Practical Asynchronous MPC from Lightweight Cryptography

Atsuki Momose
Quitee Research
atsuki.momose@gmail.com

ABSTRACT

We present an asynchronous secure multi-party computation (MPC) protocol that is practically efficient. Our protocol can evaluate any arithmetic circuit with linear communication in the number of parties per multiplication gate, while relying solely on computationally lightweight cryptography such as hash function and symmetric encryption. Our protocol is optimally resilient and tolerates t malicious parties among $n = 3t + 1$ parties.

At the technical level, we manage to apply the *player-elimination* paradigm to asynchronous MPC. This framework enables the detection and eviction of cheating parties by repeatedly attempting to generate Beaver triples. Once all malicious parties are eliminated, honest parties can proceed with efficient Beaver triple generation. While this approach is standard in synchronous MPC, it presents several technical challenges when adopted in an asynchronous network, which we address in this work.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**;

KEYWORDS

Multi-Party Computation (MPC); Asynchronous

1 INTRODUCTION

Secure Multi-Party Computation (MPC) [4, 16] enables a group of n parties to jointly evaluate a program over their private inputs in such a way that any group of t colluding parties learns no information about honest parties' inputs. MPC is a building block in cryptography with various applications such as threshold signing for secure key wallets [14], anonymous communication [18], private setup for cryptographic systems such as zero-knowledge proofs [6].

In the past four decades, MPC has been studied extensively from theoretical feasibility [4, 16, 23] to practical implemented protocols [1, 17], with some under active deployment. However, most of these protocols assume the underlying system is synchronous, i.e., network speed and computation time are predictable. Ten years ago, when distributed systems were deployed within a closed proprietary networks, these timing estimations were relatively safe. However, this is no longer the case in today's distributed computing systems, where computing parties are geographically distributed and communicate over an unreliable network under potential active attack. This demands research in asynchronous MPC.

While asynchronous MPC has also been studied since the early days [5, 9–11], most of these works focus on theoretical feasibility, and the current state-of-the-art lacks a practical solution. Existing asynchronous MPC protocols are either communication or computation inefficient. With computational assumptions, protocols with linear communication complexity (per multiplication gate) are known [11]. But they rely on computationally expensive public-key cryptography such as (threshold) homomorphic encryption. Eliminating expensive cryptography, information-theoretic (IT) secure protocols are also known [9, 10], but they incur prohibitively high communication cost.

Towards resolving this computation-communication trade-off, we take a middle-ground approach. We consider an asynchronous MPC that relies solely on *lightweight cryptography* such as hash functions and symmetric encryption. Namely, we still utilize cryptography for circumventing the technical challenge faced in the IT secure setting, but we minimize its use to retain computational efficiency. This paradigm has been explored recently in the contexts of asynchronous secret sharing [22] and consensus [13, 15], both of which are sub-problems of MPC. We further push forward these efforts towards building a complete solution to practical asynchronous MPC.

In this work, we design an asynchronous MPC protocol that is secure against $t < n/3$ malicious parties in the random oracle (RO) model. Our protocol can evaluate any arithmetic circuit with $O(C\kappa n + D\kappa n^2 + n^5)$ communication where C represents the number of multiplication gates, D denotes the multiplicative depth of the circuit, and κ is the computational security parameter. Our protocol relies solely on hash functions and symmetric encryption, both of which are available in the random oracle model. We do not use any expensive cryptography, either signature, public-key encryption, nor homomorphic encryption.

The rest of this section elaborates on the technical challenges and our key techniques.

1.1 Challenges in Asynchronous MPC

MPC protocols are commonly constructed with two components: an offline setup that generates Beaver's multiplication triples [2], and an online evaluation process that evaluates the given circuit using the prepared Beaver triples. It is well-known that the online evaluation can be done with linear communication and with even perfect security [12, 18]. Therefore, the challenge lies in the offline setup phase.

The classic approach to generating Beaver triples with linear communication involves the *player elimination* framework [12]. At a high level, the protocol repeatedly attempts to generate Beaver triples with linear communication, assuming all parties are honest. When each iteration fails, a forensic process follows to identify the cheating party for eviction. Once all corrupt parties have been evicted, all subsequent iterations will succeed. Therefore, while the eviction process can be potentially costly, these costs are amortized over iterations, making them negligible.

The player elimination paradigm is a well-established method in synchronous MPC [3, 12]. However, when attempting to apply it to asynchronous MPC, several challenges arise. To elaborate, a natural approach to implementing the player-elimination paradigm

is to utilize the *random double sharing* introduced by Damgard and Nielsen [12]. The random double sharing is a pair $[\![r]\!]^t, [\![r]\!]^{2t}$ of secret sharing of the same random value r with two different degrees, t and $2t$. When generating a Beaver triple $[\![a]\!]^t, [\![b]\!]^t, [\![ab]\!]^t$, the two random sharing $[\![a]\!]^t, [\![b]\!]^t$ are generated first, and then these two values are multiplied securely to derive $[\![ab]\!]^t$. To achieve the secure multiplication, parties first directly multiply the two shares $[\![a]\!]^t, [\![b]\!]^t$, which doubles the degree, i.e., $[\![ab]\!]^{2t}$ is derived. To reduce the degree to t, parties open

$$[\![\alpha]\!]^{2t} = [\![a]\!]^t [\![b]\!]^t + [\![r]\!]^{2t}$$

and reconstruct the masked value α. Parties then demask it with another share $[\![r]\!]^t$, which derives

$$[\![ab]\!]^t = \alpha - [\![r]\!]^t.$$

Unfortunately, this approach is not directly applicable in asynchronous MPC due to the two challenges below.

First, in an asynchronous network, honest parties are unable to robustly reconstruct α. To elaborate, in reconstructing a secret-shared value, honest parties must collect sufficient *correct* shares for error correction in case malicious parties send incorrect shares. Specifically, for the reconstruction of $[\![\alpha]\!]^{2t}$, honest parties need to gather at least $2t + 1$ correct shares to detect errors [20]. In a synchronous network, all honest parties respond timely. Thus, among $n = 3t + 1$ parties, at least $2t + 1$ shares from honest parties are guaranteed to be available for the reconstruction. However, in an asynchronous network with unbounded network delay, parties can only wait for responses from $2t + 1$ parties. Among these, up to t parties could be malicious. As a result, honest parties might end up with only $t + 1$ correct shares, which is insufficient for detecting error. Consequently, honest parties can undetectably reconstruct an incorrect value $\beta \neq \alpha$, leading to an incorrect triple.

Secondly, in an asynchronous network, generating random double sharing itself is challenging. The main difficulty lies in generating the degree-$2t$ sharing $[\![r]\!]^{2t}$. Typically, generating a secret-shared random value requires all parties to secret-share their own random values and then aggregate these shares. Thus, to generate $[\![r]\!]^{2t}$, a secret-sharing scheme that supports degree $2t$ is necessary. Unfortunately, there is currently no known secret-sharing scheme with linear communication that supports degree $2t$ [19, 24]. Moreover, even if generating $[\![r]\!]^{2t}$ were possible, it remains unclear how to generate another secret sharing $[\![r]\!]^t$ of the same value r.

The primary technical contribution of this work is to resolve these two problems, as elaborated below.

1.2 Weak Beaver Triple

To circumvent the first challenge, we introduce an *optimistic-but-agreed-upon* reconstruction. The core idea is to forgo the requirement of reconstructing the correct α and instead reach an agreement on a potentially incorrect triple using an agreed set of shares. Specifically, parties agree on $2t + 1$ shares of $[\![\alpha]\!]^{2t}$ through a consensus protocol, and use these shares to compute the unique reconstruction result β. As mentioned, given the nature of asynchronous networks, it is hard to gather sufficient correct shares, which means some of the agreed set of shares may come from corrupt parties, leading to an incorrect reconstruction result $\beta \neq \alpha$. However, a crucial observation is that all honest parties will arrive at the same reconstruction result β, which only affects the constant term c of the triple $[\![a]\!]^t, [\![b]\!]^t, [\![c]\!]^t$. In other words, while the generated triple may not be correct, it is still t-shared. More formally, these steps generate what we call the *weak Beaver triple*, which differs from the standard Beaver Triple in that it allows an adversary to introduce an additive error δ into the generated triple $[\![a]\!]^t, [\![b]\!]^t, [\![ab + \delta]\!]^t$.

Once many weak Beaver triples are generated, honest parties can batch-verify their correctness (i.e., that $\delta = 0$ for all triples) through standard polynomial identity checking over MPC [7]. Crucially, the unique t-shared triple allows all parties to agree on the verification result, enabling a unanimous abort to trigger the forensic process (to identify and eliminate cheating parties) in case of failure.

