08, CHP13] employ at least $n > 3t$ parties for this purpose[2]. These triplets are independent of the circuit and the inputs of the parties, and generated in an additional *pre-processing* phase. Generating these triplets efficiently is the important problem we solve in our protocol. In the rest of the section, we give an overview of how $(c_M + c_R)$ shared random triples are generated, where c_M and c_R are the number of multiplication gates and random gates in the circuit. As a first step, we describe how a single triple is generated (see Figure 1 for a pictorial representation of the protocols involved) and then extend this to $c_M + c_R$ triples.

Supervised Triple Generation (§5). The idea for generating a random shared multiplication triple $[u], [v], [w]$ is to compute a random sharing $[v]$ and then combining "several" $[u^{(i)}]$s and $[u^{(i)} \cdot v]$s to get $[u]$ and $[w]$. The triple generation protocol uses two sub-protocols: Sup-Sh and Sup-PreMul-Sh. Protocol Sup-Sh allows a *dealer* D to "verifiably" generate the sharing $[u]$ of his value u, while Sup-PreMul-Sh allows a dealer D to "verifiably" generate a sharing $[u]$ and $[u \cdot v]$, given u and $[v]$. To generate $([u], [v], [w])$, we use Sup-Sh and Sup-PreMul-Sh in the following way: first, we ask each party P_i to act as a dealer D and invoke an instance of Sup-Sh to share a uniformly random value, say $v^{(i)}$. The parties then agree on a common subset (say $\mathcal{T}_v$) of $t + 1$ dealers whose Sup-Sh instances will be eventually terminated by all the parties. We set $v = \sum_{P_i \in \mathcal{T}_v} v^{(i)}$. The shared value v will be random and unknown to $\mathcal{A}$, as $\mathcal{T}_v$ has at least one honest party. Next, each party P_i is asked to act as a D and invoke a Sup-PreMul-Sh instance to share a uniformly random value $u^{(i)}$ as well as $u^{(i)} \cdot v$. The parties then agree on a common subset of $t + 1$ dealers, say $\mathcal{T}_u$,

[2]Shared multiplication triples with $n = 2t + 1$ have been generated in the synchronous setting [BTH06, BSFO12]; however, their adaptability to the asynchronous setting is unclear.

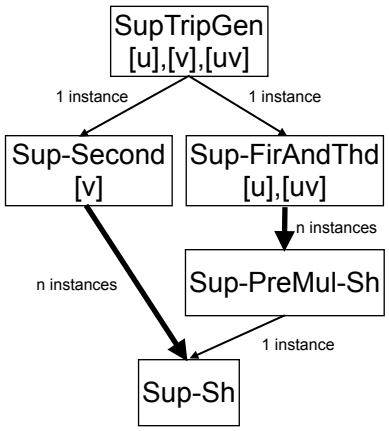

Figure 1: **Multiplication Triple Generation under Supervision of a P_{king}**

whose Sup-PreMul-Sh instances will be eventually terminated by all the parties. For $u = \sum_{P_i \in \mathcal{T}_u} u^{(i)}$ and $w = \sum_{P_i \in \mathcal{T}_u} u^{(i)} \cdot v$, the triple (u, v, w) is a random multiplication triple.

There is, however, an important subtlety: As a *precondition*, the Sup-PreMul-Sh protocol expects its dealer D to also have *encryptions* of all n shares of $[v]$, encrypted under the individual keys of the respective share-holders; here, the encryption scheme is additively homomorphic (and not threshold additively homomorphic). For any sharing, we call a party having such n encrypted shares to be *privileged*. Due to asynchronicity, the Sup-Sh protocol cannot guarantee that *all* the $n - t$ honest parties are privileged with respect to every $[v^{(i)}]$ sharing of $P_i \in \mathcal{T}_v$. We solve the problem by ensuring in Sup-Sh that there exists a designated (possibly corrupted) *supervisor* P_{king} (called *king*), who is a privileged party with respect to *each* $[v^{(i)}]$. An honest P_{king} then computes all the n encrypted shares of $[v]$ using the homomorphic properties of encryption, and reliably broadcasts those encrypted shares. Using non-equivocation, the required asynchronous reliable broadcast is possible for $n > t$ (§3.3). Once P_{king} (correctly) broadcasts the n encrypted shares of v, then each P_i can invoke its Sup-PreMul-Sh instance.

The resultant wrapper protocols are called Sup-Second and Sup-FirAndThd, where Sup-Second generates $[v]$ under the supervision of P_{king} and Sup-FirAndThd generates $[u]$ and $[w = u \cdot v]$ under the supervision of P_{king}. A combination of Sup-Second and Sup-FirAndThd leads to the protocol SupTripGen under the supervision of a designated P_{king}, which outputs a uniformly random and private multiplication triple $([u], [v], [w])$.

Preprocessing Phase Protocol. Protocol SupTripGen may not terminate for a *corrupted* P_{king}. Therefore, we ask each party P_i to act as a king and generate shared random multiplication triples under its supervision by invoking an instance of SupTripGen. As the instances of honest kings will eventually terminate, we distribute the load of generating $c_M + c_R$ shared random multiplication triples among n parties. Each party P_i is asked to act as a king and generate $\frac{c_M + c_R}{t+1}$ shared multiplication triples in its SupTripGen instance. The parties then agree on a common subset $\mathcal{T}_{\mathsf{king}}$ of $t + 1$ kings whose SupTripGen instances will be eventually completed by everyone and the $|\mathcal{T}_{\mathsf{king}}| \cdot \frac{c_M + c_R}{t+1} = c_M + c_R$ shared triples obtained in these instances are considered as the final output.

2.2 Important Sub-protocols for the Preprocessing Phase

We now discuss the realization of the main sub-protocols Sup-Sh and Sup-PreMul-Sh for the preprocessing phase.

Protocol Sup-Sh (§**4.1**)**.** Our Sup-Sh protocol is almost equivalent to the AVSS primitive [CKAS02,Can96, BKP11]: it allows a *dealer* D to "verifiably" share a secret s, thus generating $[s]$, and ensures that at least *one* honest party is privileged to obtain all the n shares encrypted for the respective share holders. The existing computational AVSS protocols (e.g., [CKAS02, BKP11]) are designed with $n = 3t + 1$ and are based on sharing a secret using a bivariate polynomial of degree t in each variable and (homomorphic) commitments. In this paradigm, it is ensured that D has distributed "consistent" shares to $n-t = 2t+1$ parties such that (at least) $t + 1$ *honest* parties among them can "enable" the remaining parties to get their shares. Unfortunately, this approach cannot be used with $n = 2t + 1$, as here we can only ensure that D has distributed consistent shares to $n - t = t + 1$ parties. In the worst case, there will be *only one* honest party in this set, who does not have sufficient information to help the other honest parties to complete a sharing.

We solve this problem by introducing encryptions of the shares[3], and by employing univariate polynomials instead of bivariate polynomials. Here, D provides a vector of n encrypted shares as well as homomorphic commitments of those shares to each party. The non-equivocation mechanism is used to ensure that a *corrupted* D does not distribute different sets of encrypted and committed shares to the different parties. Once $n - t = t + 1$ parties confirm that they have received "consistent" n encrypted and committed shares, there must exist at least one honest privileged party with all n encrypted shares, who can transfer the individual encrypted shares to the individual parties. Transferability of non-equivocation ensures that corrupted privileged parties do not transfer incorrect encryptions.

Protocol Sup-PreMul-Sh (§**4.2**)**.** The protocol takes as input an existing sharing $[v]$ of a value v unknown to everybody including $\mathcal{A}$, such that *all* the parties are privileged, i.e., all the parties hold encryptions of all shares. The protocol then allows a dealer D to verifiably share its value u as well as $u \cdot v$ (i.e. $[u]$ and $[u \cdot v]$). The protocol ensures that $u \cdot v$ remains secure in general and u is secure for an *honest* D. The idea behind the protocol is that knowing the encrypted and committed shares of v and employing the homomorphic properties of encryptions and commitments, D can compute the encrypted and committed shares corresponding to $u \cdot v$ for his choice of u, even without knowing v. The dealer can then (non-equivocally) distribute the encrypted and committed shares to the parties. Once it is confirmed that $t + 1$ parties have received all the n encrypted and committed shares of $u \cdot v$, it is ensured that there exists a honest privileged party, who can relay the individual encrypted shares of $u \cdot v$ to the respective parties.

We take a more bottom-up approach in the rest of the paper. We describe our model, and define non-equivocation and other primitives in §3. We start our construction with subprotocols Sup-Sh and Sup-PreMul-Sh in §4. We then present our supervised multiplication triple generation in §5 and finally describe the complete AMPC protocol in §6.

3 Preliminaries

In this section, we discuss our adversary and communication model, define the AMPC protocol and the non-equivocation mechanism, and describe the required primitives.

3.1 Model

We consider a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of n parties connected by pairwise authenticated channels, where $n = 2t + 1$. These communication channels are asynchronous with arbitrary but finite delay (i.e. the messages reach their destinations eventually). A centralized static adversary $\mathcal{A}$ can actively corrupt any t

[3]We argue that the problem is inherently not solvable for $n = 2t + 1$ with only commitments usually employed in computationally secure VSS protocols [CKAS02, BKP11], and that we have to employ encryptions which allow a *single honest* party to procure encrypted shares of all the parties. Interestingly, the problem persists even when we assume the adversary $\mathcal{A}$ is only passive (but crashable), not Byzantine.

out of the n parties and force them to deviate in any arbitrary manner. A party not under the control of $\mathcal{A}$ is called *honest*. The adversary $\mathcal{A}$ is modeled as a probabilistic polynomial time (PPT) algorithm, with respect to a security parameter κ. During a protocol execution, the message delivery order is decided by a *scheduler* controlled by $\mathcal{A}$. Nevertheless, the scheduler cannot modify the messages exchanged between honest parties. A protocol execution is considered as a sequence of *atomic* steps, where a single party is active in each such step. A party is activated upon receiving a message, after which it performs some computation and possibly outputs messages on its outgoing links. The scheduler controls the order of these atomic steps. At the beginning of the execution, each party will be in a special *start* state. A party is said to *terminate/complete* the execution if it reaches a *halt* state. A protocol execution is said to *be complete* when all honest parties complete it. We assume that every message sent by a party during an execution has a publicly known unique *identifier (key)* associated with it. By $[y, z]$ we denote the set $\{y, y+1, \ldots, z\} \subset \mathbb{N}$.

3.2 Definitions

Computationally Secure AMPC. We briefly review computationally secure AMPC here, and refer the readers to [HNP05, HNP08] for a formal definition. Informally, in an AMPC protocol Π_{MPC}, every party first provides its input in $\mathbb{Z}_p$ to the computation (in a secure fashion). Due to the asynchronous nature of communication, the parties cannot wait to consider the inputs of all n parties, and instead they agree on inputs from a set $\mathcal{CORE}$ of $n - t$ parties. The parties then compute an "approximation" of f on the inputs from the $\mathcal{CORE}$ set and assuming a default value (say 0) as the remaining t inputs. For every possible $\mathcal{A}$ and for all possible inputs and random coins of the (honest) parties, we expect the following properties for a Π_{MPC} instance, except with a negligible probability (in κ):

- *Termination:* all the honest parties eventually terminate Π_{MPC};
- *Correctness:* the honest parties obtain the correct output of the function f;
- *Privacy:* the adversary $\mathcal{A}$ obtains no additional information about the inputs of the honest parties other than what may be inferred from the inputs and outputs of the corrupted parties.

The above properties are formalized to the standard simulation-based definition following the real-world/ideal-world paradigm [Can96, BOCG93, HNP05, HNP08].

(Transferable) Non-equivocation. Non-equivocation restricts a corrupted party from making conflicting statements to different parties, and it has been used in several asynchronous distributed systems [CMSK07, LDLM09, CVL10, CJKR12, KBC+12] to improve their resiliency. In particular, these systems employ transferable non-equivocation, which (similar to digital signatures) allows a party to verifiably transfer a non-equivocation tag (or signature) provided by a sender to other parties. Clement et al. [CJKR12] justify the necessity of transferability of non-equivocation by proving that non-equivocation or signature alone are powerless in *asynchronous* distributed systems. Nevertheless, (transferable) non-equivocation has not been formalized so far, and we present a simplified, idealized definition for transferable non-equivocation.

In Figure 2, we define a simplified mechanism (Neq) which is intended to model the event for a transferable non-equivocation instantiation: (1) during the setup phase, every party P_i gets associated with a unique non-equivocation list L_i characterized by its index i, and all parties are informed about this association. (2) The list owner party can create a non-equivocation signature for any key-message pair except that she cannot equivocate and obtain a signature for the *same* key twice. (3) Given a key-message-signature triplet associated with a sender, any party successfully verifies only correctly generated signatures except with a negligible probability.

The Neq mechanism

Neq is parameterized by a polynomial $p(\cdot)$, and an implicit security parameter κ.

SETUP: Upon receiving a (Setup) message from party $P_i \in \mathcal{P}$, do:

1. If a list L_i exists, return $\perp$.
2. Otherwise, create an empty list L_i of key-message-signature triplets of type $\{0,1\}^* \times \{0,1\}^* \times \{0,1\}^{p(\kappa)}$. Send a message (Registered, P_i) to all parties in $\mathcal{P}$.

SIGNING: Upon receiving an (Neq-Sign, P_i, ℓ, m) message from party P_i, do:

1. If the list L_i does not exist *or* $\langle \ell, \_, \_ \rangle \in L_i$, return $\perp$.
2. Otherwise, choose an arbitrary (signature) tag $\sigma_i^{\ell,m} \in \{0,1\}^{p(\kappa)}$, update $L_i \leftarrow L_i \cup \langle \ell, m, \sigma_i^{\ell,m} \rangle$ and return $\sigma_i^{\ell,m}$.

VERIFICATION: Upon receiving an (Neq-Verify, P_i, ℓ, m, σ) message from party $P_j \in \mathcal{P}$, do:

1. If the list L_i does not exist *or* $\langle \ell, m, \sigma \rangle \notin L_i$, then return 0, else return 1.

Figure 2: **A simplified transferable non-equivocation mechanism** Neq

We survey existing transferable non-equivocation instantiations and analyze their relations to Neq in Appendix A. In most instantiations, the transferable non-equivocation is implemented using an increment-only counter for keys and signatures with public key infrastructure (PKI) [CMSK07, LDLM09, CVL10, CJKR12] or message authentication codes (MACs) generated with a replicated secret key [KBC+12]. In Neq, we generalize these using the list L_i of key-message-signature triplets associated with party P_i indexed by *party-defined ordered keys* ℓ. Similar to signatures, only P_i can use Neq-Sign to add triplets (one per each key) to L_i. Similar to PKI, anybody can Neq-Verify if a triplet $\langle \ell, m, \sigma \rangle$ belongs to L_i of P_i, and verifiably transfer authentication to others. Note that the increment-only counter provides a space-efficient way to implement a list L_i as only the counter value has to be maintained and not the whole list.

For ease of exposition, we use a phrase P_i *sends* m_{σ_i} *to* P_j and P_j *receives* m_{σ_i} *from* P_i to suggest that P_i sends a triplet $\langle \ell, m, \sigma_i^{\ell,m} \rangle$ for a key ℓ to P_j and P_j delivers it only after applying Neq-Verify to check if $\sigma_i^{\ell,m}$ is obtained by P_i using Neq-Sign on ℓ and m. Similarly, we use a phrase P_j *forwards* m_{σ_i} *to* P_k to suggest that P_j received (in the above sense) message m_{σ_i} (of P_i) from some party, and then forwards it to P_k who should also (non-equivocally) receive it. Note that we avoid the keys ℓ in the above phrases as they can be pre-assigned to protocol instance-step combinations in an unambiguous manner.

3.3 Employed Primitives

We now discuss the existing primitives used in our protocols.

