Practical Large-Scale Proof-Of-Stake Asynchronous Total-Order Broadcast

Orestis Alpos
University of Bern, Switzerland

Christian Cachin
University of Bern, Switzerland

Simon Holmgaard Kamp
Aarhus University, Denmark

Jesper Buus Nielsen
Aarhus University, Denmark

Abstract

We present simple and practical protocols for generating randomness as used by asynchronous total-order broadcast. The protocols are secure in a proof-of-stake setting with *dynamically changing* stake. They can be plugged into existing protocols for asynchronous total-order broadcast and will turn these into asynchronous total-order broadcast with dynamic stake. Our contribution relies on two important techniques. The paper "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography" [Cachin, Kursawe, and Shoup, PODC 2000] has influenced the design of practical total-order broadcast through its use of threshold cryptography. However, it needs a setup protocol to be efficient. In a proof-of-stake setting with dynamic stake this setup would have to be continually recomputed, making the protocol impractical. The work "Asynchronous Byzantine Agreement with Subquadratic Communication" [Blum, Katz, Liu-Zhang, and Loss, TCC 2020] showed how to use an initial setup for broadcast to asymptotically efficiently generate sub-sequent setups. The protocol, however, resorted to fully homomorphic encryption and was therefore not practically efficient. We adopt their approach to the proof-of-stake setting with dynamic stake, apply it to the Constantinople paper, and remove the need for fully homomorphic encryption. This results in simple and practical proof-of-stake protocols.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Total-Order Broadcast, Atomic Broadcast, Proof of Stake, Random Beacon

Digital Object Identifier 10.4230/LIPIcs.AFT.2023.31

Related Version *Full Version*: `https://eprint.iacr.org/2023/1103`

Funding This work has received funding from the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols). Simon Holmgaard Kamp and Jesper Buus Nielsen are partially funded by The Concordium Foundation.

1 Introduction

State of the art. It is well known that Asynchronous Total-Order Broadcast (ATOB) cannot be deterministic [25]. The necessary randomness is usually modelled as a *common coin* scheme [33], informally defined as a source random values observable by all participants but unpredictable for the adversary [10]. Common coins are most practically implemented using threshold cryptography [11, 23, 32, 10]. This approach has many benefits. It is conceptually simple and efficient, it achieves optimal resilience $t < n/3$, where n the number of parties running the protocol, and it results in a *perfect coin*, meaning that it is uniformly distributed and agreed-upon with probability 1. The drawback, however, is that it requires a trusted

© Orestis Alpos, Christian Cachin, Simon Holmgaard Kamp, and Jesper Buus Nielsen;
licensed under Creative Commons License CC-BY 4.0
5th Conference on Advances in Financial Technologies (AFT 2023).
Editors: Joseph Bonneau and S. Matthew Weinberg; Article No. 31; pp. 31:1–31:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

source: https://doi.org/10.48350/194692 | downloaded: 8.5.2024

setup or an Asynchronous Distributed Key Generation (ADKG) protocol. Current state of the art ADKG protocols [20, 1, 2] have communication cost of $O(\lambda n^3)$, where λ is the security parameter.

Given that state-of-the-art ATOB protocols have communication complexity $O(\lambda n^2)$, or even amortized $O(\lambda n)$, it is evident that the communication cost of ADKG becomes the bottleneck. In a permissioned setting with a static set of parties, it is common to proactively refresh the threshold setup [9]. In a Proof-of-Stake (PoS) setting, particularly, where the stake is constantly evolving and parties may dynamically join or leave the protocol, the ADKG protocol must be run periodically. Recent literature on asynchronous consensus uses committees, which contain only a subset of the parties, reducing the communication complexity of BA even further to $O(\lambda n \log n)$ at the cost of tolerating only $t < (1-\epsilon)n/3$ corruptions for any $\epsilon > 0$ [5, 18]. As the protocol run by the committee assumes an honest supermajority, this paradigm comes with one of two significant drawbacks. Either the sampled committee has to be very large, so that its maximal corruption remains below $n/3$ with overwhelming probability [22]. Otherwise, in order to keep the committee size small, the corruption level in the ground population must be assumed lower than $n/3$ by a considerable margin. Directly porting this idea to ADKG results in the same drawbacks. Finally, existing DKG protocols support only flat structures, where every party has the same weight and in total $t < n/3$ parties are corrupted. They do not readily work for a setting where every party holds a different share of the stake.