1.3 Weak Random Double Sharing

To address the second challenge of generating random double sharing, we relax the requirement and introduce *weak random double sharing*. The standard random double sharing turns out to be overkill once the Beaver triple has been weakened. Recall that in our new protocol, the reconstruction process does not guarantee the correctness of the result (i.e., it could be $\beta \neq \alpha$), and an adversary is allowed to arbitrarily alter the reconstruction outcome. This essentially implies we no longer rely on the correctness of the random double sharing. Instead, the only requirement for random double sharing in our protocol is its ability to provide sufficient masking power. Specifically, the goal is to ensure that the shares of $[\![\alpha]\!]^{2t}$ are effectively randomized, and when opened by honest parties, still conceal the two random values a and b in the triple from the adversary.

This relaxation allows us to weaken the requirements for random double sharing in two ways. First, the double sharing does not need to share the exact same random value r, as the demasking step might not be accurate anyway. Second, the degree of the second share can be greater than $2t$. We only require that the degree be at least $2t$ to ensure sufficient masking power, and there is no need for an upper bound. More concretely, the weak random double sharing is represented as

$$[\![u]\!]^t + [\![r]\!]^t, [\![v]\!]^d + [\![r]\!]^{2t}$$

where the former terms $[\![u]\!]^t, [\![v]\!]^d$ ($d \geq 2t$) are chosen arbitrarily by an adversary, and the latter $[\![r]\!]^t, [\![r]\!]^{2t}$ forms the correct random double sharing (chosen independently from the former). When used for generating a weak Beaver triple, parties open

$$[\![\alpha]\!]^d = [\![a]\!]^t [\![b]\!]^t + [\![r]\!]^{2t} + [\![v]\!]^d.$$

From the view of the adversary, this reveals no more information about a and b beyond what is learned when using the standard random double sharing, since the term $[\![r]\!]^{2t}$ effectively randomizes the shares. While the noisy terms $[\![v]\!]^d$ and $[\![u]\!]^t$ could lead to the generation of an incorrect triple, this is acceptable in our weakened Beaver triple.

The weak random double sharing is significantly easier to generate compared to the standard version. As mentioned earlier, the primary challenge in generating random double sharing was the lack of an efficient secret-sharing scheme that supports degree $2t$. The challenge in designing such a scheme lies in ensuring a degree upper bound against a malicious dealer. However, weak random

double sharing does not require the degree upper bound for the adversary generated term $[\![v]\!]^d$, which greatly simplifies the design of the secret sharing protocol. Furthermore, since the double sharing does not need to share the same random value, we no longer require a verification method for checking the equality of the shared values (i.e., checking $u = v$).

1.4 Efficient Message Transmission with Forensic Support

Finally, we note that using lightweight cryptography enables us to design a very simple and general solution for efficient message transmission with support for forensic processes. To elaborate, in the player-elimination framework, when the generation of Beaver triples fails, parties initiate a forensic process to identify the cheating party for eviction. The forensic process proceeds by tracing back the entire communication history to determine who lied. Therefore, we need a message transmission mechanism that allows other parties to later retrieve past messages when necessary. Furthermore, each message transmission must have asymptotically equivalent communication cost to pure message transmission (i.e., simply sending the message). We observe that such a scheme can be easily designed when symmetric encryption is available.

Suppose party s wants to send a message M to party r. First, the sender s secret-shares a key k with all parties. The receiver r then collects the shares $[\![k]\!]^t$ from the other parties to reconstruct the key. The sender r encrypts the message M using the secret key k. The ciphertext is then encoded into fragments using erasure coding [20], which are dispersed to all parties (a.k.a., Asynchronous Verifiable Information Dispersal or AVID [8]). The receiver r collects the coded fragments from the other parties to reconstruct the ciphertext, which is then decrypted to obtain the message M. Since each code symbol has a constant length, when the message size is sufficiently large (i.e., $\Omega(n)$), the communication cost is $O(M)$, the same as normal transmission. If another party later needs to retrieve the communication history, all parties open the shares $[\![k]\!]^t$ as well as the erasure-coded fragments of the ciphertext, allowing the message M to be publicly reconstructed.

Here, we remark that the above scheme has been used in previous works [22, 24], especially in the context of secret sharing. We review this scheme here to highlight its general applicability to cryptographic protocols with forensics, especially in the player-elimination paradigm.

1.5 Summary of Contribution

To summarize, we present an asynchronous MPC protocol with linear communication, utilizing only lightweight cryptography. The core component is a protocol for generating Beaver triples with linear communication. Our protocol is built around two key pillars that address the challenge of applying player-elimination paradigm to asynchronous MPC.

(1) **Weak Beaver Triple.** To bypass the challenge of reconstructing $2t$-shared values (which is inevitable in adopting random double sharing), we introduce an *optimistic-but-agreed-upon* reconstruction (Section 3). This approach allows the parties to generate a potentially incorrect but unique t-shared triple, which we refer to as *weak Beaver triple*, which can be effectively verified (Section 4), enabling unanimous abort to trigger forensics in case of error.

(2) **Weak Random Double Sharing.** The weakened Beaver triple allows us to relax the requirements for random double sharing, eliminating the need for the degree bound and the equality of the shared random values, which we refer to as *weak random double sharing* (Section 5), making it significantly easier to generate.

We then combine these two components along with the private message transmission scheme with forensic support (Section 6) to complete the player-elimination framework.

2 MODEL AND DEFINITIONS

We consider an asynchronous system consisting of $n = 3t + 1$ parties (numbered $1, \ldots, n$) where up to t parties are malicious. For simplicity, we assume parties $1, \ldots, t$ are malicious. The malicious parties are controlled by a probabilistic polynomial-time (PPT) adversary $\mathcal{A}$. While we assume a standard PPT adversary, our protocol utilizes only hash functions and symmetric encryption for cryptography. For simplicity, we assume the existence of an idealized random oracle (RO), denoted $\mathsf{H}(\cdot)$. Note that both hash functions and symmetric encryption are available in the random oracle model.

We assume a network with pairwise authenticated channels. Specifically, each party i has a dedicated channel to send/receive messages to/from every other party j. If both i and j are honest, an adversary is unable to see, alter, or fabricate any message in the channel. We assume the network is *asynchronous*. There is no bound on communication delay. An adversary can arbitrary delay messages sent by i; however, an adversary cannot delete any message sent by an honest party, and the message will eventually be delivered.

Notations. In this paper, unless explicitly stated otherwise, any *value* is an element of a field $\mathbb{Z}_q$ of a prime order $q = 2^\kappa$ where κ denotes a computational security parameter. $[a, b]$ refers to the set of ordered integers $\{a, a + 1, \ldots, b\}$. We use $[\![a]\!]^d$ to denote a Shamir shares [21] of a secret a with degree-d. Specifically, there is a degree-d polynomial $\phi(\cdot)$ with $\phi(0) = a$, and $[\![a]\!]^d_i$ denotes $\phi(i)$, i.e., party i's share. We use $\mathsf{poly}_d(S, j)$ to denote the point $p(j)$ on a degree-d polynomial $p(\cdot)$, defined by the set $S = \{(p(i), i)\}_{i \in X}$ of $|X| \geq d + 1$ points on the polynomial $p(\cdot)$. The evaluation of the polynomial can be done using standard linear interpolation.

2.1 Asynchronous MPC and Beaver Triple

Our goal is to design a secure multi-party computation (MPC) protocol for evaluating an arithmetic circuit. Concretely, we consider an MPC over secret-shared inputs. Given an arithmetic circuit C and secret-shared inputs $[\![x_1]\!]^t, \ldots, [\![x_m]\!]^t$, the protocol generates a secret-shared output $[\![C(x_1, \ldots, x_m)]\!]^t$. Note that the general definition of synchronous MPC assumes each party has its own input (i.e., party i inputs x_i). However, this is not well-suited for asynchronous MPC, as corrupt parties may withhold their inputs, which could prevent the protocol from terminating. Instead, we assume

Functionality: $\mathcal{F}_{\mathsf{BT}}$

The functionality generates $\ell + 1$ Beaver triples as follows.

- Receive from $\mathcal{A}$, $u_1^k, \ldots, u_t^k$ and $v_1^k, \ldots, v_t^k$ and $w_1^k, \ldots, w_t^k$, for all $k \in [0, \ell]$, which represent the malicious parties' shares of the triples.
- For all $k \in [0, \ell]$, sample a random a_k and compute shares $[\![a_k]\!]^t$ such that $[\![a_k]\!]_j^t = u_j^k$ for $j \in [1, t]$. Similarly, sample a random b_k and compute shares $[\![b_k]\!]^t$ such that $[\![b_k]\!]_j^t = v_j^k$ for $j \in [1, t]$. Then, compute shares $[\![a_k b_k]\!]^t$ such that $[\![a_k b_k]\!]_j^t = w_j^k$ for $j \in [1, t]$.
- Send to each party $i \in [1, n]$, the multiplication triple $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![a_k b_k]\!]_i^t$ for all $k \in [0, \ell]$.

Figure 1: Functionality for generating Beaver triples

the inputs are secret-shared before the protocol starts, as most applications use asynchronous MPC in this form [18].