Homomorphic Encryptions and Commitments. We assume an IND-CPA secure *linear homomorphic encryption* scheme $(\mathsf{Enc}, \mathsf{Dec})$. Every party P_i has its own key-pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, for which the public key pk_i is known to all parties. Given two ciphertexts $\mathbf{c}_{m_1} = \mathsf{Enc}_{\mathsf{pk}_i}(m_1, \cdot)$ and $\mathbf{c}_{m_2} = \mathsf{Enc}_{\mathsf{pk}_i}(m_2, \cdot)$, we require that there exist operations $\boxplus$ and $\boxdot$ on ciphertexts such that $\mathbf{c}_{m_1} \boxplus \mathbf{c}_{m_2} = \mathsf{Enc}_{\mathsf{pk}_i}(m_1 + m_2, \cdot)$ and $a \boxdot \mathbf{c}_{m_i} = \mathsf{Enc}_{\mathsf{pk}_i}(a \cdot m_i, \cdot)$ holds. We also assume an unconditional hiding and computational binding *linear homomorphic commitment* scheme (Commit, Open) with the analogous homomorphic operations; these are denoted by $\oplus$ and $\odot$. For the sake of readability, we sometimes leave the randomness of encryptions and commitments implicit. For instantiating encryptions and commitments over messages in $\mathbb{Z}_p$, we use the encoding-free additive El-Gamal encryption scheme [CMPP06] and Pedersen commitment scheme [Ped91] respectively.

Zero-knowledge (ZK) Proofs. We assume the presence of the following two-party ZK protocols.

1) Zero knowledge proof of equality of encrypted and committed values (PoE)**.** There exists a prover $\mathsf{P} \in \mathcal{P}$ who computes and publishes a commitment $\mathsf{Com}_m = \mathsf{Commit}(m, r)$, and ciphertexts $\mathbf{c}_m =$

$\mathsf{Enc}_{\mathsf{pk}_i}(m,\cdot)$ and $\mathbf{c}_r = \mathsf{Enc}_{\mathsf{pk}_i}(r,\cdot)$. Then using PoE, the prover P can prove to *any* verifier $\mathsf{V} \in \mathcal{P}$ (knowing $\mathsf{Com}_m, \mathbf{c}_m, \mathbf{c}_r$ and pk_i) that the message encrypted in $\mathbf{c}_m$ is also committed in Com_m, under the randomness encrypted in $\mathbf{c}_r$; i.e.,

$$\exists m, r, r_1, r_2 : \mathsf{Com}_m = \mathsf{Commit}(m,r) \quad \wedge \quad \mathbf{c}_m = \mathsf{Enc}_{\mathsf{pk}_i}(m, r_1) \quad \wedge \quad \mathbf{c}_r = \mathsf{Enc}_{\mathsf{pk}_i}(r, r_2).$$

2) Zero knowledge proof of correct pre-multiplication (PoCM). Given publicly known commitments $\mathsf{Com}_{v_j} = \mathsf{Commit}(v_j, r_j)$ and corresponding ciphertexts $\mathbf{c}_{v_j} = \mathsf{Enc}_{\mathsf{pk}_j}(v_j,\cdot), \mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j,\cdot)$ for $j \in [1,n]$, there exists a prover $\mathsf{P} \in \mathcal{P}$ who selects a random $u \in Z_p$ and t-degree random polynomials $m(\cdot)$ and $\hat{m}(\cdot)$ in $Z_p[x]$ with $m(0) = \hat{m}(0) = 0$. Let $m_j = m(j)$ and $\hat{m}_j = \hat{m}(j)$ for $j \in [0,n]$. In addition, P publishes $\mathsf{Com}_u = \mathsf{Commit}(u,\cdot)$ and $\mathsf{Com}_{m_j} = \mathsf{Commit}(m_j, \hat{m}_j)$. Using the homomorphic property of commitments and encryptions, P computes and publishes the commitment $\mathsf{Com}_{u\cdot v_j + m_j} = \mathsf{Commit}(u \cdot v_j + m_j, u \cdot r_j + \hat{m}_j)$ and encryptions $\mathbf{c}_{u\cdot v_j + m_j}$ and $\mathbf{c}_{u\cdot r_j + \hat{m}_j}$ of $u \cdot v_j + m_j$ and $u \cdot r_j + \hat{m}_j$ respectively. Then using PoCM, P can prove to *any* verifier $\mathsf{V} \in \mathcal{P}$ that all values $\mathsf{Com}_{u\cdot v_j + m_j}$ were generated by multiplying Com_{v_j} with the same u followed by re-randomization using the same $m(\cdot)$ and $\hat{m}(\cdot)$ polynomials, and that all $\mathbf{c}_{u\cdot v_j + m_j}$ and $\mathbf{c}_{u\cdot r_j + \hat{m}_j}$ values were generated by multiplying $\mathbf{c}_{v_j}$ and $\mathbf{c}_{r_j}$ respectively with the same u, followed by re-randomization using the same $m(\cdot)$ and $\hat{m}(\cdot)$ respectively. i.e.,

$$\begin{aligned}\exists u,\rho, m(\cdot), \hat{m}(\cdot), \{k_j, \hat{k}_j\}_{j\in[1,n]} : \mathsf{Com}_u = \mathsf{Commit}(u,\rho) \ \wedge \ \mathsf{deg}(m(\cdot)) \le t \ \wedge \ \mathsf{deg}(\hat{m}(\cdot)) \le t \ \wedge \\ m(0) = 0 = \hat{m}(0) \ \wedge \ \mathsf{Com}_{u\cdot v_j + m_j} = u \odot \mathsf{Com}_{v_j} \oplus \mathsf{Commit}(m_j, \hat{m}_j) \ \wedge \\ \mathbf{c}_{u\cdot v_j + m_j} = u \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(m_j, k_j) \ \wedge \mathbf{c}_{u\cdot r_j + \hat{m}_j} = u \boxdot \mathbf{c}_{r_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(\hat{m}_j, \hat{k}_j).\end{aligned}$$

Both the ZK protocols are based on standard Σ-protocols [BG92] and have communication complexity $\mathcal{O}(\kappa)$ bits and $\mathcal{O}(n\kappa)$ bits respectively. See Appendix B for their instantiations based on the ZK protocols in [CS97].

Certificates of Claims. Hirt, Nielsen, and Przydatek [HNP08] introduced this concept to allow a prover $\mathsf{P} \in \mathcal{P}$ to *publicly* prove correctness of a certain claim (like real-life certificates), without revealing any additional information. Here, to certify validity of a statement m, the prover P proves m to every *verifier* $P_i \in \mathcal{P}$ using an appropriate zero-knowledge (ZK) protocol. A verifier P_i, upon successful verification, sends a signature to P on an "appropriate" message (known publicly), corresponding to m. P cannot wait for all n signatures in the asynchronous environment; thus, upon receiving $(n-t) = t+1$ signatures, the prover P concatenates them to construct a certificate α for the claim m. These $t+1$ signatures ensure that at least one honest party has verified the claim, and that m is true with an overwhelming probability. Assuming each signature to be of size $\mathcal{O}(\kappa)$ bits, the size of α will be $\mathcal{O}(n\kappa)$ bits; this can be reduced to $\mathcal{O}(\kappa)$ bits using a *threshold signature scheme* with threshold t [BTHN10]. Here, the verifiers send signature *shares* and the prover P, instead of concatenating, combines $(n-t)$ shares to a single signature.[4]

Let zkp be the ZK protocol corresponding to the claim m. We denote the task of constructing a certificate α for m as $\alpha = \mathsf{certify}_{\mathsf{zkp}}(m)$. Similarly we say that "P_i verifies the certificate α for the claim m" to mean that that P_i verifies that α is a valid (threshold) signature on the appropriate message corresponding to m. The communication cost of constructing α is the same as that of executing n instances of the corresponding ZK protocol zkp.

Reliable Broadcast (r-broadcast). This asynchronous primitive [Bra84, Tou84, HT94] allows a sender S to send a message m identically to *all* the parties $\in \mathcal{P}$: For a given instance τ_b of r-broadcast, when S is *honest*, all honest parties eventually terminate with output (τ_b, m); if S is *corrupted* and some honest party terminates with (τ_b, m'), then every honest party also eventually terminates with (τ_b, m'); for any

[4]Note that the AMPC protocols of [HNP08, BTHN10] also assume a threshold signature setup.

instance, at most one message can be delivered by an honest party. The resiliency bound for r-broadcast is $n \geq 3t+1$ [Bra84, Tou84, HT94]; however, assuming transferable non-equivocation, an r-broadcast protocol with $n \geq t+1$ and $\mathcal{O}(n^2(\ell+\kappa))$ bits of communication for broadcasting an ℓ-bit message is available [CJKR12, CVL10]; see Appendix B. In the rest of the paper, a terminology "P_i *broadcasts* m" means that P_i as a sender invokes an r-broadcast instance for m. Similarly, "P_j *receives* m *from the broadcast of* P_i" means that P_j terminates the r-broadcast instance τ_b invoked by P_i with the output (τ_b, m).

Agreement on a Common Subset (ACS). This primitive allows the parties to agree on a common subset of $(n-t)$ parties, who correctly invoked some protocol, say Π, satisfying the following requirements: (a) If an honest party invokes an instance of Π then all the (honest) parties eventually terminate the instance; (b) If some honest party terminates an instance of Π invoked by a corrupted party, then every honest party eventually does the same. ACS can be realized by executing n instances (one for each party) of an asynchronous Byzantine agreement (ABA) protocol to decide if it should be included in the common subset. Assuming transferable non-equivocation, ABA, and hence ACS, can be implemented with $n \geq 2t+1$ [CJKR12, CVL10, KBC+12]. An efficient ACS protocol with expected communication complexity of $\mathcal{O}(n^3\kappa)$ bits can be obtained by using the Neq mechanism in the multi-valued ABA of [CKPS01].

3.4 Secret Sharing Notations

Given a secret $s \in \mathbb{Z}_p$, let $\phi(\cdot), \psi(\cdot) \in \mathbb{Z}_p[x]$ be, respectively, a degree t *sharing polynomial* with $\phi(0) = s$ and a t-degree *randomness polynomial* required for commitments; here, p is a κ-bit prime. For party P_j, $s_j = \phi(j)$ and $r_j = \psi(j)$ are respectively her shares of s and the randomness polynomial. Let $\mathbf{c}_{s_j} = \mathsf{Enc}_{\mathsf{pk}_j}(s_j, \cdot)$, $\mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j, \cdot)$ and $\mathsf{Com}_{s_j} = \mathsf{Commit}(s_j, r_j)$ and let $\mathsf{Com}_s = \mathsf{Commit}(s, \psi(0))$. We call $\{\mathbf{c}_{s_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}$ the encrypted shares and $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}$ the committed shares of s.

$[\cdot]$-sharing: A secret s is said to be $[\cdot]$-shared, if every (honest) party $P_i \in \mathcal{P}$ holds s_i, r_i, $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}$ and Com_s. The information held by the (honest) parties corresponding to $[\cdot]$-sharing of s is denoted as $[s]$.

Privileged party: P_i is called a privileged party of $[\cdot]$-sharing of s if it holds the encrypted shares $\{\mathbf{c}_{s_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}$.

Due to linearity of sharing and commitments, $[\cdot]$-sharing is also *linear*: given $[a]$, $[b]$ and a public constant c, every party can locally compute its information corresponding to $[a+b]$ and $[c \cdot a]$, as $[a+b] = [a] + [b]$ and $[c \cdot a] = c \cdot [a]$ resepctively.

4 Supervised Sharing Protocols

We present two protocols for generating $[\cdot]$-sharings with different properties under the supervision of a king P_{king}. Here, if the protocols terminate, then an *honest* P_{king} will be a *privileged* party with respect to the generated sharings.

4.1 Protocol Sup-Sh: Supervised $[\cdot]$-sharing

Protocol Sup-Sh (Figure 3) allows a *dealer* D to verifiably generate $[s]$ for a secret $s \in \mathbb{Z}_p$ under the supervision of $P_{\mathsf{king}} \in \mathcal{P}$. It ensures that if the protocol terminates then there exists a value (say $\overline{s}$) which will be $[\cdot]$-shared among the parties; if D is *honest* then $\overline{s} = s$, and $\mathcal{A}$ learns no new information on s from the protocol execution. Moreover, if P_{king} is *honest* then it will be a privileged party. The protocol always terminates for an *honest* D and P_{king} and has communication complexity $\mathcal{O}(n^2\kappa)$. In the protocol

Protocol $\mathsf{Sup\text{-}Sh}(\mathsf{D}, \tau, P_{\mathsf{king}}, s)$: τ is the session id

I. D-DEPENDENT PHASE:

SHARE COMPUTATION AND CERTIFICATE GENERATION—Given secret $s \in \mathbb{Z}_p$, D executes the following code:

1. Select t-degree polynomials $\phi(\cdot), \psi(\cdot) \in \mathbb{Z}_p[x]$ with $\phi(0) = s$. For $j \in [1, n]$, compute *share-pairs* $s_j = \phi(j), r_j = \psi(j)$, ciphertexts $\mathbf{c}_{s_j} = \mathsf{Enc}_{\mathsf{pk}_j}(s_j), \mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j)$ and *committed share* $\mathsf{Com}_{s_j} = \mathsf{Commit}(s_j, r_j)$. Compute $\mathsf{Com}_s = \mathsf{Commit}(s, \psi(0))$.
2. Send messages $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$, and $\{\mathsf{Com}_s\}_{\sigma_\mathsf{D}}$ to each P_i.
Start constructing a certificate $\alpha^{\mathsf{D},\tau} = \mathsf{certify}_{\mathsf{PoE}}(\mathsf{claim}_{\mathsf{D},\tau})$ claiming that "D has correctly shared a secret in session τ": This indirectly requires a proof that "$\exists s_j, r_j, \rho_{j,1}, \rho_{j,2}$ for $j \in [1, n]$: $\mathbf{c}_{s_j} = \mathsf{Enc}_{\mathsf{pk}_j}(s_j, \rho_{j,1}) \wedge \mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j, \rho_{j,2}) \wedge \mathsf{Com}_{s_j} = \mathsf{Commit}(s_j, r_j) \wedge \{\exists\, t$-degree polynomials $\phi(\cdot), \psi(\cdot)$ such that $s_j = \phi(j)$, $r_j = \psi(j)$, and $\mathsf{Com}_s = \mathsf{Commit}(\phi(0), \psi(0))\}$"; for this, run an instance of PoE for each $(\mathbf{c}_{s_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{s_j})$ triplet with every party P_i. Broadcast $\alpha^{\mathsf{D},\tau}$ once it is constructed.

SHARE VERIFICATION AND CERTIFICATION—Every party $P_i \in \mathcal{P}$ executes the following code:

1. Upon receiving $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$, and $\{\mathsf{Com}_s\}_{\sigma_\mathsf{D}}$ from D, perform the following verifications:
   (a) Verify if $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}$ and Com_s define unique t-degree polynomials (using the Vandermonde matrix properties [GRR98, CKAS02]).
   (b) If the above verification is successful, then participate in the PoE instances of D to verify $\mathsf{claim}_{\mathsf{D},\tau}$ and enable D to construct a certificate $\alpha^{\mathsf{D},\tau}$ for $\mathsf{claim}_{\mathsf{D},\tau}$. **If $P_i = P_{\mathsf{king}}$, upon successful PoE verifications, broadcast a message $(\mathtt{OK}, \mathsf{D})$.**

II. D-INDEPENDENT PHASE AND TERMINATION—Every party $P_i \in \mathcal{P}$ executes the following code:

1. Upon receiving the broadcasted certificate $\alpha^{\mathsf{D},\tau}$ from D **and the message** $(\mathtt{OK}, \mathsf{D})$ **from the broadcast of** P_{king}, verify $\alpha^{\mathsf{D},\tau}$ for $\mathsf{claim}_{\mathsf{D},\tau}$. Upon successful verification, if $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$ have been received from D,
   (a) then compute $s_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{s_i}), r_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{r_i})$, and $\forall j \in [1, n]$ forward *only* $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$, and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ to P_j,
   (b) else wait for $\{\mathbf{c}_{s_i}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_i}\}_{\sigma_\mathsf{D}}$, and $\{\mathsf{Com}_{s_i}\}_{\sigma_\mathsf{D}}$ to be forwarded by some party. Once received, compute the share-pair $s_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{s_i})$ and $r_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{r_i})$.
2. Forward $\{\mathsf{Com}_{s_i}\}_{\sigma_\mathsf{D}}$ to each P_j. On receiving $t + 1$ $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$, homomorphically compute $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}, \mathsf{Com}_s$ and terminate.