Computing the output $[\![C(x_1, \ldots, x_m)]\!]^t$ involves evaluating each gate in the circuit while secret-sharing the inputs and outputs of the gate. Among these, linear operations are known to be free and can be done entirely locally (i.e., without interaction). Namely, given known constant u, v and the secret-shared inputs $[\![x]\!]^t, [\![y]\!]^t$, party i can evaluate the linear operation as follows.

$$[\![ux + vy]\!]_i^t = u[\![x]\!]_i^t + v[\![y]\!]_i^t.$$

On the other hand, multiplication can be done with (amortized) linear communication assuming that Beaver triples are available. Beaver triple is a random multiplication triple generated as described in Figure 1. Let us consider $\ell + 1$ multiplication gates where k-th gate takes $[\![x_k]\!]^t, [\![y_k]\!]^t$ as inputs. Using the k-th Beaver triple $[\![a_k]\!]^t, [\![b_k]\!]^t, [\![a_k b_k]\!]^t$, the gate is evaluated as

$$[\![x_k y_k]\!]_i^t = [\![a_k b_k]\!]_i^t + \beta_k [\![x]\!]_i^t + \alpha_k [\![y]\!]_i^t - \alpha_k \beta_k$$

where $\alpha_k = x_k + a_k$ and $\beta_k = y_k + b_k$. Reconstructing α_k and β_k can be done in batch with linear communication using the standard coding technique as follows [12].

Let us assume for simplicity that $\ell = t$. Let $\alpha(\cdot)$ be a degree-t polynomial such that $\alpha(k) = \alpha_k$; namely the polynomial define the Reed-Solomon (RS) code of the $t + 1$ values. Each party i computes $[\![\alpha(j)]\!]_i^t$ and sends it to party j. Party j then reconstructs $\alpha(j)$ through online error correction (OEC), retrieving the RS code symbol. Party j sends $\alpha(j)$ to all other parties. Finally, each party i uses OEC once again to reconstruct the entire polynomial $\alpha(\cdot)$, obtaining $\alpha_0, \ldots, \alpha_t$. This process reconstructs $t + 1$ values with $O(n^2)$ communication, thus amortized to $O(n)$ per value.

Thus, the remaining task is to design a protocol for generating Beaver triples (i.e., instantiating the functionality $\mathcal{F}_{\mathsf{BT}}$) with linear communication, which is the main focus of this paper.

2.2 Primitives

Our protocol is built on the following primitives. We will later specify the concrete protocols in Appendix B.

- ACS: Asynchronous Common Subset (ACS) protocol [15] that allows the parties to reach an agreement on a common subset of inputs. Specifically, the protocol takes as input a message m_i from each party $i \in [1, n]$, and outputs to all parties an agreed set $X = \{m_k, k\}_{k \in T}$ of inputs where $2t + 1 = |T|$ and $T \subseteq [1, n]$.
- $\mathcal{F}_{\mathsf{RandSh}}$: The functionality for generating a secret shared random value. Specifically, the functionality receives from the adversary, t shares $u_1, \ldots, u_t$ for malicious parties. Then, it samples a random value r and computes $[\![r]\!]^t$ such that $[\![r]\!]_j^t = u_j$ for $j \in [1, t]$. Finally, it delivers $[\![r]\!]_i^t$ to each party $i \in [1, n]$.
- $\mathcal{F}_{\mathsf{Coin}}$: The functionality for generating a common coin. Specifically, upon receiving a message *Flip* from $t + 1$ parties, it samples a random value τ and delivers it to all parties.
- $\mathcal{F}_{\mathsf{DoubleRand}}$: The functionality for generating a *random double sharing* [12]. Specifically, the functionality receives from the adversary, $2t$ shares $u_1, \ldots, u_t$ and $v_1, \ldots, v_t$. Then, it samples a random value r and computes the double sharing $[\![r]\!]^t, [\![r]\!]^{2t}$ such that $[\![r]\!]_j^t = u_j$ and $[\![r]\!]_j^{2t} = v_j$ for $j \in [1, t]$ (with t additional points on $[\![r]\!]^{2t}$ chosen at random). Note that we use this functionality in Section 3 for ease of exposition but will later replace it with weak random double sharing in Section 5.
- $\mathcal{F}_{\mathsf{VSS}}$: The functionality for Verifiable Secret Sharing (VSS) [22]. If a dealer is honest, the functionality samples a random sharing $[\![s]\!]^t$ and sends $[\![s]\!]_i^t$ to each party $i \in [1, n]$. If the dealer is malicious, upon receiving the entire n shares of an arbitrary sharing $[\![s]\!]^d$ from an adversary, and if $d = t$, the functionality sends $[\![s]\!]_i^t$ to each party $i \in [1, n]$.
- $\mathcal{F}_{\mathsf{AVID}}$: The functionality for Asynchronous Verifiable Information Dispersal (AVID) [8]. Upon receiving a message M from a dealer, the functionality sends a message *Dispersed* to all parties. Upon receiving a message *Retrieve* from all honest parties, the functionality sends to all parties the message M if it has sent *Dispersed* before.

3 GENERATING WEAK BEAVER TRIPLES

This section presents a protocol for generating weak Beaver triples. We describe the functionality for Weak Beaver triple in Figure 2. The key difference from the standard Beaver triple (Figure 1) is that the adversary is allowed to inject an additive error to the generated triple. Specifically, given two shared random values a_k and b_k, the functionality shares $c_k = a_k b_k + \delta_k$ where the error δ_k is chosen arbitrarily by an adversary. Therefore, if all parties behave honestly, the protocol generates correct Beaver triples. If some parties exhibit malicious behavior, the protocol may output incorrect triples (i.e., $\delta_k \neq 0$). In Section 4, we will also design a verification process for detecting incorrect triples so that parties can safely abort and retry the generation.

It is important to note that while the generated triple may be incorrect, it still constitutes a correct degree-t sharing. Namely, there exists a unique triple t-shared by honest parties, which can then be robustly reconstructed. Looking ahead, this enables unanimous abort among honest parties in case of failure (cf. Section 4).

Remark on extra shares. Note that the functionality $\mathcal{F}_{\mathsf{WeakBT}}$ outputs ℓ extra shares $[\![c_{\ell+1}]\!]_i^t, \ldots, [\![c_{2\ell}]\!]_i^t$. Each of these values c_k

Functionality: $\mathcal{F}_{\mathsf{WeakBT}}$

The functionality generates $\ell + 1$ weak Beaver triples as follows.

- Use $\mathcal{F}_{\mathsf{RandSh}}$ to generate $2(\ell + 1)$ shared random values, denoted $[\![a_0]\!]^t, \ldots, [\![a_\ell]\!]^t, [\![b_0]\!]^t, \ldots, [\![b_\ell]\!]^t$.
- Receive from $\mathcal{A}$ the malicious parties' shares $u_1^k, \ldots, u_t^k$ for $k \in [0, 2\ell]$, and 2ℓ additive errors $\delta_0, \ldots, \delta_{2\ell}$ for the generated triples.
- Compute $a_k = \mathsf{poly}_t(\{a_j, j\}_{j\in[0,\ell]}, k)$ for $k \in [\ell + 1, 2\ell]$, namely, the ℓ extra points on the degree-ℓ polynomial defined by $a_0, \ldots, a_\ell$. Compute b_k similarly.
- Let $c_k = a_k b_k + \delta_k$. For all $k \in [0, 2\ell]$, compute $[\![c_k]\!]^t$ such that $[\![c_k]\!]_j^t = u_k^j$ for all $j \in [1, t]$.
- Send to each party $i \in [1, n]$, the weak Beaver triple $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![c_k]\!]_i^t$ for all $k \in [0, \ell]$, along with the ℓ extra shares $[\![c_{\ell+1}]\!]_i^t, \ldots, [\![c_{2\ell}]\!]_i^t$.

Figure 2: Functionality for generating weak Beaver triples

is generated similarly from two values a_k, b_k and an error δ_k, while the values a_k, b_k are not uniformly random. More specifically, a_k is a point $a(k)$ on the degree-ℓ polynomial $a(\cdot)$ defined by the first $\ell + 1$ random values $a_0, \ldots, a_\ell = a(0), \ldots, a(\ell)$. These extra shares will be consumed by the verification process (Section 4) and will not be included in the generated Beaver triples.

3.1 Our Protocol

We describe our protocol in Figure 3. In this section, we assume the existence of $\mathcal{F}_{\mathsf{DoubleRand}}$, the functionality of generating random double sharing for simplicity. We will later remove this assumption when we introduce weak random double sharing (Section 5).

3.2 Security Proof

We show our protocol realizes the functionality $\mathcal{F}_{\mathsf{WeakBT}}$. The proof follows the standard UC-style, simulation-based argument. Specifically, we first define a simulator that simulates, in an ideal world, the view of the real-world adversary $\mathcal{A}$, while interacting with the ideal functionality $\mathcal{F}_{\mathsf{WeakBT}}$. Then, we show that any PPT machine, called the *environment* (denoted $\mathcal{Z}$), which observes the adversary's view and the parties' outputs cannot distinguish whether the world is ideal or real; namely, the joint distribution of the adversary's view and the parties' outputs is computationally indistinguishable in the two world.

Simulator. The simulator $\mathcal{S}$ first locally simulates an execution of the protocol while interacting with the adversary $\mathcal{A}$ until the point where $[\![c_0]\!]^t, \ldots, [\![c_{2\ell}]\!]^t$ are determined (i.e., when the ACS decides the output). Then, it sends to the functionality $\mathcal{F}_{\mathsf{WeakBT}}$ the following items for all $k \in [0, 2\ell]$:

- The malicious parties' shares $[\![c_k]\!]_1^t, \ldots, [\![c_k]\!]_t^t$
- The error $\delta_k = c_k - a_k b_k$. Here a_k, b_k is what is defined by the simulated execution, instead of what is chosen by $\mathcal{F}_{\mathsf{RandSh}}$ (which is not accessible to the simulator).

Protocol for $\mathcal{F}_{\mathsf{WeakBT}}$

The protocol generates $\ell + 1$ weak Beaver triples. Party i operates as follows:

- Call $\mathcal{F}_{\mathsf{RandSh}}$ to generate $2(\ell + 1)$ shared random values, denoted $[\![a_0]\!]^t, \ldots, [\![a_\ell]\!]^t, [\![b_0]\!]^t, \ldots, [\![b_\ell]\!]^t$.
- Call $\mathcal{F}_{\mathsf{DoubleRand}}$ to generate $2(\ell + 1)$ random double sharing, denoted $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ for $k \in [0, 2\ell]$.
- Compute $[\![a_j]\!]_i^t = \mathsf{poly}_\ell(\{[\![a_k]\!]_i^t, k\}_{k\in[0,\ell]}, j)$ for all $j \in [\ell+1, 2\ell]$, namely, shared points on the degree-ℓ polynomial defined by $a_0, \ldots, a_\ell$. Compute $[\![b_j]\!]_i^t$ similarly.
- Compute $[\![\alpha_k]\!]^{2t} = [\![a_k]\!]^t [\![b_k]\!]^t - [\![r_k]\!]^{2t}$ for $k \in [0, 2\ell]$.
- Batch-open and agree on $\alpha_0, \ldots, \alpha_t$ as follows:
  - Compute $s_{j,i} = \mathsf{poly}_t(\{[\![\alpha_k]\!]^{2t}, k\}_{k\in[0,t]}, j)$ for all $j \in [1, n]$, and send $(s_{j,i}, i)$ to party j.
  - Let S be the set $\{(s_{i,k}, k)\}_{k\in X}$ of tuples received from $|X| \geq 2t + 1$ parties. Input $\beta_i = \mathsf{poly}_{2t}(S, i)$ to ACS.
  - Let S' be the output from ACS. For all $j \in [0, t]$, compute $\beta_j = \mathsf{poly}_{2t}(S', j)$.

  Repeat these steps (in parallel) to also batch-open the remaining values $\alpha_{t+1}, \ldots, \alpha_{2\ell}$.
- Compute $[\![c_k]\!]^t = [\![r_k]\!]^t + \beta_k$ for all $k \in [0, 2\ell]$.

Figure 3: Protocol for generating weak Beaver triples

We now show that the environment $\mathcal{Z}$ cannot distinguish the ideal world and the real world.

Lemma 3.1. *The protocol in Figure 3 realizes the functionality $\mathcal{F}_{\mathsf{WeakBT}}$.*

Proof. The environment $\mathcal{Z}$ receives the following random variables in each world:

- The entire shares of $[\![a_k]\!]^t, [\![b_k]\!]^t$ from parties.
- The entire shares of $[\![\alpha_k]\!]^{2t}$ from the adversary.
- $\delta_k = c_k - a_k b_k$ received from the adversary.
- The malicious parties' shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ from the adversary.

Note that $[\![c_k]\!]^t$ is uniquely determined by $[\![a_k]\!]^t, [\![b_k]\!]^t, [\![r_k]\!]^t$ and δ_k. Also note that honest parties send linear combinations of the shares of $[\![\alpha_k]\!]^{2t}$ instead of the shares themselves for efficient batch opening, but the information received by $\mathcal{Z}$ is equivalent to the shares.

Obviously, what $\mathcal{Z}$ receives from $\mathcal{S}$ in the ideal world is identically distributed to what is received from $\mathcal{A}$ in the real world, because $\mathcal{S}$ locally runs the protocol without any deviation and forwards the view of $\mathcal{A}$ in the simulated execution. Since the latter three items are received from $\mathcal{A}$ (or $\mathcal{S}$), all we have to show is the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$ conditioned on the adversary's view is identical (in fact, we show it is uniformly distributed in the possible space) in both worlds. To this end, we consider the three points in the execution trace below.

Let us first consider the execution trace until $[\![a_k]\!]^t, [\![b_k]\!]^t$ and $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ are generated by $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{DoubleRand}}$. Let P_{a_k}

be the set of all possible $[\![a_k]\!]^t$ conditioned on the given malicious parties shares of $[\![a_k]\!]^t$. Similarly, let P_{b_k} be the set of all possible $[\![b_k]\!]^t$, and let P_{r_k} be the set of all possible $[\![r_k]\!]^{2t}$, conditioned on the malicious parties' shares. Obviously, $([\![a_k]\!]^t, [\![b_k]\!]^t)$ is uniformly distributed over $P_{a_k} \times P_{b_k}$ from the view of $\mathcal{A}$ by the definition of $\mathcal{F}_{\mathsf{RandSh}}$. Similarly, $[\![r_k]\!]^{2t}$ is uniformly distributed over P_{r_k} by the definition of $\mathcal{F}_{\mathsf{DoubleRand}}$.

Next, consider the execution trace up until $[\![\alpha_k]\!]^{2t}$ are revealed. Let P_{c_k} be the distribution of the product $[\![a_k]\!]^t[\![b_k]\!]^t$ without the knowledge of $[\![\alpha_k]\!]^{2t}$. Since $[\![r_k]\!]^{2t}$ is uniformly distributed over P_{r_k}, it follows that

$$[\![a_k]\!]^t[\![b_k]\!]^t = [\![\alpha_k]\!]^{2t} - [\![r_k]\!]^{2t}$$

is still distributed as P_{c_k}. Therefore, $[\![a_k]\!]^t, [\![b_k]\!]^t$ is still uniformly distributed over $P_{a_k} \times P_{b_k}$.

Finally, consider the rest, specifically, after β_k is decided by the ACS protocol. This is where the execution trace is fully determined. Recall that β_k is determined by the ACS protocol where honest parties' inputs only depend on $[\![\alpha_k]\!]^{2t}$. Thus, β_k is independent of anything beyond the view of $\mathcal{A}$ up to this point. As we have seen, the view of $\mathcal{A}$ is independent of the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$. Therefore, given δ_k, the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$ is still uniform over $P_{a_k} \times P_{b_k}$.

Now, in the ideal world, $\mathcal{F}_{\mathsf{WeakBT}}$ samples a_k and b_k uniformly randomly and use malicious parties' shares given by $\mathcal{S}$ to generate $[\![a_k]\!]^t, [\![b_k]\!]^t$. This is equivalent to sampling $[\![a_k]\!]^t, [\![b_k]\!]^t$ uniformly randomly from $P_{a_k} \times P_{b_k}$. Thus, the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$ conditioned on the view of $\mathcal{A}$ is identical in both worlds. □

4 BATCH-VERIFY WEAK BEAVER TRIPLES

In Section 3, we have introduced weak Beaver triple $\mathcal{F}_{\mathsf{WeakBT}}$ and presented a protocol for its generation. This section shows how to verify the weak Beaver triples in batch with linear communication. The weak Beaver triple $\mathcal{F}_{\mathsf{WeakBT}}$ together with the verification method provides the generation of the standard Beaver triples with ability to abort. Specifically, we present a protocol for realizing the functionality $\mathcal{F}_{\mathsf{BTAbort}}$ as described in Figure 4.

The functionality $\mathcal{F}_{\mathsf{BTAbort}}$ is equivalent to the generation of the standard Beaver triple $\mathcal{F}_{\mathsf{BT}}$ except that the adversary is allowed to send an *abort* instruction. In that case, the functionality sends an *Abort* message to all parties, resulting in a unanimous abort. If no abort instruction is sent, all parties receive correct Beaver triples. Intuitively, the abort instruction is sent (by the simulator) when the real-world adversary injects an error δ into the weak Beaver triples and the error detected by our verification process.

4.1 Our Protocol

We describe our protocol in Figure 5. We assume the existence of the functionality $\mathcal{F}_{\mathsf{BT}}$ for generating standard Beaver triples. Note that our goal is to realize this functionality *with linear communication*, and we can use any existing protocol with super-linear communication [10] for this purpose.

Functionality: $\mathcal{F}_{\mathsf{BTAbort}}$

The functionality generates $\ell + 1$ Beaver triples or aborts as follows.