Figure 3: **Protocol for a dealer D to generate $[s]$ under P_{king}**

pseudocode, a certain part is in boldface; in §4.1.1, we will argue that removing this part of the code leads to an asynchronous verifiable secret sharing (AVSS) scheme.

In the Sup-Sh protocol, D polynomial-shares s with threshold t to generate the shares $\{s_j\}_{j\in[1,n]}$ and computes the commitments $\mathsf{Com}_{s_j} = \mathsf{Commit}(s_j, r_j)$, where r_j is the share of a random t-degree randomness polynomial $\psi(\cdot)$. It also computes the encryptions $\mathbf{c}_{s_j} = \mathsf{Enc}_{\mathsf{pk}_j}(s_j)$ and $\mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j)$ of each share-pair s_j, r_j, and the commitment $\mathsf{Com}_s = \mathsf{Commit}(s, \psi(0))$. D then (non-equivocally) *sends* $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$, and $\{\mathsf{Com}_s\}_{\sigma_\mathsf{D}}$ to *every* party and claims that it has correctly $[\cdot]$-shared a secret: this involves proving that the plaintexts in $\mathbf{c}_{s_j}$ and $\mathbf{c}_{r_j}$ are committed in Com_{s_j} and that the values committed in $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}$ constitute shares of the secret committed in Com_s with threshold t. Notice that sending the full vector $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$, and $\{\mathsf{Com}_s\}_{\sigma_\mathsf{D}}$ to each party is quite non-intuitive; however, as discussed in §2, it is the *crux* of our Sup-Sh protocol to ensure that every party eventually receives its shares for $[s]$.

To verify D's claim, upon (non-equivocally) receiving information from D, every P_i homomorphically

uses the Vandermonde matrix properties [CKAS02,GRR98] to check if the committed shares $\{\mathsf{Com}_{s_j}\}_{j\in[1,n]}$ constitute valid Shamir sharing of the secret committed in Com_s with threshold t. Additionally P_i engages in n instances of PoE (one per triplet $\{\mathbf{c}_{s_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{s_j}\}$) with D. By non-equivocally sending messages to the parties, it is ensured that the parties who receive the messages from D, receive the *same* messages. D then constructs a certificate $\alpha^{\mathsf{D},\tau}$ (for session id τ) to support its claim of correct sharing and broadcasts it. A party proceeds further only upon receiving a valid certificate. A valid $\alpha^{\mathsf{D},\tau}$ ensures that at least one honest party, say P_h, has verified D's claim; P_h will be an *honest privileged party*. To satisfy our requirement that even P_{king} is a privileged party, we additionally enforce that every party should also receive an acknowledgement from P_{king} of having verified the claim of D before proceeding further.

A valid certificate from D and an acknowledgement from P_{king} does not ensure that every (honest) P_i holds its information corresponding to $[s]$: due to asynchrony or possible corrupted behavior of D, t honest parties may not receive their shares corresponding to $[s]$. We solve this problem by using *two* additional "rounds" of communication, which we call the D INDEPENDENT PHASE. Every *privileged* party first (non-equivocally) *forwards only* $\{\mathbf{c}_{s_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_j}\}_{\sigma_\mathsf{D}}$, and $\{\mathsf{Com}_{s_j}\}_{\sigma_\mathsf{D}}$ to every party P_j, who can decrypt $\mathbf{c}_{s_j}$ and $\mathbf{c}_{r_j}$ to obtain s_j, r_j. Existence of at least one honest privileged party ensures that every P_j eventually receives s_j, r_j and Com_{s_j}. Next, every P_i forwards $\{\mathsf{Com}_{s_i}\}_{\sigma_\mathsf{D}}$ to all parties. As all $t+1$ honest parties would eventually receive their respective $\{\mathbf{c}_{s_i}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{r_i}\}_{\sigma_\mathsf{D}}$, and $\{\mathsf{Com}_{s_i}\}_{\sigma_\mathsf{D}}$ messages at the end of first "round" of the D INDEPENDENT PHASE, eventually every honest party will receive $t+1$ forwarded committed shares. Now using the homomorphic property of commitments every party can compute the remaining committed shares and Com_s, thus possessing all the necessary information of $[s]$. Notice that the above communication "pattern" is crucial towards maintaining the communication complexity of this phase to $\mathcal{O}(n^2\kappa)$; i.e. using commitments to the sharing polynomial coefficients (as done traditionally in VSS schemes), instead of commitments to the shares, would incur an additional $\Theta(n)$ communication overhead as the non-equivocation mechanism is not homomorphic in nature. The properties of Sup-Sh are proved in Appendix C.1.

4.1.1 Designing AVSS through a Variant of Sup-Sh

A closer look at Sup-Sh reveals that if P_{king} is *corrupted*, then the protocol may not terminate even if D is *honest*: the *corrupted* P_{king} may not broadcast the required $(\mathtt{OK}, \mathsf{D})$ message. Thus, Sup-Sh fails to qualify as an AVSS sharing protocol, which needs to terminate for an honest D [BKP11, CKAS02]. However, we can easily get rid of this problem by removing the requirement for P_{king} to broadcast the $(\mathtt{OK}, \mathsf{D})$ message. During the D-INDEPENDENT PHASE of the resultant (Sh) protocol, every party waits *only* to receive a valid certificate from the broadcast of D. Sh will allow (an honest) D to generate $[s]$, and its pseudocode can be obtained by removing the instructions in boldface in Figure 3.

Given $[s]$ generated using Sh, the standard reconstruction (Rec) protocol used in the existing computationally secure VSS [Ped91, BKP11, CKAS02] will allow the parties to *robustly* reconstruct s. In the protocol, each party sends its share pair to all the parties, which are verified with the corresponding commitment, available with the parties (as part of $[s]$). Once $t+1$ "correct" share pairs are received, the sharing polynomial, and hence s, is reconstructed. As there exist at least $t+1$ honest parties whose shares will eventually be communicated among themselves, the Rec protocol eventually terminates. For the formal details, see [Ped91,BKP11,CKAS02]. The pair of protocols (Sh, Rec) constitutes an AVSS scheme with $n = 2t+1$ and communication complexity $\mathcal{O}(n^2\kappa)$ bits. Note that the scheme is also publicly verifiable [Sta96] as any third party can verify the consistency of the shares using the valid certificate broadcasted by D.

4.2 Supervised Pre-multiplication Protocol

The Sup-PreMul-Sh protocol (Figure 4) takes input a $[\cdot]$-shared uniformally random and private value v and allows a dealer $\mathsf{D} \in \mathcal{P}$ to verifiably generate $[u]$ as well as $[u \cdot v]$ for a value u of his choice, under the

Protocol $\mathsf{Sup\text{-}PreMul\text{-}Sh}(\mathsf{D}, \tau, P_{\mathsf{king}}, \mathcal{P}, [v])$: τ is the session id

Let $\{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{v_j} = \mathsf{Commit}(v_j, r_j)\}_{j\in[1,n]}$ and $\mathsf{Com}_v = \mathsf{Commit}(v, \cdot)$ be information corresponding to $[v]$ available to *all* parties; each P_i also has the share-pair v_i, r_i.

I. GENERATING $[u]$: On having a value u, D invokes an instance $\mathsf{Sup\text{-}Sh}(\mathsf{D}, \tau, P_{\mathsf{king}}, u)$ of Sup-Sh to generate $[u]$ under the supervision of P_{king} and every party in $\mathcal{P}$ participates in this instance and wait for its termination. Let $\mathsf{Com}_u = \mathsf{Commit}(u, \cdot)$ be the commitment of u which is computed and communicated during this instance of Sup-Sh (along with the other information corresponding to $[u]$).

II. GENERATING $[u \cdot v]$—The following code is executed by every party only upon terminating the $[u]$ instance:

a. D DEPENDENT PHASE

1. SHARE COMPUTATION AND CERTIFICATE GENERATION—The following code is executed only by D:
   (a) Select random *masking polynomials* $m(\cdot)$ and $\hat{m}(\cdot)$ of degree at most t with $m(0) = \hat{m}(0) = 0$. For $j \in [1, n]$, compute $m_j = m(j), \hat{m}_j = \hat{m}(j), \mathsf{Com}_{m_j} = \mathsf{Commit}(m_j, \hat{m}_j)$ and $\mathsf{Com}_m = \mathsf{Commit}(m(0), \hat{m}(0))$.
   (b) For $j \in [1, n]$, using the homomorphic property of the encryption and commitment scheme, compute the *new encrypted share pair* $\mathbf{c}_{u\cdot v_j+m_j} = u \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(m_j, \cdot), \mathbf{c}_{u\cdot r_j+\hat{m}_j} = u \boxdot \mathbf{c}_{r_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(\hat{m}_j, \cdot)$ and the *new committed share* $\mathsf{Com}_{u\cdot v_j+m_j} = u \odot \mathsf{Com}_{v_j} \oplus \mathsf{Com}_{m_j}$. In addition, compute the *new commitment* $\mathsf{Com}_{u\cdot v} = u \odot \mathsf{Com}_v \oplus \mathsf{Com}_m$.
   (c) Send messages $\{\mathbf{c}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}, \{\mathbf{c}_{u\cdot r_j+\hat{m}_j}\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{m_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$ and $\{\mathsf{Com}_u\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_m\}_{\sigma_\mathsf{D}}$ to each P_i. Start constructing a certificate $\beta^{\mathsf{D},\tau} = \mathsf{certify}_{\mathsf{PoCM}}(\mathsf{claim}_{\mathsf{D},\tau})$ claiming that "D has correctly done the pre-multiplication in session τ." This claim indirectly requires the following proof:

$$\exists u, \rho, m(\cdot), \hat{m}(\cdot), \{k_j, \hat{k}_j\}_{j\in[1,n]} : \mathsf{Com}_u = \mathsf{Commit}(u, \rho) \ \wedge \ \mathsf{deg}(m(\cdot)) \le t \ \wedge \ \mathsf{deg}(\hat{m}(\cdot)) \le t \ \wedge$$
$$m(0) = 0 = \hat{m}(0) \ \wedge \ \mathsf{Com}_{u\cdot v_j+m_j} = u \odot \mathsf{Com}_{v_j} \oplus \mathsf{Com}_{m_j} \wedge$$
$$\mathbf{c}_{u\cdot v_j+m_j} = u \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(m_j, k_j) \wedge \mathbf{c}_{u\cdot r_j+\hat{m}_j} = u \boxdot \mathbf{c}_{r_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(\hat{m}_j, \hat{k}_j)$$

   For this, run an instance of PoCM for $(\{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{v_j}, \mathbf{c}_{u\cdot v_j+m_j}, \mathbf{c}_{u\cdot r_j+\hat{m}_j}, \mathsf{Com}_{u\cdot v_j+m_j}, \mathsf{Com}_{m_j}\}_{j\in[1,n]}, \mathsf{Com}_u, \mathsf{Com}_m)$ with every party P_i. Broadcast the certificate $\beta^{\mathsf{D},\tau}$ once it is constructed.
2. SHARE VERIFICATION AND CERTIFICATION—Every $P_i \in \mathcal{P}$ upon receiving $\{\mathbf{c}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{u\cdot r_j+\hat{m}_j}\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_{m_j}\}_{\sigma_\mathsf{D}}$ for all $j \in [1, n]$ and $\{\mathsf{Com}_u\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v}\}_{\sigma_\mathsf{D}}$ and $\{\mathsf{Com}_m\}_{\sigma_\mathsf{D}}$ from D, participate (as a V) in the instance of PoCM with D to verify the claim $\mathsf{claim}_{\mathsf{D},\tau}$ and enable D to construct the certificate $\beta^{\mathsf{D},\tau}$ for $\mathsf{claim}_{\mathsf{D},\tau}$. Upon successful verification, broadcast the message $(\mathtt{approve}, \mathsf{D})$ if $P_i = P_{\mathsf{king}}$.

b. D INDEPENDENT PHASE AND TERMINATION—Every party $P_i \in \mathcal{P}$ executes the following code:

1. Upon receiving the broadcasted certificate $\beta^{\mathsf{D},\tau}$ from D and the broadcasted message $(\mathtt{approve}, \mathsf{D})$ from P_{king}, verify $\beta^{\mathsf{D},\tau}$ for $\mathsf{claim}_{\mathsf{D},\tau}$. Upon successful verification:
   (a) If $\forall j \in [1, n], \{\mathbf{c}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{u\cdot r_j+\hat{m}_j}\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$ have been received from D, then compute $w_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{u\cdot v_i+m_i}), \hat{r}_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{u\cdot r_i+\hat{m}_i})$, and forward *only* $\{\mathbf{c}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{u\cdot r_j+\hat{m}_j}\}_{\sigma_\mathsf{D}}, \{\mathsf{Com}_{u\cdot v_j+m_j}\}_{\sigma_\mathsf{D}}$ to each P_j,
   (b) else wait for $\{\mathbf{c}_{u\cdot v_i+m_i}\}_{\sigma_\mathsf{D}}$, $\{\mathbf{c}_{u\cdot r_i+\hat{m}_i}\}_{\sigma_\mathsf{D}}$, and $\{\mathsf{Com}_{u\cdot v_i+m_i}\}_{\sigma_\mathsf{D}}$ to be forwarded by some party. Once received, compute the share-pair $w_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{u\cdot v_i+m_i})$ and $\hat{r}_i = \mathsf{Dec}_{\mathsf{sk}_i}(\mathbf{c}_{u\cdot r_i+\hat{m}_i})$.
2. Forward $\{\mathsf{Com}_{u\cdot v_i+m_i}\}_{\sigma_\mathsf{D}}$ to every P_j. On receiving $t + 1$ $\{\mathsf{Com}_{u\cdot v_i+m_i}\}_{\sigma_\mathsf{D}}$ messages, homomorphically compute $\{\mathsf{Com}_{u\cdot v_j+m_j}\}_{j\in[1,n]}, \mathsf{Com}_{u\cdot v}$ and terminate.

Figure 4: **Protocol for generating $[u]$ and $[u \cdot v]$ under P_{king}**

supervision of a designated $P_{\mathsf{king}} \in \mathcal{P}$. As a *pre-condition*, the protocol assumes that every (honest) party is a privileged party with respect to the input $[v]$. The protocol ensures that v and $u \cdot v$ remains private, and when D is *honest* then $\mathcal{A}$ learns no new information on u. Finally if P_{king} is *honest* then it will be a

privileged party with respect to $[u]$ as well as $[u \cdot v]$. The protocol always terminates for an *honest* D and P_{king} and has communication complexity $\mathcal{O}(n^2\kappa)$ bits.

Let $\{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{v_j}\}_{j\in[1,n]}$ and Com_v be the encrypted shares and the committed shares corresponding to $[v]$ that is available to all the parties. Let $\phi(\cdot)$ and $\psi(\cdot)$ be the sharing and randomness polynomial corresponding to $[v]$. Thus $v_i = \phi(i)$, $r_i = \psi(i)$ is the share-pair available with party P_i and $\mathbf{c}_{v_j} = \mathsf{Enc}_{\mathsf{pk}_j}(v_j), \mathbf{c}_{r_j} = \mathsf{Enc}_{\mathsf{pk}_j}(r_j), \mathsf{Com}_{v_j} = \mathsf{Commit}(v_j, r_j)$ and $\mathsf{Com}_v = \mathsf{Commit}(\phi(0), \psi(0))$. To generate $[u]$, D first invokes an instance of $\mathsf{Sup\text{-}Sh}$. The next task for D would be to generate $[u \cdot v]$ and that without knowing v. To do this, we observe that $u \cdot \phi(\cdot) + m(\cdot)$ and $u \cdot \psi(\cdot) + \hat{m}(\cdot)$ constitue correct sharing and randomness polynomial respectively for $[u \cdot v]$, where $m(\cdot)$ and $\hat{m}(\cdot)$ are random *masking polynomials* of degree at most t selected by D with the constraint $m(0) = \hat{m}(0) = 0$. This is because $u \cdot \phi(\cdot) + m(\cdot)$ and $u \cdot \psi(\cdot) + \hat{m}(\cdot)$ will have degree at most t, with the constant term of $u \cdot \phi(\cdot) + m(\cdot)$ being $u \cdot v$. Thus $w_j = u \cdot v_j + m(j)$ and $\hat{r}_j = u \cdot r_j + \hat{m}(j)$ constitute valid share-pair for $[u \cdot v]$ and so by using the homomorphic property of encryption and commitment, D can compute the encrypted shares and committed shares of $[u \cdot v]$ and distribute the same to the parties; the presence of masking molynomials preserves the privacy of u and $u \cdot v$. The rest of the protocol is now similar to $\mathsf{Sup\text{-}Sh}$, except that PoCM is used instead of PoE by D to construct the certificate that it has done "correct pre-multiplication". As nothing about the masking polynomials is revealed, u remains private; see Appendix C.2 for the properties of the protocol.