- Use $\mathcal{F}_{\mathsf{RandSh}}$ to generate $2(\ell + 1)$ shared random values, denoted $[\![a_0]\!]^t, \ldots, [\![a_\ell]\!]^t, [\![b_0]\!]^t, \ldots, [\![b_\ell]\!]^t$.
- Receive from $\mathcal{A}$ the malicious parties' shares $u_1^k, \ldots, u_t^k$ for $k \in [0, 2\ell]$. Then, for all $k \in [0, \ell]$, compute $[\![a_k b_k]\!]^t$ such that $[\![a_k b_k]\!]_j^t = u_j^k$ for all $j \in [1, t]$.
- Receive from the adversary abort $\in \{1, 0\}$. If abort $= 1$, send to all parties a message *Abort*. Otherwise, send to each party $i \in [1, n]$, the Beaver triple $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![a_k b_k]\!]_i^t$ for all $k \in [0, \ell]$.

Figure 4: Functionality for generating Beaver triples with the ability to abort

Protocol for $\mathcal{F}_{\mathsf{BTAbort}}$

The protocol generates $\ell + 1$ Beaver triples or abort. Party i operates as follows:

- Call $\mathcal{F}_{\mathsf{WeakBT}}$ to generate $\ell+1$ weak Beaver triples, denoted $[\![a_k]\!]^t, [\![b_k]\!]^t, [\![c_k]\!]^t$ for all $k \in [0, \ell]$.
- Call $\mathcal{F}_{\mathsf{Coin}}$ to generate a random value τ. Then, compute $[\![a_\tau]\!]_i^t = \mathsf{poly}_\ell(\{[\![a_j]\!]_i^t, j\}_{j\in[0,\ell]}, \tau)$, and $[\![b_\tau]\!]_i^t$ similarly. Also, compute $[\![c_\tau]\!]_i^t = \mathsf{poly}_{2\ell}(\{[\![c_j]\!]_i^t, j\}_{j\in[0,2\ell]}, \tau)$.
- Securely compute $a_\tau b_\tau - c_\tau$ through MPC using a Beaver triple. Specifically,
  - Call $\mathcal{F}_{BT}$ to generate a Beaver triple $[\![r]\!]^t, [\![s]\!]^t, [\![rs]\!]^t$.
  - Compute $[\![\alpha]\!]_i^t = [\![a_\tau]\!]_i^t - [\![r]\!]_i^t$ and $[\![\beta]\!]_i^t = [\![b_\tau]\!]_i^t - [\![s]\!]_i^t$, and reconstruct α and β through OEC.
  - Compute $[\![\gamma]\!]_i^t = \alpha\beta - \alpha[\![s]\!]_i^t + \beta[\![r]\!]_i^t + [\![rs]\!]_i^t$ - $[\![c_\tau]\!]_i^t$, and reconstruct γ.
- If $\gamma = 0$, output $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![c_k]\!]_i^t$ for $k \in [0, \ell]$. Otherwise, output *Abort*.

Figure 5: Protocol for generating Beaver triples with the ability to abort

4.2 Security Proof

We first prove the following lemma that helps prove the simulatability.

Lemma 4.1. *If $\gamma = 0$, then the protocol outputs a correct Beaver triple $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![a_k b_k]\!]_i^t$ for all $k \in [0, \ell]$.*

Proof. Let $a(\cdot)$ be the degree-ℓ polynomial such that $a(k) = a_k$ for all $k \in [0, \ell]$. Similarly, let $b(\cdot)$ be the degree-ℓ polynomial such that $b(k) = b_k$ for all $k \in [0, \ell]$. Also, let $c(\cdot)$ be the degree-2ℓ polynomial such that $c(k) = c_k$ for all $k \in [0, 2\ell]$. It follows that $a_\tau = a(\tau)$, $b_\tau = b(\tau)$, and $c_\tau = c(\tau)$. By the correctness of the MPC multiplication using Beaver triples, we have $\gamma = a(\tau)b(\tau) - c(\tau)$.

Recall that $\mathcal{F}_{\mathsf{Coin}}$ samples the random coin τ upon receiving a message *Flip* from $t+1$ parties. At least one of these $t+1$ parties must be honest. The honest party invokes $\mathcal{F}_{\mathsf{Coin}}$ after receiving the share of the weak Beaver triple $[\![a_k]\!]^t, [\![b_k]\!]^t, [\![c_k]\!]^t$ for all $k \in [0,\ell]$ from $\mathcal{F}_{\mathsf{WeakBT}}$. This implies the polynomials $a(\cdot)$, $b(\cdot)$, and $c(\cdot)$ are determined before τ is chosen. Thus, $a(\tau)b(\tau) - c(\tau) = 0$ implies $a(\cdot)b(\cdot) = c(\cdot)$ w.h.p. Therefore, we have $c_k = a_k b_k$ for all $k \in [0,\ell]$. □

Now we describe a simulator and shows the view of $\mathcal{Z}$ is indistinguishable from that of the real world.

Simulator. The simulator $\mathcal{S}$ locally simulates an execution of the protocol with the adversary $\mathcal{A}$. If the protocol aborts in the local execution, the simulator sends abort = 1 to the functionality $\mathcal{F}_{\mathsf{BTAbort}}$. If the protocol outputs Beaver triples, the simulator sends abort = 0 and $u_i^k = [\![c_k]\!]_i^t$ for all $k \in [0,\ell]$ and $i \in [1,t]$.

Lemma 4.2. *The protocol in Figure 5 realizes the functionality* $\mathcal{F}_{\mathsf{BTAbort}}$.

Proof. In an aborted execution, parties simply output *Abort* in both worlds. Thus, it is obvious that the view of $\mathcal{Z}$ is identically distributed in both worlds. So, we only consider executions where parties do not abort. By Lemma 4.1, the triples generated by $\mathcal{F}_{\mathsf{WeakBT}}$ forms correct Beaver triples, i.e., $c_k = a_k b_k$ for all $k \in [0,\ell]$. The rest of the proof follows from the well-known security of MPC based on Beaver triples. □

5 WEAK RANDOM DOUBLE SHARING

The protocol in Section 3 assumes the existence of random double sharing $[\![r]\!]^t, [\![r]\!]^{2t}$. However, generating random double sharing in asynchronoy is difficult. To circumvent this challenge, this section introduces *weak random double sharing*. Weak random double sharing is a pair of random sharing $[\![w]\!]^t, [\![w']\!]^d$ with two relaxations: 1) the two random values w and w' can be different, and 2) the degree d of the second sharing can be greater than $2t$. More specifically, we describe the functionality for generating weak random double sharing in Figure 6.

5.1 Generating Weak Beaver Triple from Weak Random Double Sharing

We observe that weak random double sharing is sufficient for generating of weak Beaver triple. Specifically, the previous protocol (Figure 3) with the standard random double sharing replaced with weak random double sharing works. Our new protocol is described in Figure 7, with the differences from Figure 3 highlighted.

Security proof. We show our protocol in Figure 7 realizes the $\mathcal{F}_{\mathsf{WeakBT}}$. The proof goes with the same simulator $\mathcal{S}$ as in Section 3.2. Specifically, the simulator locally runs the protocol until the ACS stops, and then it sends to the functionality 1) the malicious parties' shares of $[\![c_k]\!]^t$, and 2) the error δ_k.

Proof Sketch. The proof goes similarly to Lemma 3.1. The main goal is to show that $[\![a_k]\!]^t, [\![b_k]\!]^t$ are uniformly distributed over the possible space (conditioned on the malicious parties' shares) from the adversary's view. The key difference in the adversary's view is what is received from the $[\![\alpha_k]\!]^{d_k}$, which is opened by the honest parties. However, as we have explained, it still hides the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$ due to the randomization by the correctly chosen term $[\![r_k]\!]^{2t}$ in the weak random double sharing. Therefore, the same argument applies as in Lemma 3.1.

Functionality: $\mathcal{F}_{\mathsf{WeakDR}}$

- Receive from $\mathcal{A}$ the malicious parties' shares $\rho_1^k, \ldots, \rho_t^k$ and $\mu_1^k, \ldots, \mu_t^k$ for all $k \in [0,\ell]$.
- For all $k \in [0,\ell]$, sample a random r_k and compute two random sharing $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ such that $[\![r_k]\!]_j^t = \rho_j^k$ and $[\![r_k]\!]_j^{2t} = \mu_j^k$ for all $j \in [1,t]$.
- Receive from $\mathcal{A}$ an arbitrary sharing $[\![u_k]\!]^t$ and $[\![v_k]\!]^{d_k}$ (the entire n shares) for all $k \in [k,\ell]$ where the degree d_k is also arbitrary.
- Send to each party $i \in [1,n]$, weak random double sharing $[\![w_k]\!]_i^t, [\![w'_k]\!]_i^{d_k}$ for all $k \in [0,\ell]$ defined as follows:

$$[\![w_k]\!]_i^t = [\![r_k]\!]_i^t + [\![u_k]\!]^t$$
$$[\![w'_k]\!]_i^{d_k} = [\![r_k]\!]_i^{2t} + [\![v_k]\!]_i^{d_k}$$

Figure 6: Functionality for generating weak random double sharing

Protocol for $\mathcal{F}_{\mathsf{WeakBT}}$

Let i be the party.

- Call $\mathcal{F}_{\mathsf{RandSh}}$ to generate $2(\ell+1)$ shared random values, denoted $[\![a_0]\!]^t, \ldots, [\![a_\ell]\!]^t, [\![b_0]\!]^t, \ldots, [\![b_\ell]\!]^t$.
- Call $\mathcal{F}_{\mathsf{WeakDR}}$ to generate $2(\ell+1)$ random double sharing, denoted $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ for $k \in [0,2\ell]$.
- Compute $[\![a_j]\!]_i^t = \mathsf{poly}_\ell(\{[\![a_k]\!]_i^t, k\}_{k\in[0,\ell]}, j)$ for all $j \in [\ell+1, 2\ell]$, namely, the share of points on the degree-ℓ polynomial defined by $a_0, \ldots, a_\ell$. Compute $[\![b_j]\!]_i^t$ similarly.
- Compute $[\![\alpha_k]\!]^{d_k} = [\![a_k]\!]^t [\![b_k]\!]^t - [\![w'_k]\!]^{d_k}$ for $k \in [0,2\ell]$.
- Batch-open and agree on $\alpha_0, \ldots, \alpha_{2\ell}$ as in Figure 3.
- Compute $[\![c_k]\!]^t = [\![w_k]\!]^t + \beta_k$ for all $k \in [0,2\ell]$.

Figure 7: Protocol for generating weak Beaver triples using weak random double sharing. The differences from the protocol in Figure 3 are highlighted in gray for clarify.

Lemma 5.1. *The protocol in Figure 7 realizes the functionality* $\mathcal{F}_{\mathsf{WeakBT}}$.

Proof. The environment $\mathcal{Z}$ receives the following random variables in each world:

- The entire shares of $[\![a_k]\!]^t, [\![b_k]\!]^t$ from parties.
- The entire shares of $[\![\alpha_k]\!]^d$ from the adversary.
- $\delta_k = c_k - a_k b_k$ received from the adversary.

Functionality: $\mathcal{F}_{\mathsf{WeakVSS}}$

Let d be the prescribed degree of the sharing.

- If the dealer is honest, the functionality samples a random sharing $[\![s]\!]^d$ and sends $[\![s]\!]^d_i$ to each party $i \in [1, n]$.
- If the dealer is malicious, upon receiving the whole n shares of an arbitrary sharing $[\![s]\!]^*$ from $\mathcal{A}$, the functionality sends $[\![s]\!]^*_i$ to each party $i \in [1, n]$.

Figure 8: Functionality for weak VSS. The differences from the standard VSS $\mathcal{F}_{\mathsf{VSS}}$ are highlighted in gray for clarity.

- The entire shares of $[\![u_k]\!]^t, [\![v_k]\!]^{d_k}$ from the adversary as part of the weak random double sharing $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$.
- The malicious parties' shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ from the adversary as part of the weak random double sharing.

As in Lemma 3.1, we show the distribution of $[\![a_k]\!]^t, [\![b_k]\!]^t$ conditioned on the adversary's view is identical in both worlds at any point in the execution trace.

Let P_{a_k} be the set of all possible $[\![a_k]\!]^t$ conditioned on the given malicious parties shares of $[\![a_k]\!]^t$. Similarly, let P_{b_k} be the set of all possible $[\![b_k]\!]^t$ conditioned on the malicious parties' shares. Also, let P_{r_k} be the set of all possible $[\![r_k]\!]^{2t}$, conditioned on the malicious parties' shares. Up until the point where $[\![a_k]\!]^t, [\![b_k]\!]^t$ and $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ are generated, $[\![a_k]\!]^t, [\![b_k]\!]^t$ is uniformly distributed over $P_{a_k} \times P_{b_k}$ from the view of $\mathcal{A}$ by the same argument as in Lemma 3.1.

Next, consider the execution trace up until $[\![\alpha_k]\!]^{d_k}$ are revealed. Let P_{c_k} be the distribution of the product $[\![a_k]\!]^t[\![b_k]\!]^t$ without the knowledge of $[\![\alpha_k]\!]^{2t}$. Recall that second sharing $[\![w'_k]\!]^{d_k}$ of weak random double sharing consists of two sharing $[\![r_k]\!]^{2t}$ and $[\![v_k]\!]^{d_k}$ chosen independently except for the malicious parties' shares. Since $[\![r_k]\!]^{2t}$ is uniformly distributed over P_{r_k}, it follows that

$$[\![a_k]\!]^t[\![b_k]\!]^t = ([\![\alpha_k]\!]^{d_k} - [\![v_k]\!]^{d_k}) - [\![r_k]\!]^{2t}$$

is still distributed as P_{c_k}. Therefore, $[\![a_k]\!]^t, [\![b_k]\!]^t$ is still uniformly distributed over $P_{a_k} \times P_{b_k}$. The rest of the proof goes as in Lemma 3.1. □

5.2 Generating Weak Random Double Sharing

The weak random double sharing is significantly easier to generate than the standard random double sharing. To elaborate, the common approach to generating a secret-shared random value involves all parties secret-sharing their own random values through VSS and then aggregating the shared secrets. Therefore, generating the random double sharing requires a VSS protocol with support for degree $2t$, which is currently unavailable. However, recall that the weak random double sharing does not impose the degree bound for the adversary-chosen part of the sharing, i.e., $[\![v_k]\!]$. This allows us to use a relaxed version we call *weak VSS* described in Figure 8. Namely, the relaxed functionality does not verify whether the degree d is as prescribed (e.g., it can be $d \neq 2t$), which makes it much easier to design compared to the standard VSS. We describe a protocol for $\mathcal{F}_{\mathsf{WeakVSS}}$ in Appendix A.

Protocol for $\mathcal{F}_{\mathsf{WeakDR}}$

The protocol generates $t + 1$ weak random double sharings. Party i operates as follows:

- Sample a random value s_i, and compute two random sharing $[\![s_i]\!]^t, [\![s_i]\!]^{2t}$ for each k.
- Distribute the shares $[\![s_i]\!]^t$ and $[\![s_i]\!]^{2t}$ by calling $\mathcal{F}_{\mathsf{VSS}}$ and $\mathcal{F}_{\mathsf{WeakVSS}}$, respectively.
- Run ACS to agree on a set of S of $2t + 1$ dealers, whose $\mathcal{F}_{\mathsf{VSS}}$ and $\mathcal{F}_{\mathsf{WeakVSS}}$ have both terminated.
- Let $[\![s_j]\!]^t_i$ and $[\![s'_j]\!]^*_i$ be the share received from $\mathcal{F}_{\mathsf{VSS}}$ and $\mathcal{F}_{\mathsf{WeakVSS}}$, respectively. For each $j \in [1, n]$, consider $[\![s_j]\!]^t_i = [\![s'_j]\!]^{2t}_i = 0$ if $j \notin S$. Then, compute $t + 1$ weak random double sharing as follows:

$$([\![w_1]\!]^t_i, \ldots, [\![w_{t+1}]\!]^t_i) = \boldsymbol{M}([\![s_1]\!]^t_i, \ldots, [\![s_n]\!]^t_i)$$
$$([\![w'_1]\!]^{d_1}_i, \ldots, [\![w'_{t+1}]\!]^{d_{t+1}}_i) = \boldsymbol{M}([\![s'_1]\!]^*_i, \ldots, [\![s'_n]\!]^*_i)$$

where $\boldsymbol{M}$ denotes a Vandermonde matrix.

Figure 9: Protocol for generating weak random double sharing

Furthermore, while the standard random double sharing requires verifying that each of the two VSS (i.e., used for $[\![r]\!]^t$ and $[\![r]\!]^{2t}$) shares the same values, this is no longer needed in the weak random double sharing. Thus, generating weak random double sharing involves simply running a standard VSS with degree-t and a weak VSS with degree-$2t$ independently. Specifically, our protocol for generating weak random double sharing is described in Figure 9.

5.3 Security proof.

We show that our protocol for generating weak random double sharing (Figure 9) realizes the functionality $\mathcal{F}_{\mathsf{WeakDR}}$. For simplicity we consider $\ell = t$.

Simulator. The simulator $\mathcal{S}$ locally simulates the execution of the protocol with the adversary $\mathcal{A}$ until ACS stops. Then, the simulator sends to the functionality the following items:

Let $\boldsymbol{m}_k = m_{k,1}, \ldots, m_{k,n}$ be the k^{th} row of the matrix $\boldsymbol{M}$, and $\boldsymbol{m}$

- The malicious parties' shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ for all $k \in [0, t]$ defined as follows:

$$[\![r_k]\!]^t = (m_{k,t+1}, \ldots, m_{k,n})([\![s_{t+1}]\!]^t, \ldots, [\![s_n]\!]^t)$$
$$[\![r_k]\!]^{2t} = (m_{k,t+1}, \ldots, m_{k,n})([\![s'_{t+1}]\!]^{2t}, \ldots, [\![s'_n]\!]^{2t}).$$

  Namely, these are the honest parties' generated part of the weak random double sharing.