5 Supervised Triple Generation and the NeqAMPC Protocol

Protocol $\mathsf{SupTripGen}$ generates $[u], [v], [w]$ for a uniformly random and private multiplication triple (u, v, w) under the supervision of a king P_{king} with $\mathcal{O}(n^3\kappa)$ communication complexity. It employs two sub-protocols, $\mathsf{Sup\text{-}Second}$ and $\mathsf{Sup\text{-}FirAndThd}$. We only include informal description of the protocols here; for details see Appendix D.

5.1 Generating the Second Component of the Triple

Protocol $\mathsf{Sup\text{-}Second}$ generates $[v]$ for a uniformly random value v, unknown to $\mathcal{A}$, under the supervision of P_{king}. For a requirement clarified in the sequel, $\mathsf{Sup\text{-}Second}$ also ensures that each (honest) party becomes a privileged party with respect to $[v]$. In the protocol, each P_i invokes an instance $\mathsf{Sup\text{-}Sh}_i$ of $\mathsf{Sup\text{-}Sh}$ (as a D) to generate $[v^{(i)}]$ for a uniformly random $v^{(i)}$. Let $\mathcal{T}_{\mathsf{king}}$ be the set of first $t+1$ parties whose instance of $\mathsf{Sup\text{-}Sh}$ is terminated by P_{king}, and let $v = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} v^{(k)}$. As at least one party in $\mathcal{T}_{\mathsf{king}}$ is honest, v is uniformly random and private. Next, P_{king} broadcasts $\mathcal{T}_{\mathsf{king}}$ and every party waits until it terminates all $\mathsf{Sup\text{-}Sh}_k$ instances of $P_k \in \mathcal{T}_{\mathsf{king}}$; this ensures that every party obtains its share-pair and all committed shares of v.

As P_{king} is a privileged party for *every* $[v^{(k)}]$, it computes all encrypted shares of $[v]$, using the homomorphic property of encryptions. However, unlike P_{king}, other honest parties may be non-privileged for one or more $[v^{(k)}]$s, and thus may not compute the encrypted shares of $[v]$. The way out is that P_{king} "helps" other parties by broadcasting the encrypted shares of v, which costs $\mathcal{O}(n^3\kappa)$ bits. To confirm whether P_{king} indeed broadcasted the correct encrypted shares of v, P_{king} is also asked to non-equivocally forward the encrypted shares corresponding to each $[v^{(k)}]$ to every other party. We stress that this information is *not* broadcasted, and rather communicated over the *point-to-point* channels, which costs $\mathcal{O}(n^3\kappa)$ bits. Once this information is *non-equivocally received* by a party, it *re-computes* the encrypted shares of v, verifies them with the P_{king}'s broadcast, and broadcasts the verification result. If $t+1$ parties broadcasts "positively" for P_{king}, then at least one honest party must have successfully verified those encrypted shares; so every party terminates the protocol with $[v]$ and the broadcasted encrypted shares of $[v]$.

5.2 Generating First and Third Components of the Triple

Protocol Sup-FirAndThd takes as input a $[\cdot]$-shared uniformly random, say v, unknown to $\mathcal{A}$ such that *every party is a privileged party for* $[v]$. It then generates $[u]$ for a uniformly random u, unknown to $\mathcal{A}$, along with the $[\cdot]$-sharing $[u \cdot v]$, under the supervision of P_{king}. The protocol follows the same principle as Sup-Second, except that each party P_i now invokes an instance Sup-PreMul-Sh$_i$ of Sup-PreMul-Sh with $[v]$ and a uniformly random value $u^{(i)}$ to generate $[u^{(i)}]$ and $[u^{(i)} \cdot v]$; this is possible as every P_i is a privileged party with respect to $[v]$. Now the parties set $u = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} u^{(k)}$ and $w = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} u^{(k)} \cdot v$, where $\mathcal{T}_{\mathsf{king}}$ is the set of $t+1$ parties P_k such that the instance Sup-PreMul-Sh$_k$ has been terminated by P_{king}.

Finally protocol SupTripGen consists of two steps: **(1).** The parties execute Sup-Second and output $[v]$; **(2).** On terminating Sup-Second, the parties execute Sup-FirAndThd, output $[u]$ and $[w] = [u \cdot v]$ and terminate.

6 The NeqAMPC Protocol

The key idea of the NeqAMPC protocol has already been discussed in §2. It is a sequence of three phases, where a party proceeds to the next phase only after completing the current phase:

Preprocessing Phase. To generate $c_M + c_R$ $[\cdot]$-shared random multiplication triples, the preprocessing phase protocol PreProcess performs the following steps: each party $P_i \in \mathcal{P}$ is asked to act as a king and invoke $\frac{c_M+c_R}{t+1}$ *parallel* instances of SupTripGen to generate $\frac{c_M+c_R}{t+1}$ random $[\cdot]$-shared multiplication triples under its supervision. The parties then execute an instance of ACS and agree on a common subset $\mathcal{T}_{\mathsf{king}}$ of $(n-t) = t+1$ kings whose instances of SupTripGen (as a king) will eventually be terminated by all the parties. The parties finally output the shared multiplication triples, generated during the instances of SupTripGen, corresponding to the kings in $\mathcal{T}_{\mathsf{king}}$ and terminate; thus they will obtain $|\mathcal{T}_{\mathsf{king}}| \cdot \frac{c_M+c_R}{t+1} = c_M + c_R$ shared multiplication triples. As there exist at least $t+1$ honest parties, whose instances of SupTripGen as a king will be eventually terminated by all the (honest) parties (see Lemma D.3), protocol PreProcess will eventually terminate. Similarly, as the shared triples generated in the instances of SupTripGen corresponding to each king in $\mathcal{T}_{\mathsf{king}}$ remain private, the output shared triples remain private. It is easy to see that PreProcess has communication complexity $\mathcal{O}(n \cdot \frac{c_M+c_R}{t+1} \cdot n^3\kappa) = \mathcal{O}((c_M + c_R)n^3\kappa)$ bits as $t = \Theta(n)$. As the protocol is quite straightforward, we skip the formal details.

Input Phase. The goal of the input phase protocol Input is to allow each individual party $P_i \in \mathcal{P}$ to generate $[\cdot]$-sharing of its private input x_i for the computation. For this each party $P_i \in \mathcal{P}$ invokes an instance of the sharing protocol Sh (see section. 4.1.1) as D to generate $[x_i]$. To avoid indefinite waiting, the parties execute an instance of ACS and agree on a common subset of $(n-t)$ parties, say $\mathcal{CORE}$, whose instances of Sh (as a dealer) will eventually be terminated by all the parties. The parties finally output the sharings, generated during the instances of Sh, corresponding to the parties in $\mathcal{CORE}$; on behalf of the remaining parties in $\mathcal{P} \setminus \mathcal{CORE}$, a default $[\cdot]$-sharing of 0 is considered. As there exist at least $t+1$ honest parties, whose instances of Sh as a dealer will eventually be terminated by all the (honest) parties, protocol Input will eventually terminate. The shared inputs generated in the instances of Sh corresponding to the *honest* parties in $\mathcal{CORE}$ remain private due to the privacy property of Sh. The Input protocol runs n instances of Sh, and has communication complexity of $\mathcal{O}(n \cdot n^2\kappa) = \mathcal{O}(n^3\kappa)$ bits. Again as the protocol is quite straight-forward, we skip the formal details.

Computation Phase. The computation phase protocol Compute performs the shared circuit evaluation on a gate-by-gate basis, by maintaining the following *invariant* for each gate of the circuit: given the $[\cdot]$-sharing of the input(s) of a gate, the protocol allows the parties to securely compute the $[\cdot]$-sharing of the output

of the gate. The invariant is trivially maintained for the addition (linear) gates in the circuit, thanks to the lineaity property of $[\cdot]$-sharings. For a multiplication gate, the invariant is maintained by applying the Beaver's circuit randomization technique and using a $[\cdot]$-shared multiplication triple from the pre-processing stage (recall from section 2). For a random gate, a $[\cdot]$-shared multiplication triple from the pre-processing stage is considered and the first component of the triple is associated with the random gate. Finally, once the $[\cdot]$-sharing $[y]$ of the circuit output y is generated, the parties execute the reconstruction protocol Rec, reconstruct y and terminate.

Again as the protocol is quite standard in the literature (see for example [CHP13]), we omit the complete details and state only the main theorem here.

Theorem 6.1 (The NeqAMPC Theorem). *Let $f : \mathbb{Z}_p^n \rightarrow \mathbb{Z}_p$ be a function expressed as an arithmetic circuit over $\mathbb{Z}_p$, consisting of c_M multiplication gates and c_R random gates. Assume a non-equivocation oracle associated with every party. Then for every possible $\mathcal{A}$ and for every possible scheduler, there exists a computationally secure AMPC protocol to securely compute f with communication complexity $\mathcal{O}(((c_M + c_R) \cdot n^3 + n^3)\kappa)$ bits.*

Acknowledgments. We would like to thank Martin Hirt for his suggestions on an earlier draft of the paper. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the German Universities Excellence Initiative.

References

[BDK13] M. Backes, A. Datta, and A. Kate. Asynchronous Computational VSS with Reduced Communication Complexity. In *Proc. of CT-RSA'13*, pages 259–276, 2013.

[Bea91] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *Proc. of CRYPTO'91*, pages 420–432, 1991.

[BG92] M. Bellare and O. Goldreich. On Defining Proofs of Knowledge. In *Proc. of CRYPTO'92*, pages 390–420, 1992.

[BKP11] M. Backes, A. Kate, and A. Patra. Computational Verifiable Secret Sharing Revisited. In *Proc. of ASIACRYPT'11*, pages 590–609, 2011.

[BOCG93] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous Secure Computation. In *Proc. of STOC'93*, pages 52–61, 1993.

[BOGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proc. of STOC'88*, pages 1–10, 1988.

[BOKR94] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous Secure Computations with Optimal Resilience (Extended Abstract). In *Proc. of PODC'94*, pages 183–192, 1994.

[Bra84] G. Bracha. An Asynchronous [(n-1)/3]-Resilient Consensus Protocol. In *Proc. of PODC'84*, pages 154–162, 1984.

[BSFO12] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In *Proc. of CRYPTO'12*, pages 663–680, 2012.

[BTH06] Z. Beerliová-Trubíniová and M. Hirt. Efficient Multi-party Computation with Dispute Control. In *Proc. of TCC'06*, pages 305–328, 2006.

[BTH07] Z. Beerliová-Trubíniová and M. Hirt. Simple and Efficient Perfectly-Secure Asynchronous MPC. In *Proc. of ASIACRYPT'07*, pages 376–392, 2007.

[BTH08] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-Secure MPC with Linear Communication Complexity. In *Proc. of TCC'08*, pages 213–230, 2008.

[BTHN10] Z. Beerliová-Trubíniová, M. Hirt, and J. B. Nielsen. On the Theoretical Gap between Synchronous and Asynchronous MPC Protocols. In *Proc. of PODC'10*, pages 211–218, 2010.

[Can96] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, The Weizmann Institute of Science, 1996.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols. In *Proc. of STOC'88*, pages 11–19, 1988.

[CDN01] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Proc. of EUROCRYPT'01*, pages 280–299, 2001.

[CHP13] A. Choudhury, M. Hirt, and A. Patra. Asynchronous Multiparty Computation with Linear Communication Complexity. In *Proc. of DISC'13*, pages 388–402, 2013.

[CJKR12] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. On the (limited) Power of Non-Equivocation. In *Proc. of PODC'12*, pages 301–308, 2012.

[CKAS02] C. Cachin, K. Kursawe, A.Lysyanskaya, and R. Strobl. Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems. In *Proc. of CCS'02*, pages 88–97, 2002.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *CRYPTO*, pages 524–541, 2001.

[CMPP06] B. Chevallier-Mames, P. Paillier, and D. Pointcheval. Encoding-Free Elgamal Encryption Without Random Oracles. In *Proc. of PKC'06*, pages 91–104, 2006.

[CMSK07] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-only Memory: Making Adversaries Stick to their Word. In *Proc. of SOSP'07*, pages 189–204, 2007.

[CS97] J. Camenisch and M. Stadler. Proof Systems for General Statements about Discrete Logarithms. Technical report, 1997. 260, Dept. of Computer Science, ETH Zurich.

[CVL10] M. Correia, G. S. Veronese, and L. C. Lung. Asynchronous Byzantine Consensus with 2f+1 Processes. In *Proc. of SAC'10*, pages 475–480, 2010.

[DN07] I. Damgård and J. B. Nielsen. Scalable and Unconditionally Secure Multiparty Computation. In *Advances in Cryptology—CRYPT0*, pages 572–590, 2007.

[DS83] D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[FM85] P. Feldman and S. Micali. Byzantine Agreement in Constant Expected Time (and Trusting No One). In *FOCS*, pages 267–276, 1985.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proc. of STOC '87*, pages 218–229, 1987.

[GRR98] R. Gennaro, M.O. Rabin, and T. Rabin. Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography. In *Proc. of PODC'98*, pages 101–111, 1998.

[GT89] A. S. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments (Preliminary Version). In *WDAG*, pages 110–123, 1989.

[HN06] M. Hirt and J. B. Nielsen. Robust Multiparty Computation with Linear Communication Complexity. In *Proc. of CRYPTO'06*, pages 463–482, 2006.

[HNP05] M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *Proc. of EUROCRYPT'05*, pages 322–340, 2005.

[HNP08] M. Hirt, J. B. Nielsen, and B. Przydatek. Asynchronous Multi-Party Computation with Quadratic Communication. In *Proc. of ICALP'08*, pages 473–485, 2008.

[HT94] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Ithaca, NY, USA, 1994.

[JMS12] A. Jaffe, T. Moscibroda, and S. Sen. On the Price of Equivocation in Byzantine Agreement. In *Proc. of PODC'12*, pages 309–318, 2012.

[KBC+12] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proc. of EuroSys'12*, pages 295–308, 2012.

[LDLM09] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proc. of NSDI'09*, pages 1–14, 2009.

[LSP82] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MLQ+10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[PCR09] A. Patra, A. Choudhary, and C. Pandu Rangan. Efficient Asynchronous Byzantine Agreement with Optimal Resilience. In *Proc. of PODC'09*, pages 92–101, 2009.

[Ped91] T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Proc. of CRYPTO'91*, pages 129–140, 1991.

[RBO89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In *Proc. of STOC'89*, pages 73–85, 1989.

[Sha79] A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

[SR00] K. Srinathan and C. Pandu Rangan. Efficient Asynchronous Secure Multiparty Distributed Computation. In *Proc. of INDOCRYPT'00*, pages 117–129, 2000.

[Sta96] M. Stadler. Publicly Verifiable Secret Sharing. In *Proc. of EUROCRYPT'96*, pages 190–199, 1996.

[Tou84] S. Toueg. Randomized Byzantine Agreements. In *Proc. of PODC'84*, pages 163–178, 1984.

[Yao82] A.C. Yao. Protocols for secure computations. In *Proc. of FOCS'82*, pages 160–164. IEEE, 1982.

A Non-equivocation Implementations

Chun et al. [CMSK07] observed that the fundamental distributed computing problem of "Byzantine generals" has been proved unsolvable for three parties when one of those is corrupted [LSP82] precisely because the corrupted party can spread contradictory messages to the remaining two honest parties. They demonstrated that if we can stop the corrupted party from *equivocating* (i.e., making conflicting statements to different honest parties) using a small trusted module on every party, it is possible to improve the resilience of distributed computing tasks in the asynchronous setting. They implemented non-equivocation using a (signed) trusted log abstraction called Attested Append-Only Memory (A2M), and designed a Byzantine-tolerant state machine replication (SMR) system for $n \geq 2t + 1$.