- The entire shares of $[\![u_k]\!]^t, [\![v_k]\!]^{d_k}$ for all $k \in [0, t]$ defined as follows:

$$[\![u_k]\!]^t = (m_{k,1}, \ldots, m_{k,t})([\![s_1]\!]^t, \ldots, [\![s'_t]\!]^t)$$
$$[\![v_k]\!]^{d_k} = (m_{k,1}, \ldots, m_{k,t})([\![s'_1]\!]^*, \ldots, [\![s'_t]\!]^*)$$

  These are the malicious parties' generated part of the weak random double sharing.

We then show the simulated execution is indistinguishable from the real execution.

Lemma 5.2. *The protocol in Figure 9 realizes the functionality* $\mathcal{F}_{\mathsf{WeakDR}}$.

Proof. The environment $\mathcal{Z}$ receives the following random variables in each world:

- The entire parties' shares of $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ from parties.
- The entire shares of $[\![u_k]\!]^t, [\![v_k]\!]^{d_k}$ from the adversary.
- The malicious parties' shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ from the adversary.

Among these random variables, the latter two items must be distributed identically in both worlds, as the simulator locally runs the protocol without any deviation. So we show that the final output $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ received from the parties conditioned on the adversary's view is identical in both worlds. Obviously, the malicious parties' shares are generated in the same way in both worlds, and we focus on the honest parties' shares. The only difference in the generation of $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ between the two worlds is how the honest parties' shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ are chosen. In the ideal world, these are chosen uniformly randomly except for the malicious parties' shares, and its choice is independent from $[\![u_k]\!]^t, [\![v_k]\!]^{d_k}$. On the other hand, in the real world, these are defined by a linear combination of honest parties' secret sharing $[\![s_i]\!]^t, [\![s_i]\!]^{2t}$ (i.e., the Vandermonde matrix applied). From the view of the adversary, each of these secret sharing $[\![s_i]\!]^t, [\![s_i]\!]^{2t}$ is distributed uniformly randomly except for the malicious parties' shares. Furthermore, due to the hyperinvertible nature of the Vandermonde matrix, the shares of $[\![r_k]\!]^t, [\![r_k]\!]^{2t}$ are uniformly distributed except for the malicious parties' shares. Therefore, $[\![w_k]\!]^t, [\![w'_k]\!]^{d_k}$ are distributed identically in both worlds. □

6 DETECTING CHEATING PARTIES FOR ELIMINATION

In the previous sections, we have presented the *optimistic path* of the player-elimination framework, i.e., generating weak Beaver triples and verifying its correctness. To complete the protocol, this section presents the *forensic* process of the framework, i.e., identifying the cheating parties for eviction.

6.1 Private Message Transmission with Provable Revelation

To support forensics, we first extend the method for private message transmission with the provable revelation capability. Specifically, every private message transmission will be replaced with the functionality $\mathcal{F}_{\mathsf{PrivSend}}$ described in Figure 10.

Protocol. We describe a protocol for $\mathcal{F}_{\mathsf{PrivSend}}$ in Figure 11. It adopts the standard technique utilizing symmetric encryption and AVID.

Security proof. The proof is straightforward except for the simulation of encryption. Specifically, the simulator's local execution proceeds until the revelation starts. The environment $\mathcal{Z}$ who receives the message M from the receiver r cannot distinguish from the real execution until this point since the simulated ciphertext $\hat{C}$ cannot be distinguished from the real ciphertext C of the message M (due to the CPA security). When the revelation is invoked and the decryption key k is revealed, the simulator must show to $\mathcal{Z}$ that the encryption of the message M with the key k yield the simulated ciphertext $\hat{C}$. To this end, we use a programmable random oracle. Specifically, the ciphertext is generated as $C = \mathsf{H}^*(k) \oplus M$ where $\mathsf{H}^*(k)$ is a key extension for generating a random pad. Then, the simulator reprograms the random oracle as $\mathsf{H}(k)^* = \hat{C} \oplus M$ to convince the environment $\mathcal{Z}$ with the simulated ciphertext $\hat{C}$.

Functionality: $\mathcal{F}_{\mathsf{PrivSend}}$

Let s be the sender and r be the receiver.

- Receive from the sender s any message M. Then, send the message M to the receiver r ,and send a message *Delivered* to all parties.
- **Reveal.** If it receives a message *Reveal* from $t+1$ parties, send the message M (if it has sent *Delivered*) to all parties.

Figure 10: Functionality for private message transmission with support for provable revelation.

Protocol for $\mathcal{F}_{\mathsf{PrivSend}}$

Let s be the sender and r be the receiver.

- The sender s secret-shares a random key k through $\mathcal{F}_{\mathsf{VSS}}$. Each party i, upon receiving the share $[\![k]\!]^t_i$, forwards it to the receiver r. The receiver r reconstructs the key k through OEC.
- The sender s encrypts the message M with the key k and then disperse the ciphertext C through $\mathcal{F}_{\mathsf{AVID}}$. The receiver r sends a message *Retrieve* to $\mathcal{F}_{\mathsf{AVID}}$ to receive the ciphertext C. Then, the receiver decrypts the ciphertext with the key k and outputs the message M.
- Each party i outputs *Delivered* if it has received both the share from $\mathcal{F}_{\mathsf{VSS}}$ and a message *Dispersed* from $\mathcal{F}_{\mathsf{AVID}}$.
- **Reveal.** Upon receiving *Reveal* as input, each party i sends the share $[\![k]\!]^t_i$ to all parties and reconstruct the key k through OEC. Then, party i sends *Retrieve* to $\mathcal{F}_{\mathsf{AVID}}$ to receive the ciphertext C. Finally, party i decrypts the ciphertext C and outputs the message M.

Figure 11: Protocol for private message transmission with support for provable revelation.

6.2 Generating Beaver Triples with the Player-Elimination Framework

Now, putting everything together, we describe our protocol for generating Beaver triples in Figure 12.

The protocol consists of two components: optimistic generation of Beaver triples, and the forensic process to identify the cheating parties. In the optimistic path, the parties run our protocol for $\mathcal{F}_{\mathsf{BTAbort}}$. Here, all private message transmission in the protocol is

Protocol for $\mathcal{F}_{\mathsf{BT}}$

The protocol generates $\ell + 1$ Beaver triples. Party i operates as follows:

- **Optimistic generation.** Run the protocol for $\mathcal{F}_{\mathsf{BTAbort}}$ (in Figure 5) with all private message transmission conducted through $\mathcal{F}_{\mathsf{PrivSend}}$. If the party receives *Abort*, go to **Forensics**. Otherwise, output the Beaver triple $[\![a_k]\!]_i^t, [\![b_k]\!]_i^t, [\![c_k]\!]_i^t$ (for $k \in [0, \ell]$) received.
- **Forensics.** Send *Reveal* to every past $\mathcal{F}_{\mathsf{PrivSend}}$ instance to reveal all message transmissions. Then determine the malicious party j that has deviated from the protocol in either the following ways:
  - The party j has shared $[\![s_j]\!]^d$ with $d > 2t$ in $\mathcal{F}_{\mathsf{WeakVSS}}$ during the generation of random double sharing (i.e., the protocol in Figure 9).
  - The party j has sent to any party k an incorrect share $s_{j,k} \neq \mathsf{poly}_t(\{[\![\alpha_l]\!]_j^{2t}, l\}_{l\in[0,t]}, k)$ during the batch-opening step in the generation of weak Beaver triples (i.e., the protocol in Figure 3).
  - The party j has input an incorrect $\beta_j \neq \mathsf{poly}_{2t}(S, j)$ to ACS during the batch-opening step in the generation of weak Beaver triples.

  Then, go to **Optimistic generation** and reattempt to generate Beaver triples, while ignoring all later messages received from party j.

Figure 12: Protocol for generating Beaver triples

replaced with the functionality $\mathcal{F}_{\mathsf{PrivSend}}$ so any private communication can later be revealed during the forensic process. If the optimistic generation succeeds, parties output the correct Beaver triples received from $\mathcal{F}_{\mathsf{BTAbort}}$. In this case, no information is revealed beyond the generated triples, thus the protocol securely realizes the functionality $\mathcal{F}_{\mathsf{BT}}$.

When the optimistic generation fails (i.e., parties receive *Abort*), the generated weak Beaver triples are no longer used for output. Thus, parties invoke the revelation process of every past private communication to trace back the entire execution history. The failure in the optimistic generation occurs when at least one malicious party deviates from the protocol in either the following ways:

First, the party might have used a high degree $d > 2t$ for sharing a random value through $\mathcal{F}_{\mathsf{WeakVSS}}$ in the protocol in Figure 9. In this case, an incorrect random double sharing must be generated, resulting in an incorrect triple. This deviation is easily detected by retrieving the entire shares sent through $\mathcal{F}_{\mathsf{WeakVSS}}$.