Levin et al. [LDLM09] further simplified the trust assumption from [CMSK07] and showed that a minimal trusted module called TrInc consisting of only a non-decreasing counter $c \in \mathbb{N}$ and a signing key-pair $(\mathsf{pk}, \mathsf{sk})$ is sufficient to generate A2M logs and to implement SMR with $n \geq 2t + 1$. Conceptually, TrInc provides a unique, once-in-a-lifetime attestations, and implements non-equivocation using the fact that the counter cannot be decreased, and consequently for every counter value c there is at most one message signed by the module.

Levin et al. implemented TrInc on Gemalto .NET SmartCards. It is also possible to implement TrInc over the computers enabled with TPM chips, where its features of trusted identity, sealed storage, and remote code attestation will be used. Although the TPM specification does not readily implement a trusted counter, it can be achieved using a TPM-based hypervisor framework such as TrustVisor [MLQ+10]. Recently, Kapitza et al. [KBC+12] further simplified the TrInc design by replacing individual singing key-pairs with a replicated message authentication code (MAC) key.

Clement el al. [CJKR12] observed that the *definitional* non-equivocation itself actually does not provide any improvement to the resiliency bound; however, appropriately combining it with digital signature oracles (or MAC oracles with a key replicated across all oracles) provides the improvements observed in [CMSK07, LDLM09, CVL10, KBC+12], where the transferability of verifications provided by signatures is a key along with non-equivocation.[5] They further noted that this combination (i.e., transferable non-equivocation) also provides a generic transformation that allows a crash fault tolerant protocol to tolerate the same number of Byzantine faults. Nevertheless, their generic transformation does not consider privacy (or confidentiality), required in the AVSS and AMPC tasks, and we observe in this paper that encryptions and zero-knowledge proofs are required along with signatures when privacy is required.

A.1 Realizing the Neq mechanism using TrInc

As discussed earlier, the Neq mechanism as described in Figure 2 can easily be realized using a TrInc implementation [LDLM09] (a simplified version of a TrInc device's state and API can be found in Figure 5). Here, we provide an informal proof for the realization.

[5] They also prove that signatures themselves are also powerless.

Notation	Meaning
K_{priv}	Unique private key
K_{pub}	Public key corresponding to K_{priv}
A	Certificate of the device's validitiy
id	Identity of the current counter (Initially 0)
c	Current value of the counter

(a) State of a device

Function	Operation
CreateCounter()	Increases id. Sets c to 0 and returns id.
Attest(id,c',h)	Verifies that id is a valid counter with some value c and key K_{pub}. Verifies that $c \leq c'$. If any verfiication fails, return $\perp$, otherwise create an attestation $a =< \mathsf{COUNTER}, id, c, c', h >_{K_{priv}}$. Set $c = c'$. Return a.
getCertificate()	Returns (K_{pub}, A)

(b) API for the device

Figure 5: The API and the state of a simplified TrInc device.

We will use the fixed counter identity 1 for our protocols as we need only one TrInc instance for each party. In addition, we need to assume that the protocol steps are mapped to the natural numbers such that the order of these steps is preserved.

1. Implementation of a (Setup) invocation by party P_i: Invoke CreateCounter() on TrInc. Let i be the return value of the invocation. If $i \neq 1$ return $\perp$. Store the value oldL $= 0$. Invoke TrInc on GetCertificate() and let crt be the result. Send (Registered, P_i, crt) to all parties. Upon receiving the registered message, every party checks the validity of the crt certificate.

2. Implementation of a (Neq-Sign, P_i, l, m) invocation by party P_i: Invoke Attest on TrInc with the arguments $(1, l, m)$. Let a be the result. If $a = \perp$ return $\perp$, otherwise return $t = (a, \mathsf{oldL}, l)$ and update oldL $= l$.

3. Implementation of a (Neq-Verify, P_i, l, m, σ) invocation by party P_j: If the message (Registered, P_i, crt) was not received by P_j, return 0. Otherwise split crt into (K, A) and σ into (s, oldL, l'). If $l' \neq l$ return 0, otherwise compute a signature verification on message (COUNTER, 1, oldL, l, m) with signature s and verification key K and return the outcome of the verification.

In the beginning every counter has internal value 0 stored. The method CreateCounter() increases that value and returns it, as the counter for that identity is created now (with counter value 0). Therefore, the method always returns a value greater than 1 after the first invocation and consequently the setup returns $\perp$ as done by the Neq mechanism. For the first invocation, the setup sends the value (Registered, P_i, crt) where crt is necessary for the other methods to work. In particular, the checking of crt ensures that the certificate belongs to a TrInc device assuming that the manufacturer was honest and did not sign other devices or keys of this form with the same key.

The signing request (Neq-Sign, P_i, l, m) from P_i is done by invoking her TrInc device on Attest$(1, l, m)$. The Attest method checks that 1 is a valid counter, i.e., if the setup was not done before it outputs $\perp$ which leads to the signing request returning $\perp$. The same holds for the Neq mechanism since $\perp$ is returned in case that the internal list L_i was not generated (by the setup) before. The next step of the attestation process

is that the key (or label) l – which we map to an integer – is greater than the previous key value.[6] This check corresponds to the check that l is in the list L_i done by the Neq mechanism. And in the mechanism as in the implementation, a fail of the check leads to the outcome $\perp$. If all checks so far succeeded, the mechanism returns a random value and the TrInc implementation returns a triple t consisting of a signature a, the previous key oldL and l.

Finally, consider the implementation of the verification query. The mechanism returns 0 if the setup was not done before. The implementation does the same, however, since the Registered message can be forged, there is a different reason here. Since the setup creates the counter, there can only be a signature on any message $(\mathsf{COUNTER}, 1, \_, \_, \_)$ if the counter was created. This is implied by the unforgeability of the signature scheme. Combined with the requirement that the Registered message has to arrive beforehand, this implies that the setup method was executed (in particular the setup checks that the Registered message contains a valid TrInc certificate, i.e., there was no other "virtual" counter implemented). In the mechanism's case that $(l, m, \sigma) \in L_i$, i.e., the output 1 is done, the signing request was done before giving the corresponding output since there is no other way to add elements to the list L_i except the signing queries. Consequently, in the TrInc implementation's case, σ has the corresponding form and the signature verification succeeds leading to an output of 1, as well. If the mechanism outputs 0 because $(l, m, \sigma) \notin L_i$ that means that m was never signed with key l by TrInc in the implementation. Thus the implementation outputs only 1 if the signature was forged, i.e., only with negligible probability.

As we have seen, the TrInc usage as described above implements the Neq mechanism with the restriction that the keys are ordered and the assumptions that the signature scheme require and that the TrInc manufacturer can be trusted. Finally, we note that it is easy to realize our Neq mechanism with the modified TrInc mechanism by Kapitza et al. [KBC+12] in the similar manner.

B Instantiation of Various Primitives

In this section we instantiate the primitives we used in our protocol construction. These are the following: commitment scheme, encryption scheme and zero-knowledge proofs. As commitment scheme we simply use Pedersen commitments [Ped91], i.e., we commit to m using randomness r by computing $g^m h^r$ for two generators g, h of a suitable group. In particular, this commitment scheme has the properties we required in section 3.3.

B.1 Encryption scheme Enc

We use the encoding-free ElGamal encryption scheme proposed in [CMPP06]. Let p, q be primes such that $q \mid p-1$ and let g be an integer of order pq modulo p^2 that generates a group $\mathbb{G} = \langle g \rangle$. Let $\langle x, y \rangle$ be the unique integer in $\mathbb{Z}_{pq}$ such that $\langle x, y \rangle = x \mod p$ and $\langle x, y \rangle = y \mod q$. The class of an element of $w = g^{\langle x,y \rangle} \in \mathbb{G}$ is x. We denote the class of w as $[\![w]\!]$. It is easy to see that $[\![w \cdot w']\!] = [\![w]\!] + [\![w']\!]$ and that $\mathbb{G}_p := \langle g \mod p \rangle$ has order q.

Definition B.1 (Encoding-Free ElGamal [CMPP06])**.**

1. Setup*: Let p, q be primes such that $q \mid p-1$ and let g be a generator of $\mathbb{G}_p$ of order q.*
2. Key generation*: The private key is a random $x \in \mathbb{Z}_q$; the public key is $h = g^x \mod p$.*
3. Encryption*: The encryption algorithm chooses randomly an $r \in \mathbb{Z}_q$ and computes*

$$\mathsf{Enc}(m, r) = (g^r \mod p, m + [\![h^r \mod p]\!] \mod p)$$

[6] This is slightly more restrictive than the Neq mechanism since we require the signature requests to be done by a certain order, but in order to use TrInc it seems to be necessary to require an order.

4. Decryption*: The decryption works as follows:*

$$\mathsf{Dec}(x,(R,c)) = c - [\![R^x \mod p]\!] \mod p$$

The scheme is CPA-secure under the Decisional Class Diffie-Hellman problem [CMPP06] which is defined as the Diffie-Hellman problem under the class operation $[\![\cdot]\!]$. It has been shown that the Computational Class Diffie-Hellman problem is equivalent to the Computational Diffie-Hellman problem. However, the same result for the decisional case has not been shown.

We define two operations on ciphertexts in order to describe their homomorphic properties.

Definition B.2 (Homomorphic operations)**.** *Let $x_1, y_1, x_2, y_2, v \in \mathbb{Z}_p$. Define the following operations:*
- $(x_1, y_1) \boxplus (x_2, y_2) := (x_1 \cdot x_2 \mod p, y_1 + y_2 \mod p)$
- $v \boxdot (x_1, y_1) := (x_1^v \mod p, v \cdot y_1 \mod p)$

Note that $[\![w]\!] + [\![w']\!] = [\![w \cdot w']\!]$ and that the latter operation can also be done by iteratively applying the first operation.

Lemma B.3 (Homomorphic operations)**.** *The operations defined in Definition B.2 implement the following operations on the plaintexts: Let $a, b, c, d, v \in \mathbb{Z}_q$.*
- $\mathsf{Enc}(a,b) \boxplus \mathsf{Enc}(c,d) = (g^{b+d} \mod p, (a+c) + [\![h^{b+d} \mod p]\!] \mod p)$
- $v \boxdot \mathsf{Enc}(a,b) = (g^{vb} \mod p, va + [\![h^{vb} \mod p]\!] \mod p)$

Proof. The proof is straight-foreward considering definition B.1. □

The encryption scheme has the properties required in section 3.3. For more details, we refer to [CMPP06].

B.2 Zero-knowledge Proof Schemes

In this subsection we will present a proof scheme using the primitivies instantiated previously. The structure of the both ZK protocols is based on Σ-protocols [BG92]. These protocols have been well-studied and are usually easy to understand. Intuitively, a Σ-protocol is a proof that a party knows a witness w for a statement x such that $(x, w) \in \mathcal{R}$. The relation $\mathcal{R}$, which can be proven, is specific for the Σ protocol.

Definition B.4 (Σ-Protocol [BG92])**.** *Let $\mathcal{R}$ be a relation. A Sigma Protocol for a relation $\mathcal{R}$ is a 3-round protocol, i.e., it consists of four algorithms (P_1, P_2, V_1, V_2) where P_1, P_2 is for the prover and V_1, V_2 is for the verifier such that the following holds. Let x, w be bitstrings and $(a, s_P) := P_1(x, w)$, $(c, s_V) := V_1(x, a)$, $p := P_2(c, s_P)$ and $d := V_2(s_V, p)$. Then the following holds:*

1. Completeness*: If $(x, w) \in \mathcal{R}$ then a protocol run using P_1, P_2, V_1, V_2 leads to $d = 1$.*
2. Special soundness*: There is an extraction algorithm E such that for any fixed statement x and for any two transcripts (a, c, p) and (a, c', p') such that the V_2 outputs 1 for both and where $c \neq c'$ holds, it follows that $(x, \mathsf{E}(a, c, c', p, p')) \in \mathcal{R}$.*
3. Special honest verifier zero-knowledge*: There is a simulator S such that for any x for which there is a w such that $(x, w) \in \mathcal{R}$ holds, the simulator produces on input x and random input c a transcript (a, c, p) which is computationally indistinguishable from a protocol transcript generated during the execution of the protocol using P_1, P_2, V_1, V_2.*

ZK Proofs technique: hiding the representation with respect to a fixed set of generators. We briefly recall this technique which was described in [CS97].

For a set of generators $\{g_i\}$ we prove that we know a set $\{x_i\}$ such that $y = \prod_i g_i^{x_i}$ by randomly choosing r_i values and sending $t = \prod_i g_i^{r_i}$ to the verifier. The verifier then sends a challenge c and the prover computes $s_i := r_i - cx_i$ and sends the s_i to the verifier. The verifier accepts iff $t = y^c \prod_i g_i^{s_i}$.

B.2.1 ZK-Proof Scheme PoE

The relation for PoE we need to prove is the following:

$$\exists m, r, r_1, r_2 : \mathsf{Com}_m = \mathsf{Commit}(m, r) \wedge \mathbf{c}_m = \mathsf{Enc}(m, r_1) \wedge \mathbf{c}_r = \mathsf{Enc}(r, r_2)$$

Using the instantiations we get the statement:

$$\exists m, r, r_1, r_2 : \mathsf{Com}_m = g^m h^r \wedge (\mathbf{c}_{m,1}, \mathbf{c}_{m,2}) = (g^{r_1}, m + [\![h^{r_1}]\!]) \wedge (\mathbf{c}_{r,1}, \mathbf{c}_{r,2}) = (g^{r_2}, r + [\![h^{r_2}]\!])$$

In order to prove the equations for Com_m and $\mathbf{c}_{m,1}, \mathbf{c}_{r,1}$, we use the technique mentioned above.

For the remaining part, i.e., $\mathbf{c}_{m,2} = m + [\![h^{r_e}]\!]$ ($\mathbf{c}_{r,2}$ works analogously), we give a Σ-protocol following the idea of the one above. This is done by computing r_i and s_i as in [CS97], however, the t is computed and verified differently; this is done by computing t as $r_m + [\![h^{r_e}]\!]$ and verifying it to be $c \cdot \mathbf{c}_{m,2} + s_m + [\![h^{s_e}]\!]$. Given this construction it is straightforward to prove that the corresponding protocol is indeed a Σ-protocol.

Combining these two Σ-protocols we get a Σ-protocol for PoE.

B.2.2 Zero-knowledge Proof Scheme PoCM

As for the previous proof scheme, we will use a Σ-protocol in order to construct an interactive ZK proof scheme. The overall statement that we show is the following.

$$\begin{aligned}&\exists u, \rho, m(\cdot), \hat{m}(\cdot), \{k_j, \hat{k}_j\}_{j \in [1,n]} : \mathsf{Com}_u = \mathsf{Commit}(u, \rho) \ \wedge \ \deg(m(\cdot)) \leq t \ \wedge \ \deg(\hat{m}(\cdot)) \leq t \ \wedge \\ &\quad m(0) = 0 = \hat{m}(0) \ \wedge \ \mathsf{Com}_{u \cdot v_j + m_j} = u \odot \mathsf{Com}_{v_j} \oplus \mathsf{Com}_{m_j} \ \wedge \ \mathbf{c}_{u \cdot v_j + m_j} = u \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(m_j, k_j) \\ &\quad \wedge \mathbf{c}_{u \cdot r_j + \hat{m}_j} = u \boxdot \mathbf{c}_{r_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_j}(\hat{m}_j, \hat{k}_j)\end{aligned}$$

We need to mention here that it is unintuitive that $\mathbf{c}_{r_j}$ is multiplied by u instead of r. However, this is correct since we need to maintain the property that every party knows how to open their share, i.e., $\mathsf{Com}_{u \cdot v_j + m_j}$ which is derived from Com_{v_j} by exponentiating by u. Hence not only the value of v_j is multiplied by u but also the value of the corresponding randomness. This is also the reason why we need to rerandomize by adding Com_{m_j}; if we would not a party could try out wheter u has some particular value u' by exponentiating the value Com_{v_j} by u' and equality-checking with the new value. Now, we show how to achieve a zero-knowledge construction for the overall statement.

The check for the polynomial degree can be done using the representation problem, e.g., in order to check that variables x and y satisfy the equation $2x + 5y = 1$ we can check that we know the representation of g^1 with respect to the base $\{g^2, g^5\}$. Consequently, we can check the degree by evaluating the (inverse) Vandermonde matrix on the quantified values and comparing the result with constants.

However, in order to give an instantiation with respect to the previously defined instantiations of Com and Enc we need to existentially quantify over the corresponding randomnesses as well. In addition, we split the proof into two statements which can be combined into a proof for the conjunction using standard techniques, i.e., using the same r_x, s_x for shared variables x.

The first relation can be proven using results described in [CS97]. We want to prove the knowledge of a representation of the commitments — with respect to the base of the Pedersen commitments — Com_u and Com_{m_j} together with an additional verification step to verify the property we require for $\mathsf{Com}_{u \cdot v_j + m_j}$. This additional check can be rephrased as follows:

$$X := \mathsf{Com}_{u \cdot v_j + m_j} \ominus \mathsf{Com}_{m_j} = (g^{v_j} h^{r_{v_j}})^u$$

where r_{v_j} is the randomness used to construct Com_{v_j} which is unknown to the prover.

Protocol r-broadcast(τ_b, m)

SENDER S (WITH INPUT m) —S executes the following code:

1. Send (τ_b, m_{σ_S}) to every party in $\mathcal{P}$.

PARTY P_i — Every party in $\mathcal{P}$, including S, executes this code:

1. If (τ_b, m_{σ_S}) is received from S or forwarded by some party $P_j \in \mathcal{P}$ (for the first time), then forward (τ_b, m_{σ_S}) to every party in $\mathcal{P}$, output (τ_b, m) and terminate.

Figure 6: Asynchronous r-broadcast protocol using transferable non-equivocation for $t < n$ [CVL10, CJKR12]

Hence, we need to verify that we know the representation of X with respect to the base Com_{v_j} (and that this is the same as for the representation of Com_u). Therefore we can simply use the proof scheme [CS97], as for PoE, for the first part of the scheme.

However, it is not obvious that we can use this technique in order to verify $\mathbf{c}_{u \cdot v_j + m_j} = u \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_i}(m_j, k_j)$. Looking at the first component of our instantiation of Enc leads to an equation of the form $X = Y^u \cdot g^{k_j}$ which corresponds to the representation knowledge problem, since the X is represented via the base $\{Y, g\}$ using u and k_j. Hence we can use the same technique. We need to show the correspondence for the second part as well. This part looks like

$$X = uY + m_j + k_j \cdot Z$$

here Y is second part of $\mathbf{c}_{v_j}$ and Z is $[\![h]\!]$. This is a linear version of the base representation problem and the technique can be applied here as well. The corresponding t constructed using randomness r_1, r_2, r_3 of this equation is $r_1 \boxdot \mathbf{c}_{v_j} \boxplus \mathsf{Enc}_{\mathsf{pk}_i}(r_2, r_3)$.

B.3 Asynchronous Reliable Broadcast (r-broadcast) Using Transferable Non-equivocation

The r-broadcast primitive allows a $\mathsf{S} \in \mathcal{P}$ to reliably send a message m to all the parties. Using transferable non-equivocation, it can achieved for $t < n$ [CVL10, CJKR12]. The high-level idea of this r-broadcast protocol is simple, and it follows from the crash-fault tolerant r-broadcast [GT89]: S first (non-equivocally) sends m_{σ_S} to all the parties; this prevents a corrupted S from sending different messages to different honest parties. However, a corrupted S can avoid sending m_{σ_S} to some honest parties. Thus, to ensure that all the honest parties eventually receive m_{σ_S}, whenever an honest party (non-equivocally) receives some message m_{σ_S}, before delivering the message, it once non-equivocally forwards it to every other party. This ensures that whenever an honest party receives m_{σ_S} then it will be eventually received by every other honest party. We present the pseudocode for the protocol r-broadcast in Figure 6. .
The properties of the protocol are stated in Theorem B.5, which follows easily from the protocol description and the properties of transferable non-equivocation.

Theorem B.5. *Protocol* r-broadcast *achieves the following properties for every instance τ_b, and every possible $\mathcal{A}$ and scheduler:*

(1) TERMINATION: *If* S *is honest, then all the honest parties eventually terminate the protocol. Moreover, even if* S *is corrupted and some honest party terminates the protocol, then except with negligible probability, every other honest party eventually terminates the protocol* . (2) CORRECTNESS: **(a)** *If* S *is honest then except with negligible probability, all honest parties output (τ_b, m).* **(b)** *If* S *is corrupted and some honest party outputs (τ_b, m'), then except with negligible probability, all the honest parties output (τ_b, m').* (3)

COMMUNICATION COMPLEXITY: *The protocol incurs communication of $\mathcal{O}(n^2(\ell+\kappa))$ bits, where the message m is of size ℓ bits.*

C Properties of the Various Supervised Sharing Protocols and Proofs

C.1 Properties of the Protocol Sup-Sh

Lemma C.1. *Let s be the D's secret. Then for every possible $\mathcal{A}$ and scheduler, protocol* Sup-Sh *achieves the following properties:* (1) TERMINATION*: if D and P_{king} are honest then all the honest parties eventually terminate the protocol, except with negligible probability. Moreover, if some honest party terminates the protocol, then every other honest party eventually does the same, except with negligible probability.* (2) CORRECTNESS*: if some honest party terminates the protocol, then there exists a value $\overline{s}$ which will eventually be $[\cdot]$-shared among the parties, except with negligible probability. Moreover, if D is honest then $\overline{s}=s$. Furthermore if P_{king} is honest then P_{king} will be a privileged party.* (3) PRIVACY*: if D is honest then s remains private.* (4) COMMUNICATION COMPLEXITY*: the protocol has communication complexity $\mathcal{O}(n^2\kappa)$ bits.*

PROOF: For TERMINATION, we first consider an honest D and P_{king}. In this case, D will (non-equivocally) send $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}, \{\mathsf{Com}_{s_j}\}_{\sigma_D}, \{\mathsf{Com}_s\}_{\sigma_D}$ to all parties. In particular all honest parties will eventually receive them and start participating in the instances of PoE, where all the verifications will pass. So P_{king} will eventually broadcast the $(\mathtt{OK}, \mathsf{D})$ message, which from the properties of r-broadcast will eventually reach every honest party with high probability. Moreover, since there exist at least $n-t=t+1$ honest parties, D will be able to construct the certificate $\alpha^{\mathsf{D},\tau}$ and eventually broadcast the same. Therefore every honest party eventually receives $\alpha^{\mathsf{D},\tau}$ as well as the $(\mathtt{OK}, \mathsf{D})$ message. Moreover, P_{king} will be a privileged party and forwards $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_D}$ to every P_j. It now follows easily that every honest P_j will eventually receive $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_D}$ from P_{king} and obtain its share pair s_j, r_j by decrypting $\mathbf{c}_{s_j}$ and $\mathbf{c}_{r_j}$. Moreover, since every such P_j forwards its $\{\mathsf{Com}_{s_j}\}_{\sigma_D}$ to every other party and there are at least $t+1$ such P_js, it follows that every honest party will eventually have at least $t+1$ committed shares with high probability, using which it will homomorphically obtain the remaining committed shares and terminate.

Now consider a corrupted D (and possibly a corrupted P_{king}) and let P_i be an honest party that terminates the protocol. We show that all other honest parties will eventually do the same. Since P_i terminated the protocol, it implies that P_i received $\alpha^{\mathsf{D},\tau}$ from the broadcast of D as well as $(\mathtt{OK}, \mathsf{D})$ from P_{king}'s broadcast. From the properties of broadcast, it follows that with high probability, every other honest party will eventually receive them. In addition, since α^{D} was constructed, at least $t+1$ and hence at least one honest party, say P_h, must have received $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}, \{\mathsf{Com}_{s_j}\}_{\sigma_D}, \{\mathsf{Com}_s\}_{\sigma_D}$ from D and successfully performed all the verifications. Since P_h is a privileged party, the rest of the proof follows using the same arguments as above, except that P_h plays the role of P_{king}.

CORRECTNESS: If some honest party, say P_i, has terminated the protocol, then it follows that it has received a valid certificate $\alpha^{\mathsf{D},\tau}$ from the broadcast of D, which implies that with overwhelming probability, there exists at least one honest party, say P_h, who would have participated in the construction of α^{D}. This further implies that P_h must have received $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}, \{\{\mathsf{Com}_{s_j}\}_{\sigma_D}\}_{j\in[1,n]}$ and $\{\mathsf{Com}_s\}_{\sigma_D}$ from D and successfully performed the required verifications. Particularly, P_h would have verified that there exist polynomials of degree at most t, say $\overline{\phi}(\cdot)$ and $\overline{\psi}(\cdot)$, such that $\mathsf{Com}_s = \mathsf{Commit}(\overline{\phi}(0), \overline{\psi}(0))$ and $\mathsf{Com}_{s_j} = \mathsf{Commit}(\overline{\phi}(j), \overline{\psi}(j))$. We define $\overline{s}$ to be $\overline{\phi}(0)$ and show that eventually $\overline{s}$ will be $[\cdot]$-shared. Since P_i has terminated the protocol, from the termination property of the protocol, it follows that each honest party will eventually terminate with its shares s_j and r_j and a vector of committed shares, so what remains

is to show that they correspond to $[\overline{\phi}(0)]$. However, this follows from the properties of non-equivocation. Specifically, neither a corrupted D nor any corrupted party can send or forward any other encrypted and committed shares, different from $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}$ and $\{\mathsf{Com}_{s_j}\}_{\sigma_D}$ respectively, to any honest P_j. Similarly, no corrupted party P_k can forward its committed share, different from $\{\mathsf{Com}_{s_k}\}_{\sigma_D}$, to any honest party. Thus with high probability, $\overline{s}$ will be $[\cdot]$-shared.

It follows easily that if D is honest then $\overline{s} = s$, as in this case the polynomials $\overline{\phi}(\cdot)$ and $\overline{\psi}(\cdot)$ are the same as $\phi(\cdot)$ and $\psi(\cdot)$, as selected by D. Moreover it follows easily that if P_{king} is honest then it will be a privilged party, since an honest P_{king} will broadcast the $(\mathtt{OK}, \mathsf{D})$ message only after receiving $\{\mathbf{c}_{s_j}\}_{\sigma_D}, \{\mathbf{c}_{r_j}\}_{\sigma_D}, \{\mathsf{Com}_{s_j}\}_{\sigma_D}, \{\mathsf{Com}_s\}_{\sigma_D}$ from D and successfully verifying it.

PRIVACY: We show that for an honest dealer D, and any $s, \overline{s}$ the adversary can not distinguish whether D shared s or $\overline{s}$. Since the non-equivocation signature $\{x\}_{\sigma_D}$ does not provide any information additional to the signed value x, we drop the tag for this part of the proof for the sake of readability. So let $\mathcal{T}_{cor}$ be the set of corrupted parties. Hence define $\mathcal{K}_{corr} := \{s_i, r_i, k_i \mid P_i \in \mathcal{T}_{corr}$, where s_i, r_i are the shares of party P_i and k_i its encryption and decryption keys$\} \cup \{sk_i \mid sk_i$ is the signing key of $P_i\}$. Note that we only assumed authentic channels; therefore, it is easy to see that the view of the adversary $\mathsf{view}_A(x)$ during the execution of the protocol with secret x consists of $\mathsf{Com}_x, \{\mathsf{Com}_{x_j} = \mathsf{Commit}(x_j, r_{x,j}), \mathbf{c}_{x_j} = \mathsf{Enc}_{\mathsf{pk}_j}(x_j, \cdot), \mathbf{c}_{r_{x,j}} = \mathsf{Enc}_{\mathsf{pk}_j}(r_{x,j}, \cdot)\}_{j\in[1,n]}$ as well as $\alpha^{\mathsf{D},\tau}, (\mathtt{OK}, \mathsf{D})$ and the messages during the protocol executions of PoE.

Assume there is an adversary A that can distinguish whether $x = s$ or $x = \overline{s}$ is shared with non-negligible probability. We then show that there is an adversary that can distinguish Com_s from $\mathsf{Com}_{\overline{s}}$ with non-negligible probability. We do this in several steps; in particular, we define the following views and show that these are indistinguishable for s and $\overline{s}$.

- $\mathsf{view}^1_A(x) := \mathsf{view}_A(x) \cup \mathcal{K}_{corr}$
- $\mathsf{view}^2_A(x) := \mathsf{Com}_x, \{\mathsf{Com}_{x_j}, \mathbf{c}_{x_j}, \mathbf{c}_{r_{x,j}}\}_{j\in[1,n]}$ and $\mathcal{K}_{corr}$
- $\mathsf{view}^3_A(x) := \mathsf{Com}_x, \{\mathsf{Com}_{x_j}\}_{j\in[1,n]}$ and $\mathcal{K}_{corr}$
- $\mathsf{view}^4_A(x) := \mathsf{Com}_x$ and $\mathcal{K}_{corr}$
- $\mathsf{view}^5_A(x) := \mathcal{K}_{corr}$

Next, we show for $i \in [1, 4]$ that $\mathsf{view}^i_A(x) \sim \mathsf{view}^{i+1}_A(x)$. For each step we need to show that if for each adversary A^i having input $\mathsf{view}^i_A(x)$, there is an adversary A^{i+1} which has an indistinguishable output on input $\mathsf{view}^{i+1}_A(x)$.[7]

1. $\mathsf{view}^1_A(x) \sim \mathsf{view}^2_A(x)$

   Let A^1 be given, A^2 internally uses A^1 by computing $(\mathtt{OK}, \mathsf{D})$, computing $\alpha^{\mathsf{D},\tau}$ and simulating the zero-knowledge proofs PoE. The first part is trivial, the second part can be done since the message signed in α can be deduced from $\mathsf{view}^2_A(x)$ and A^2 has access to all signing key shares. In order to prove the last part we need to distinguish two cases: PoE executions with honest parties and PoE executions with corrupted parties. The executions with honest parties can be computed by A^2 using the simulator of the honest-verifier zero-knowledge property, since the honest parties choose their challenge randomly. For the corrupted parties, the adversary A^2 knows their s_i. Consequently, he can act as the dealer in PoE and use the adversary of the protocol execution in order to generate these proofs.

[7] Note that the adversary even knows the signing keys of the parties.

Finally, we need to show that the output of A^1 is indistinguishable from the output of A^2. By construction of A^2 it is sufficient to show that the input given to A^1 inside A^2 is indistinguishable from the input that A^1 gets. For $(\texttt{OK}, \mathsf{D})$ and $\alpha^{\mathsf{D},\tau}$ this is obvious. For the ZK proofs of honest parties, this is implied by the honest-verifier zero-knowledge. The zero-knowledge proofs of the corrupted parties consist of messages (a, c, e). Here a has the same distribution as in $\mathsf{view}_A^1(x)$, i.e., uniformly at random. The part c has the same distribution since A^2 internally invokes the adversary of the protocol execution. Finally, e is completely determined by a and c. Therefore e has the same distribution as well.

2. $\mathsf{view}_A^2(x) \sim \mathsf{view}_A^3(x)$

   In this step we basically remove the ciphertexts from the input of A^2. We construct A^3 by internally running A^2 on $\mathsf{view}^3(x)$ and the ciphertexts computed by A^3. In order to compute the ciphertexts A^3 needs to distinguish two cases, ciphertexts of corrupted and ciphtertexts of honest parties. For corrupted parties, A^3 can simply access the plaintexts using $\mathcal{K}_{corr}$ and encrypt them as D does, hence having indistinguishablility. The ciphertexts of honest parties are replaced by encryptions of 0s. By the IND-CPA property, it follows that this ciphertext is indistinguishable from the original message's ciphertext. Therefore the input to A^2 is indistinguishable to $\mathsf{view}^2(x)$ and consequently its output as well.