If random double sharing was successfully generated, then the only malicious behavior that could affect the result is in the batch-opening step in the protocol Figure 3. Specifically, there are two possible deviations. First, the party j might have sent an incorrect share $s_{j,k}$ of $[\![\alpha_k]\!]_j^{2t}$ (i.e., the share of the k^{th} symbol of the RS code) to an honest party k. In this case, the party k must have failed in reconstructing its code word α_k. This deviation is also detected easily since correct $s_{j,k}$ can be computed from the shares $[\![a_*]\!]_j^t, [\![b_*]\!]_j^t$ and $[\![w_*]\!]_j^t, [\![w_*]\!]_j^{2t}$, which are revealed. Second the malicious party j might have input an incorrect symbol α_j to be included in the ACS output S, which must have resulted in a failure in reconstructing the correct $\alpha_0, \ldots, \alpha_\ell$. This is also easily detected by recomputing the correct α_j from the shares $[\![\alpha_j]\!]$.

The detected malicious party is then eliminated from all future attempts of the optimistic generation. Specifically, other parties simply delete any message from the party without inspecting the contents. After at most t repetitions, all malicious parties are eliminated from the protocol and all subsequent attempts will be successful.

7 CONCLUSION

In this paper, we have presented an asynchronous MPC protocol with linear communication complexity that relies solely on computationally lightweight cryptography, specifically hash functions and symmetric encryption. To achieve this, we have addressed several technical challenges in applying the player-elimination paradigm to asynchronous MPC. Specifically, we introduced optimistic-but-agreed-upon reconstruction to address challenges in the robust reconstruction of degree-$2t$ secret-shared values. We also presented weak random double sharing to address challenges in generating degree-$2t$ shared random values. We believe that our protocol is practically efficient and can contribute to the development of fully asynchronous distributed cryptographic systems, particularly in contexts such as anonymous communication networks and the secure generation of public parameters (e.g., powers-of-tau) for proof systems.

ACKNOWLEDGMENTS

This work is supported by a research grant from the Ethereum Foundation.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.

[2] Donald Beaver. 1992. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 420–432.

[3] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference (TCC)*. Springer, 213–230.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Annual ACM Symposium on Theory of Computing (STOC)*. 1–10.

[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. 183–192.

[6] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. 2015. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 287–304.

[7] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. 2012. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Annual Cryptology Conference*. Springer, 663–680.

[8] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 191–201.

[9] Ashish Choudhury and Arpita Patra. 2016. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory* 63, 1 (2016), 428–468.

[10] Ashish Choudhury and Arpita Patra. 2023. On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. *Journal of Cryptology* 36, 2 (2023), 13.

[11] Ran Cohen. 2016. Asynchronous secure multiparty computation in constant time. In *Public Key Cryptography (PKC)*. Springer, 183–207.

[12] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 572–590.
[13] Sourav Das, Sisi Duan, Shengqi Liu, Atsuki Momose, Ling Ren, and Victor Shoup. 2024. Asynchronous Consensus without Trusted Setup or Public-Key Cryptography. *Cryptology ePrint Archive* (2024).
[14] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. 2024. Threshold ECDSA in three rounds. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3053–3071.
[15] Sisi Duan, Xin Wang, and Haibin Zhang. 2023. FIN: practical signature-free asynchronous common subset in constant time. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 815–829.
[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 307–328.
[17] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2023. Efficient {3PC} for Binary Circuits with Application to {Maliciously-Secure}{DNN} Inference. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5377–5394.
[18] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 887–903.
[19] Atsuki Momose, Sourav Das, and Ling Ren. 2023. On the Security of KZG Commitment for VSS. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2561–2575.
[20] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
[21] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.
[22] Victor Shoup and Nigel P Smart. 2024. Lightweight asynchronous verifiable secret sharing with optimal resilience. *Journal of Cryptology* 37, 3 (2024), 27.
[23] Andrew C Yao. 1982. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 160–164.
[24] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. 2022. hbACSS: How to Robustly Share Many Secrets. In *Network and Distributed System Security Symposium (NDSS)*.

A WEAK VERIFIABLE SECRET SHARING

We describe a protocol for $\mathcal{F}_{\mathsf{WeakVSS}}$ in Figure 13.

Security proof. We first define the simulator $\mathcal{S}$ as follows:

- If the dealer is malicious, the simulator locally runs the protocol with the $\mathcal{A}$. When an honest party i outputs a share $[\![s]\!]_i^d$ in the simulated execution, the simulator sends the entire shares $[\![s]\!]^d$ to the functionality.
- If the dealer is honest, the simulator first receive the malicious parties' shares $[\![s]\!]_1^d, \ldots, [\![s]\!]_t^d$ from the functionality. Then, the simulator compute a random fake sharing $[\![r]\!]^d$ such that $[\![r]\!]_k^d = [\![s]\!]_k^d$ for all $k \in [1, t]$. Finally, the simulator locally runs the protocol where the simulated honest dealer shares $[\![r]\!]^d$.

We show the simulated execution is indistinguishable from the real execution.

Lemma A.1. *The protocol in Figure 13 realizes the functionality* $\mathcal{F}_{\mathsf{WeakVSS}}$.

Proof. Consider the malicious dealer case. If an honest party i outputs a share $[\![s]\!]_i^d$, it implies party i must have received a message *Delivered* from all n instances of $\mathcal{F}_{\mathsf{PrivSend}}$. Then, all parties will receive the shares of $[\![s]\!]^d$. Thus, the entire share $[\![s]\!]^d$ is defined and can be extracted from the simulated execution. Since the simulator simulates the execution exactly as in the real world, and the shares delivered by the functionality is provided by the simulator, the view of the environment $\mathcal{Z}$ is identically distributed in both worlds.

Protocol for $\mathcal{F}_{\mathsf{WeakVSS}}$

Let d be the degree of the sharing.

- The dealer samples a random secret s and compute the random shares $[\![s]\!]^d$. Then, the dealer sends $[\![s]\!]_i^d$ to each party $i \in [0, n]$ by calling $\mathcal{F}_{\mathsf{PrivSend}}$.
- Each party i outputs the share $[\![s]\!]_i^d$ if it has received a message *Delivered* from all n instances of $\mathcal{F}_{\mathsf{PrivSend}}$.

Figure 13: Protocol for weak VSS.

In the honest dealer case, the simulated dealer behaves as in the real execution except that the shares $[\![r]\!]^d$ is determined by the simulator. Here, the environment $\mathcal{Z}$ receives only the malicious parties' view of the simulated execution from the simulator $\mathcal{S}$. Since the malicious parties' shares of $[\![r]\!]^d$ are given by the functionality, the view of $\mathcal{Z}$ received from $\mathcal{S}$ must be identically distributed to the real world where the honest dealer shares $[\![s]\!]^d$. □

B PROTOCOLS FOR THE PRIMITIVES

We specify the protocols for the primitives listed in Section 2.2.

ACS. Asynchronous Common Subset (ACS) has been studied extensively in recent years and has several options. In our protocol, we use the state-of-the-art protocol by Duan et al. [15] that use only hash functions for cryptography, which fits in our model. The protocol also uses a common coin $\mathcal{F}_{\mathsf{Coin}}$, which is instantiated below.

VSS. We use the Verifiable Secret Sharing (VSS) protocol by Shoup et al. [22] that use only hash functions and symmetric encryption for cryptography. The protocol also uses a common coin $\mathcal{F}_{\mathsf{Coin}}$, which is instantiated below.

AVID. We use the Asynchronous Verifiable Information Dispersal (AVID) protocol by Cachin et al. [8].

Common coin. Common coin $\mathcal{F}_{\mathsf{Coin}}$ protocols are commonly constructed with VSS and ACS. Specifically, parties first secret shares a random value through VSS, and then agree on a set of VSS instances through ACS. The shares from the agreed set of VSS are linearly combined to derive a random sharing $[\![r]\!]^t$. Upon $t+1$ *Flip* calls, parties open the shares of $[\![r]\!]^t$ to compute the coin value r (or its hash). Unfortunately, we cannot directly use this approach since VSS and ACS themselves require a common coin protocol. To resolve this circular dependency, we use the Secret Key Sharing (SKS) protocol and the ACS protocol without common coin both introduced in [13]. Specifically, SKS is a weak version of VSS where only $t+1$ honest parties are guaranteed to receive the share. However, as observed in [13], this is sufficient for generating a common coin since the degree-t shared secrets are reconstructed from the $t+1$ honest parties shares. The ACS protocol is used to determine the set of $t+1$ completed SKS instances, which defines a common coin. The ACS protocol assumes only hash functions for cryptography.

Batched shared randomness. Once bootstrapping the common coin as above, the VSS protocol [22] can efficiently share many secrets in batch, and the ACS protocol of [15] works more efficiently

than [13]. Then, we can generate many shared random values in batch (i.e., instantiating $\mathcal{F}_{\mathsf{RandSh}}$) with linear communication by applying the Vandermonde matrix to the shared secrets as in Figure 9.