3. $\mathsf{view}_A^3(x) \sim \mathsf{view}_A^4(x)$

   In this step we remove all commitments except the commitment to x. The adversary $A^4(x)$ can compute the set $\{\mathsf{Com}_{x_j}\}$ for the corrupted parties P_j by recomputing them. For the other commitments, the adversary A^4 can interpolate the polynomial inside the commitments; since he has t values Com_{x_j} and the value Com_x this leads to a unique polynomial inside the commitments, i.e., $\{\mathsf{Com}_{x_j}\}_{j\in[1,n]}$. Then A^4 invokes A^3 on the computed input.

4. $\mathsf{view}_A^4(x) \sim \mathsf{view}_A^5(x)$

   Finally we want to remove the commitment Com_x. Since there are exactly t shares s_i in the adversaries' knowledge, any x can be used in order to determine a unique polynomial. By the computational hiding property, committing to any random value using uniform randomness cannot be distinguished from Com_x. Therefore $A^5(x)$ computes such a commitment and runs A^4 on this input.

We can conclude that $\mathsf{view}_A(s) \sim \mathsf{view}_A^5(s)$ and $\mathsf{view}_A(\overline{s}) \sim \mathsf{view}_A^5(\overline{s})$. Since we assume that $\mathsf{view}_A(s)$ is distinguishable from $\mathsf{view}_A(\overline{s})$, we can conclude that $\mathcal{K}_{corr}(s) = \mathsf{view}_A^4(s)$ is distinguishable from $\mathcal{K}_{corr}(\overline{t}) = \mathsf{view}_A^4(\overline{t})$. However, both $\mathcal{K}_{corr}(s)$ and $\mathcal{K}_{corr}(\overline{s})$ consists of the adversaries keys and — since D is honest — t values which are uniformly random. Therefore they cannot be distinguished (a contradiction). Hence the assumption has to be wrong and *privacy* follows by the contraposition.

COMMUNICATION COMPLEXITY: During the D-DEPENDENT PHASE, D has to non-equivocally distribute $\mathcal{O}(n)$ encryted shares and committed shares to every party, which costs $\mathcal{O}(n^2\kappa)$ bits. Construction of the certificate α^{D} requires $\mathcal{O}(n^2\kappa)$ bits of communication, as there are n ecnrypted and committed shares and so D needs to execute in total n^2 instances of PoE. Broadcasting α^{D} costs $\mathcal{O}(n^2\kappa)$ bits of communication, as the certificate is of size $\mathcal{O}(\kappa)$ bits. During the D-INDEPENDENT PHASE, each party just needs to send one encrypted share and one committed share to every other party, incurring a communication of $\mathcal{O}(n^2\kappa)$ bits. □

C.2 Properties of the Protocol Sup-PreMul-Sh

Lemma C.2. *Let v be a completely random and unknown value which is $[\cdot]$-shared among $\mathcal{P}$ and let u be a value selected by* D*. Then for every possible $\mathcal{A}$ and scheduler, protocol* Sup-PreMul-Sh *achieves the following (properties (1) and (2) up to a negligible error probability):* (1) TERMINATION*: if* D *and P_{king} are honest then all the honest parties eventually terminate the protocol. Moreover, if some honest party terminates the protocol, then every other honest party eventually does the same.* (2) CORRECTNESS*: if some honest party terminates the protocol, then there exists a value $\overline{u}$, such that $\overline{u}$ and $\overline{u} \cdot v$ will eventually be $[\cdot]$-shared among the parties. If* D *is honest then $\overline{u} = u$. Moreover if P_{king} is honest then P_{king} will be a privileged party with respect to $[\overline{u}]$ as well as $[\overline{u} \cdot v]$.* (3) PRIVACY*: v and $u \cdot v$ remains private at the end of the protocol. Additionally, if* D *is honest then u also remains private.* (4) COMMUNICATION COMPLEXITY*: the protocol has communication complexity $\mathcal{O}(n^2\kappa)$ bits.*

Proof. 1. COMMUNICATION COMPLEXITY: The *generating* phase of the protocol consists of one instantce of the Sup-Sh protocol which has communication complexity $\mathcal{O}(n^2\kappa)$. The D independent phase of the protocol is similar as in the Sup-Sh protocol and hence has communication complexity $\mathcal{O}(n^2\kappa)$. The same holds for the *share verification and certification* part of the protocol, a broadcast from the king with complexity $\mathcal{O}(n^2\kappa)$ complexity and n^2 instances of PoCM (n parties running n instances each). Hence this part has communication complexity $\mathcal{O}(n^2\kappa)$ as well. Finally, in the *share communication and certificate generation* part, the certificate β is broadcasted by the dealer and the dealer takes part in all executions of PoCM. In addition the dealer sends $3 + 2n$ commitments and n ciphertexts non-equivocally to every party. The broadcast and PoCM executions have communication complexity $\mathcal{O}(n^2\kappa)$ and the same holds for sending $\mathcal{O}(n)$ values of size $\mathcal{O}(\kappa)$ non-equivocally to every party. Consequently the overall protocol has complexity $\mathcal{O}(n^2\kappa)$.

2. TERMINATION: Termination of the *generating* $\langle u \rangle$ phase follows by the corresponding properties of the Sup-Sh protocol. For the remaining phases of the protocol the termination properties follow completely analogously to the corresponding properties of Sup-Sh since the protocol structure is essentially the same.

3. CORRECTNESS: For correctness we have to show that in the end there is an t-sharing $\bar{u}$ and $\bar{u}v$. We have to show that for an honest D it holds that $\bar{u} = u$ and that if the king is honest, he is priviledged with respect to $[u]$ and $[u \cdot v]$.

   The correctness of the protocol Sup-Sh already implies that there is a t-sharing of $\bar{u}$ and that an honest D will share $u = \bar{u}$. In addition, the correctness of Sup-Sh implies that P_{king} is privileged with respect to $[u]$.

   Therefore we only need to show that upon termination, there is a t-sharing of $\bar{u} \cdot v$ and that an honest P_{king} is privileged with respect to this sharing. Since an honest P_{king} only broadcasts $(\texttt{approve}, \mathsf{D})$ when he has received $\{\mathsf{Com}_u\}_{\sigma_D}, \{\{\mathbf{c}_{u \cdot v_j + m_j}\}_{\sigma_D}, \{\mathbf{c}_{u \cdot r_j + \hat{m}_j}\}_{\sigma_D}\}_{j \in [1,n]}, \{\{\mathsf{Com}_{u \cdot v_j + m_j}\}_{\sigma_D}\}_{j \in [1,n]}$ and $\{\mathsf{Com}_{u \cdot v}\}_{\sigma_D}$, as a subset of this message received all necessary information to be priviledged with respect to $\bar{u} \cdot v$. Finally, upon termination, every party P_i received $\{\mathbf{c}_{\bar{u} \cdot v_i + m_i}\}_{\sigma_D}$ and $\{\mathbf{c}_{\bar{u} \cdot r_i + \hat{m}_i}\}_{\sigma_D}$ as well as the corresponding commitment. For the same reason as in the proof of protocol Sup-Sh, these values correspond to the verified shares, i.e., belong to a degree t polynomial, and by the nonequivocation property, $\bar{u} = u$ is ensured.

4. PRIVACY: The privacy proof is threefold. First, we have to show that v remains private, i.e., communication during the protocol does not help in distinguishing the value of two different v. Second, we have to show that $u \cdot v$ remains private in the same sense and third, we have to show that u is private if the dealer D is honest. As in the proof for Sup-Sh we drop the σ_D annotation here, since there is no additional information in the tag than the tagged value already provides regarding the privacy.

(a) *v remains private:* Assume an adversary A can distinguish v from v' after seeing the additional information of the Sup-PreMul-Sh execution for some u. Since the execution requires that $[v]$ (or $[v']$) is already shared, it follows that an adversary $\mathcal{B}_A$ can internally simulate an execution of Sup-PreMul-Sh using u and then invoke A in order to distinguish v from v'. Hence v an adversary does not gain any additional information about v by the execution of Sup-PreMul-Sh.

(b) *$u \cdot v$ remains private:* Assume an adversary A can distinguish $w = u \cdot v$ from $w' = u' \cdot v'$. Since D may be corrupted, D can choose $u = u' = 1$. As a consequence A can now distinguish the cases $v = w$ from $v' = w'$ contradicting the privacy of v. Hence $u \cdot v$ remains private as well.

(c) *u remains private if D is honest:* This case is more difficult than the other two cases since the protocol leaks more information about u than the protocol Sup-Sh executed on u. However, we can follow the privacy proof of Sup-Sh.

Assume there is an adversary A that distinguishes the protocol execution for some u and u'. In addition to the execution of Sup-Sh on u, A gets $\{\mathbf{c}_{u \cdot v_j + m_j}, \mathbf{c}_{u \cdot r_j + \hat{m}_j}, \mathsf{Com}_{u \cdot v_j + m_j}\}_{j \in [1,n]}$, $\mathsf{Com}_{u \cdot v}$, the executions of PoCM and $(\texttt{approve}, \mathsf{D})$ as well as $\beta^{\mathsf{D},\tau}$. As in the proof of privacy with respect to the protocol Sup-Sh, the information $(\texttt{approve}, \mathsf{D})$ and $\beta^{\mathsf{D},\tau}$ does not help the adversary in distinguishing u from u'.

Since $\mathsf{Com}_{u \cdot v}$ can be computed from $\mathsf{Com}_{u \cdot v_j + m_j}$, we know there is an adversary that distinguishes u from u' without the input $\mathsf{Com}_{u \cdot v}$. Using the zero-knowledge property we can also simulate the proofs leading to an indistinguishable outcome of A (by definiton of zero-knowledge). As a consequence there is an adversary that distinguishes u from u' by seeing the output of Sup-Sh, $\mathbf{c}_{u \cdot v_j + m_j}, \mathbf{c}_{u \cdot r_j + \hat{m}_j}, \mathsf{Com}_{u \cdot v_j + m_j}$. Since the dealer D is honest, we can use the IND-CPA property to remove the ciphertexts $\mathbf{c}_{u \cdot v_j + m_j}, \mathbf{c}_{u \cdot r_j + \hat{m}_j}$ as we did in the privacy proof for Sup-Sh. Also following the privacy proof for Sup-Sh, we can reduce the input of the adversary to $\mathsf{Com}_{u \cdot v}$, which finally contradicts the computational hiding property of our commitment scheme. Consequently, there is no such adversary A.

□

D Protocol for Supervised Triple Generation and its Properties

Here we present our supervised triple-generation protocol; we first present the subprotocols used.

D.1 Protocol Sup-Second and Its Properties

Protocol Sup-Second (for generating the second component of the shared multiplication triple) is presented in Fig. 7.

The properties of the protocol Sup-Second are stated in Lemma D.1.

Lemma D.1. *For every possible $\mathcal{A}$ and every possible scheduler, the protocol* Sup-Second *achieves the following properties (properties (1) and (2) up to a negligible error probability):* (1) TERMINATION: *if P_{king} is honest, then all honest parties eventually terminate the protocol. Moreover, even if P_{king} is corrupted and some honest party terminates the protocol, then every other honest party eventually does the same.* (2) CORRECTNESS: *if the honest parties terminate the protocol, then the parties output $[\cdot]$-sharing $[v]$ of a value v. Moreover, each party will be a privileged party having all the encrypted share-pairs of v.* (3) PRIVACY: *the output shared value will be random from the viewpoint of $\mathcal{A}$.* (4) COMMUNICATION COMPLEXITY: *the protocol has communication complexity $\mathcal{O}(n^3\kappa)$ bits.*

Protocol $\mathsf{Sup\text{-}Second}(P_{\mathsf{king}}, \tau)$: τ is the session id

I. SHARING RANDOM VALUES—Every party $P_i \in \mathcal{P}$ including P_{king} executes the following code:

1. Select a random value $v^{(i)}$ and invoke an instance $\mathsf{Sup\text{-}Sh}(P_i, \tau, P_{\mathsf{king}}, v^{(i)})$ of $\mathsf{Sup\text{-}Sh}$ as a D to generate $[v^{(i)}]$ under the supervision of P_{king}; let this instance of $\mathsf{Sup\text{-}Sh}$ be denoted as $\mathsf{Sup\text{-}Sh}_i$. Moreover, let $\phi_i(\cdot), \psi_i(\cdot), \{\mathbf{c}_{v_{i,j}}, \mathbf{c}_{r_{i,j}}\}_{j\in[1,n]}, \{\mathsf{Com}_{v_{i,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{v^{(i)}}$ denote the sharing polynomial, randomness polynomial, encrypted share-pair, committed shares and commitment, generated during $\mathsf{Sup\text{-}Sh}_i$, where $v^{(i)} = \phi_i(0), v_{i,j} = \phi_i(j), r_{i,j} = \psi_i(j)$, $\mathbf{c}_{v_{i,j}} = \mathsf{Enc}_{\mathsf{pk}_j}(v_{i,j}, \cdot), \mathbf{c}_{r_{i,j}} = \mathsf{Enc}_{\mathsf{pk}_j}(r_{i,j}, \cdot), \mathsf{Com}_{v_{i,j}} = \mathsf{Commit}(v_{i,j}, r_{i,j})$ and $\mathsf{Com}_{v^{(i)}} = \mathsf{Commit}(\phi_i(0), \psi_i(0))$.
2. For $j \in [1, n]$, participate in the instance $\mathsf{Sup\text{-}Sh}_j$, invoked by P_j as a D.

II. COLLECTING AND DISTRIBUTING THE INFORMATION FOR THE FINAL OUTPUT—Only P_{king} executes the following code:

1. Include party P_k in an accumulative set $\mathcal{T}_{\mathsf{king}}$, which is initially $\emptyset$, if the instance $\mathsf{Sup\text{-}Sh}_k$ is (locally) terminated.
2. Wait till $|\mathcal{T}_{\mathsf{king}}| = t + 1$. Then using the linearity property of the encryption scheme, compute $\mathbf{c}_{v_j} = \boxplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathbf{c}_{v_{k,j}}$ and $\mathbf{c}_{r_j} = \boxplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathbf{c}_{r_{k,j}}$ for $j \in [1, n]$ and broadcast $\mathcal{T}_{\mathsf{king}}, \{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}$.
3. For every $P_k \in \mathcal{T}_{\mathsf{king}}$, forward the encrypted share-pairs $\{\mathbf{c}_{v_{k,j}}\}_{\sigma_k}, \{\mathbf{c}_{r_{k,j}}\}_{\sigma_k}$, for all $j \in [1, n]$, received as a privileged party from the dealer P_k during the instance $\mathsf{Sup\text{-}Sh}_k$, to every party in $\mathcal{P}$.

III. RESPONDING TO P_{king} AND TERMINATION—Every party $P_i \in \mathcal{P}$ including P_{king} executes the following code:

1. Include party P_k in an accumulative set $\mathcal{T}_i$, which is initially $\emptyset$, if the instance $\mathsf{Sup\text{-}Sh}_k$ is (locally) terminated.
2. Wait to receive $\mathcal{T}_{\mathsf{king}}$ and $\{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}$ from the broadcast of P_{king}.
3. Upon receiving $\{\mathbf{c}_{v_{k,j}}\}_{\sigma_k}, \{\mathbf{c}_{r_{k,j}}\}_{\sigma_k}$ for all $j \in [1, n]$ from P_{king} corresponding to each $P_k \in \mathcal{T}_{\mathsf{king}}$, wait till $\mathcal{T}_{\mathsf{king}} \subseteq \mathcal{T}_i$. Once $\mathcal{T}_{\mathsf{king}} \subseteq \mathcal{T}_i$, broadcast the message $(\mathtt{OK}, i)$ only if $\mathbf{c}_{v_j} = \boxplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathbf{c}_{v_{k,j}}$ and $\mathbf{c}_{r_j} = \boxplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathbf{c}_{r_{k,j}}$ holds for every $j \in [1, n]$.
4. Wait to receive the $(\mathtt{OK}, \star)$ message from the broadcast of at least $t + 1$ parties. Upon receiving, wait till $\mathcal{T}_{\mathsf{king}} \subseteq \mathcal{T}_i$ and then compute $v_i = \sum_{P_k\in\mathcal{T}_{\mathsf{king}}} v_{k,i}, r_i = \sum_{P_k\in\mathcal{T}_{\mathsf{king}}} v_{k,i}, \{\mathsf{Com}_{v_j} = \oplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathsf{Com}_{v_{k,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_v = \oplus_{P_k\in\mathcal{T}_{\mathsf{king}}}\mathsf{Com}_{v^{(k)}}$, where $v_{k,i}, r_{k,i}$ denotes the share-pair obtained at the end of the instance $\mathsf{Sup\text{-}Sh}_k$ and $\{\mathsf{Com}_{v_{k,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{v^{(k)}}$ denotes the vector of committed shares and the commitment obtained at the end of the instance $\mathsf{Sup\text{-}Sh}_k$. Finally, output $v_i, r_i, \{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}, \{\mathsf{Com}_{v_j}\}_{j\in[1,n]}$ and Com_v and terminate.

Figure 7: **Supervised generation of $[v]$ for a random v under the supervision of P_{king}; if the protocol terminates then each party will be a privileged party and will have all n encrypted shares of v.**

Proof. 1. TERMINATION: in order to show termination, we need to show two properties; first, if P_{king} is honest, then every honest party eventually terminates and second, if any honest party terminates, all other honest parties will do the same.

(a) *Honest king leads to termination:* if the king is honest, then the set $\mathcal{T}_{\mathsf{king}}$ will eventually reach the size $t + 1$, because of Theorem C.1 and since there are $t + 1$ honest dealers in the executions of $\mathsf{Sup\text{-}Sh}$. In particular, since the honest party P_{king} terminates for all executions corresponding to $\mathcal{T}_{\mathsf{king}}$, all honest P_i will eventually terminate for the same instances by the termination property of $\mathsf{Sup\text{-}Sh}$. Since P_{king} is honest, the checks done by the honest parties in the *response and termination* phase will succeed and they will broadcast $(\mathtt{OK}, \star)$. Thus, eventually the number of received $(\mathtt{OK}, \star)$ broadcasts will reach $t + 1$ and the honest parties terminate.

(b) *If an honest party terminates, then all honest parties do so:* let P_i be the honest party that terminates. We show that if a party P_j is honest, then P_j eventually terminates. The set $\mathcal{T}_i$ contains all parties for which P_i terminated the corresponding $\mathsf{Sup\text{-}Sh}$ when P_i terminated. By termination of the $\mathsf{Sup\text{-}Sh}$ protocol, it follows that eventually $\mathcal{T}_i \subseteq \mathcal{T}_j$. Since P_{king} broadcasts $\mathcal{T}_{\mathsf{king}}$ all parties will eventually receive the same $\mathcal{T}_{\mathsf{king}}$ and the condition $\mathcal{T}_{\mathsf{king}} \subseteq \mathcal{T}_j$ will eventually be satisfied. In addition, since P_i terminated, it received at least $t+1$ broadcasted messages $(\mathtt{OK}, \star)$

which will — since these messages were broadcast — eventually arrive at P_j. Consequently, this condition is satisfied as well. Thus P_j finally terminates.

2. CORRECTNESS: At the end of the protocol execution every party outputs $v_i, r_i, \{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}, \mathsf{Com}_{v_j}\}_{j\in[1,n]}$ and Com_v. There is an honest party that verifies that this message is a linear combination of the sharings run before. Since these precomputed sharings follow the correct protocol Sup-Sh and since sharings are linear, it follows that the parties hold a sharing of some value v when terminating.

3. PRIVACY: For all honest parties it holds that their sharing is indistinguishable for any value they shared, by the privacy of Sup-Sh. Since the overall output is a linear combination of $t+1$ sharings, at least one honest sharing is contained in this combination. Assuming the adversary A could distinguish this for any two different values v, v', then the adversary could use this to break the privacy of Sup-Sh, since he can control the t other parties from the output sharing. Thus, the privacy of Sup-Sh implies the privacy of Sup-Second.

   Moreover, the honest party of which a share is contained in the linear combination chooses this share uniformly at random. Consequently, the linear combination contains a value that is uniformly random as well.

4. COMMUNICATION COMPLEXITY: in the *Sharing random values* part of the protocol, there are n instances of the Sup-Sh protocol, i.e., a communication complexity of $\mathcal{O}(n^3\kappa)$.

   The *collection and distribution of the information* is done only by the party P_{king}. During the execution of this part of the protocol, the king broadcasts $\mathcal{T}_{\mathsf{king}}$ and $\{\mathbf{c}_{v_j}, \mathbf{c}_{r_j}\}_{j\in[1,n]}$ and forwards $\{\{\mathbf{c}_{v_{k,j}}\}_{\sigma_k}, \{\mathbf{c}_{r_{k,j}}\}_{\sigma_k}\}_{j\in[1,n]}$ for all $P_k \in \mathcal{T}_{\mathsf{king}}$. The broadcast message has size $\mathcal{O}(n\kappa)$ leading to a communication complexity of $\mathcal{O}(n^3\kappa)$ and the non-equivocally forwarded data has size $\mathcal{O}(n^2\kappa)$ leading to communication complexity $\mathcal{O}(n^3\kappa)$ as well.

   The final *response and termination* part which is executed by all parties consists only of broadcasting $(\mathtt{OK}, i)$. This message has size $\mathcal{O}(1)$ because the identity is encoded in the broadcast protocol. However, there are n broadcasts in the worst case, leading to a communication complexity of $\mathcal{O}(n^3\kappa)$.

□

D.2 Protocol Sup-FirAndThd and Its Properties

Protocol Sup-FirAndThd (for generating the first and third component of the shared multiplication triple) is presented in Figure 8.

The properties of the protocol Sup-FirAndThd are presented in Lemma D.2.

Lemma D.2. *For every possible A and every possible scheduler, the protocol* Sup-FirAndThd *achieves the following properties (properties (1) and (2) up to a negligible error probability):* (1) TERMINATION: *if P_{king} is honest, then all honest parties eventually terminate. Moreover, if one honest party terminates, then all honest parties eventually terminate.* (2) CORRECTNESS: *after termination, all parties hold a sharing $[u], [w]$ such that $[w] = [u \cdot v]$.* (3) PRIVACY: *the view of the adversary is indistinguishable for different values of u and v.* (4) COMMUNICATION COMPLEXITY: *the protocol has communication complexity $\mathcal{O}(n^3\kappa)$ bits.*

Proof. The proof completely follows the proof of Sup-Second, except that properties are now implied from Sup-Sh, instead of Sup-PreMul-Sh. □

Protocol $\mathsf{Sup\text{-}FirAndThd}(P_{\mathsf{king}}, [v], \tau)$: τ is the session id

i. SHARING RANDOM VALUES—Every party $P_i \in \mathcal{P}$ including P_{king} executes the following code:

1. Select a random value $u^{(i)}$ and invoke an instance $\mathsf{Sup\text{-}PreMul\text{-}Sh}(P_i, \tau, P_{\mathsf{king}}, \mathcal{P}, [v])$ of $\mathsf{Sup\text{-}PreMul\text{-}Sh}$ on $[v]$ as a D to generate $[u^{(i)}]$ and $[u^{(i)} \cdot v] = [w^{(i)}]$ under the supervision of P_{king}; let this instance of $\mathsf{Sup\text{-}PreMul\text{-}Sh}$ be denoted as $\mathsf{Sup\text{-}PreMul\text{-}Sh}_i$. Moreover, let $\{u_{i,j}, \overline{r}_{i,j}\}_{j\in[1,n]}, \{\mathbf{c}_{u_{i,j}}, \mathbf{c}_{\overline{r}_{i,j}}\}_{j\in[1,n]}, \{\mathsf{Com}_{u_{i,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{u^{(i)}}$ denote the vector of share-pairs, vector of encrypted share-pairs, vector of committed shares and the commitment corresponding to $[u^{(i)}]$ generated during $\mathsf{Sup\text{-}PreMul\text{-}Sh}_i$. Similarly, let $\{w_{i,j}, \hat{r}_{i,j}\}_{j\in[1,n]}, \{\mathbf{c}_{w_{i,j}}, \mathbf{c}_{\hat{r}_{i,j}}\}_{j\in[1,n]}, \{\mathsf{Com}_{w_{i,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{w^{(i)}}$ denote the vector of share-pairs, vector of encrypted share-pairs, vector of committed shares and the commitment corresponding to $[w^{(i)}]$ generated during $\mathsf{Sup\text{-}PreMul\text{-}Sh}_i$.
2. For $j \in [1, n]$, participate in the instance $\mathsf{Sup\text{-}PreMul\text{-}Sh}_j$, invoked by P_j as a D.

ii. COLLECTING AND DISTRIBUTING THE INFORMATION FOR THE FINAL OUTPUT—Only P_{king} executes the following code:

1. Include party P_k in an accumulative set $\mathcal{T}_{\mathsf{king}}$, which is initially $\emptyset$, if the instance $\mathsf{Sup\text{-}Sh}_k$ is (locally) terminated.
2. Wait till $|\mathcal{T}_{\mathsf{king}}| = t + 1$. Then broadcast $\mathcal{T}_{\mathsf{king}}$.

iii. RESPONDING TO P_{king} AND TERMINATION—Every party $P_i \in \mathcal{P}$ including P_{king} executes the following code:

1. Include party P_k in an accumulative set $\mathcal{T}_i$, which is initially $\emptyset$, if the instance $\mathsf{Sup\text{-}Sh}_k$ is (locally) terminated.
2. Wait to receive $\mathcal{T}_{\mathsf{king}}$ from the broadcast of P_{king}.
3. On receiving $\mathcal{T}_{\mathsf{king}}$, check if it is of size $t + 1$ and if so then wait till $\mathcal{T}_{\mathsf{king}} \subseteq \mathcal{T}_i$.
4. Compute $u_i = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} u_{k,i}, \overline{r}_i = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} \overline{r}_{k,i}, \{\mathsf{Com}_{u_j} = \oplus_{P_k \in \mathcal{T}_{\mathsf{king}}} \mathsf{Com}_{u_{k,j}}\}_{j\in[1,n]}, \mathsf{Com}_u = \oplus_{P_k \in \mathcal{T}_{\mathsf{king}}} \mathsf{Com}_{u^{(k)}}$, where $u_{k,i}, r_{k,i}, \{\mathsf{Com}_{u_{k,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{u^{(k)}}$ is obtained at the end of $\mathsf{Sup\text{-}PreMul\text{-}Sh}_k$, corresponding to $[u^{(k)}]$. Similarly compute $w_i = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} w_{k,i}, \hat{r}_i = \sum_{P_k \in \mathcal{T}_{\mathsf{king}}} \hat{r}_{k,i}, \{\mathsf{Com}_{w_j} = \oplus_{P_k \in \mathcal{T}_{\mathsf{king}}} \mathsf{Com}_{w_{k,j}}\}_{j\in[1,n]}, \mathsf{Com}_w = \oplus_{P_k \in \mathcal{T}_{\mathsf{king}}} \mathsf{Com}_{w^{(k)}}$, where $w_{k,i}, \hat{r}_{k,i}, \{\mathsf{Com}_{w_{k,j}}\}_{j\in[1,n]}$ and $\mathsf{Com}_{w^{(k)}}$ is obtained at the end of $\mathsf{Sup\text{-}PreMul\text{-}Sh}_k$, corresponding to $[w^{(k)}]$. Finally, output $u_i, \overline{r}_i, \{\mathsf{Com}_{u_j}\}_{j\in[1,n]}, \mathsf{Com}_u$ as well as $w_i, \hat{r}_i, \{\mathsf{Com}_{w_j}\}_{j\in[1,n]}, \mathsf{Com}_w$ and terminate.

Figure 8: **Supervised generation of $[u]$ and $[w = u \cdot v]$ for a random u under the supervision of P_{king}, where v is an existing $[\cdot]$-shared value, with every party being a privileged party with respect to $[v]$.**

D.3 The Supervised Tripled Generation Protocol SupTripGen

Protocol SupTripGen for the supervised triple generation, which is a combination of Sup-Second and Sup-FirAndThd, is presented in Fig. 9.

The following lemma follows easily from the properties of Sup-Second and Sup-FirAndThd and the protocol steps:

Lemma D.3. *For every possible $\mathcal{A}$ and every possible scheduler, the protocol* SupTripGen *achieves the following properties (properties (1) and (2) up to a negligible error probability):* (1) TERMINATION: *if P_{king} is honest then all honest parties eventually terminate the protocol. Moreover, even if P_{king} is corrupted*

Protocol $\mathsf{SupTripGen}(P_{\mathsf{king}}, \tau)$: τ is the session id

i. GENERATING THE SECOND COMPONENT OF THE TRIPLE—The parties in $\mathcal{P}$ execute an instance of $\mathsf{Sup\text{-}Second}(P_{\mathsf{king}}, \tau)$ to generate a uniformly random $[\cdot]$-shared value, say $[v]$.

ii. GENERATING THE FIRST AND THIRD COMPONENT OF THE TRIPLE—On terminating the instance of $\mathsf{Sup\text{-}Second}(P_{\mathsf{king}}, \tau)$, the parties execute $\mathsf{Sup\text{-}FirAndThd}(P_{\mathsf{king}}, [v], \tau)$ to obtain $[u]$ and $[w = u \cdot v]$, output $([u], [v], [w])$ and terminate.

Figure 9: **Supervised generation of a uniformaly random $[\cdot]$-shared multiplication triple, unknown to $\mathcal{A}$, under the supervision of P_{king}.**

and some honest party terminates the protocol, then every other honest party eventually does the same. (2) CORRECTNESS: *if the honest parties terminate the protocol, then the parties output $[\cdot]$-sharing $([u], [v], [w])$ of a multiplication triple (u, v, w).* (3) PRIVACY: *the shared multiplication triple (u, v, w) will be random from the viewpoint of $\mathcal{A}$.* (4) COMMUNICATION COMPLEXITY: *the protocol has communication complexity $\mathcal{O}(n^3\kappa)$ bits.*

E Analysis of the AMPC Protocol of [BTHN10]

The AMPC protocol of [BTHN10] operates over $\mathbb{Z}_N$. The input stage consists of a synchronous broadcast round, where every party encrypts its input and broadcasts it, along with a non-interactive zero-knowledge (NIZK) proof that it knows the underlying plaintext, corresponding to the ciphertext. Thus the input stage consists of a broadcast of $\mathcal{O}(n\kappa)$ bits. The secure evaluation of the circuit is then done using the *king-slave* paradigm, where every party in $\mathcal{P}$ acts as a king and all the n parties (including the king) act as slaves and perform the computation on behalf of the king, so as to enable the king to obtain the output of the function (to be computed). So in principle, the actual circuit is evaluated n times, once on behalf of each party. We focus on the actual communication done among the slaves to evaluate the circuit on the behalf of a single king.

Due to the homomorphic property of the encryption scheme, evaluating the addition gates required no interaction among the slaves. For a multiplication gate, a random encrypted multiplication triple unknown to $\mathcal{A}$ is generated for the slaves, under the supervision of the king. For this, the parties begin with a publicly known default encrypted multiplication triple, which is then randomized to new encrypted triples, for $t + 1$ iterations, by different slaves; the triple obtained after $t + 1$th iteration is taken as the final triple. In every iteration, to perform the randomization of an encrypted triple, the king sends a randomization request to all the n slaves. A slave, on receiving a randomization request, performs the randomization, and to prove to the king that he has the performed the randomization correctly, the slave provides a NIZK proof of $\mathcal{O}(\kappa)$ bits to every other slave, so as to obtain a threshold signature. In short, in every iteration, each slave performs a randomization and communicates $\mathcal{O}(n\kappa)$ bits to the other slaves to prove that he has the performed the randomization correctly.

Each iteration involves a communication of $\mathcal{O}(n^2\kappa)$ bits in total, and so $t + 1$ iterations require a total communication of $\mathcal{O}(n^3\kappa)$ bits. Thus evaluating a single multiplication gate under the king requires a communication of $\mathcal{O}(n^3\kappa)$ bits and so for c_M multiplication gates, it will incur a total communication of $\mathcal{O}(c_M n^3\kappa)$ bits for a single king. Therefore, for n kings, the protocol will require an overall communication of $\mathcal{O}(c_M n^4\kappa)$ bits.