Seeds in PoS protocols. PoS-based ATOB protocols and blockchains require, apart from common coins, a second type of randomness, usually referred to as a *seed*. In PoS blockchains there is the notion of *accounts* with stake on them, of *roles*, such as "produce the 42-nd block", and of a *lottery*, through which accounts win the right to execute roles. This is typically [21, 26] implemented using a *Verifiable Pseudo-Random Function* (VRF) [30]: each account has a private key for a VRF and applies it to the role, producing a pseudorandom value. If this value is above a threshold then the account wins the right to execute the role. However, for this approach to work the lottery needs as input not only a role but also a seed. Without the seed, a party can operate with several accounts and move all its stake to the luckiest account. By including a seed in the lottery and using the stake distribution from a point in time when the seed was unpredictable one can mitigate this attack [21].

In practice one can use a common-coin protocol to produce the seeds. We remark, however, that the two randomness-generation protocols have different requirements. A common-coin scheme does not have to be always unpredictable and agreed-upon, but only with some constant probability [31, 12]. It should, however, be efficient, as it is used in every agreement instance within the broadcast protocol. On the other hand, the seed-generation protocol must always be unpredictable and agreed upon, but it can be slow, as it is only run periodically (e.g., once per epoch).

Related work. Multiple common-coin constructions without a trusted dealer have been proposed in the literature. Ben-Or [4] presents a simple protocol, where every party flips a *local coin*. As a result, parties agree on the value of the coin only with probability $\Theta(2^{-n})$, A common-coin scheme from *verifiable secret sharing* has been shown by Canetti and Rabin [12], but their resulting Byzantine agreement protocol has communication complexity $\mathcal{O}(n^{11})$. Patra, Choudhury, and Rangan [31] bring this down to $\mathcal{O}(n^3)$.

A different approach constructs common coins from *publicly verifiable secret sharing*. The resulting protocols [13, 14, 19, 36, 38] are efficient, yet they all make synchrony assumptions. RandShare [38] has been formalized in the asynchronous communication setting but it is

not scalable, as it requires $\mathcal{O}(n^3)$ communication per party. Another line of work is based on *time-based cryptography*. Protocols in this category [29, 16] employ *verifiable delay functions* [7] and rely on the assumption that certain functions (such as exponentiation in groups of unknown order [35]) can only be computed serially. None of the aforementioned works explicitly mentions the network assumptions. Overviews of random beacon protocols are given by Raikwar and Gligoroski [34], and by Choi, Manoj, and Bonneau [17].

Multiple works that circumvent ADKG exist in the literature, but they either make more assumptions, have non-optimal resilience, or result in inefficient protocols. Existing PoS blockchains rely on the timely delivery of honestly generated blocks, hence make timing assumptions. Ouroboros Praos [21] implements a randomness beacon protocol, used as seed in their leader-election algorithm, by hashing a large number of VRF outputs. Partial-synchrony assumptions assure that the honestly generated VRF outputs cannot be delayed arbitrarily by the adversary. King and Saia [27, 28] propose a synchronous common-coin protocol that makes uses of pseudorandomly selected committees, but achieves non-optimal resilience. This is improved in the protocol of Algorand [26, 15], where each committee member applies a VRF on the seed of previous block, and then the smallest valid VRF value sent by some committee member is kept. The protocol is first described in the synchronous model [26] and later extended to the partially synchronous [15]. Cohen, Keidar, and Spiegelman [18] extend this idea to the asynchronous model, but their protocol achieves an $n = 4.5t$ resilience. In all these protocols the coins are not reusable and the whole coin-generation algorithm has to be run repeatedly.

Blum *et al.* [5] also generate randomness without ADKG. Their ATOB protocol works in the following way. Assume first that a trusted dealer publishes on a ledger all the setup material required for one instance of Byzantine agreement and one instance of a Multiparty Computation (MPC) protocol. Then, on every invocation of the agreement protocol, parties use the Byzantine-agreement setup in the agreement protocol and the MPC setup in a tailor-made MPC protocol that refreshes the whole setup. Finally, they replace the trusted dealer with a standard MPC protocol, executed once in a distributed setup phase. This blueprint solves the problem of dynamic stake elegantly, but, the proposed MPC protocol for refreshing the setup, which has to be executed for *every* Byzantine agreement instance, is not efficient: it employs Threshold Fully Homomorphic Encryption (TFHE), digital signatures, and zero-knowledge proofs.

Contributions. In this paper we address all the aforementioned limitations of randomness generation for the first time. We present asynchronous seed-generation and common-coin protocols that

- require no trusted setup,
- support optimal resilience $t < n/3$,
- employ small committees and are concretely efficient,
- directly support the PoS setting and dynamic participation,
- are modular and can be generically used in any ATOB broadcast.

Our methods. We are motivated by the question whether one can use the simple, practical, and efficient approach of getting common coins from threshold setup without running inefficient and complicated protocols whenever the stake has shifted. Building on the idea of Blum *et al.* [5], we rely on the fact that there already exists a functional ATOB: we generate the setup assuming that we already have the ATOB, and then use the generated setup to

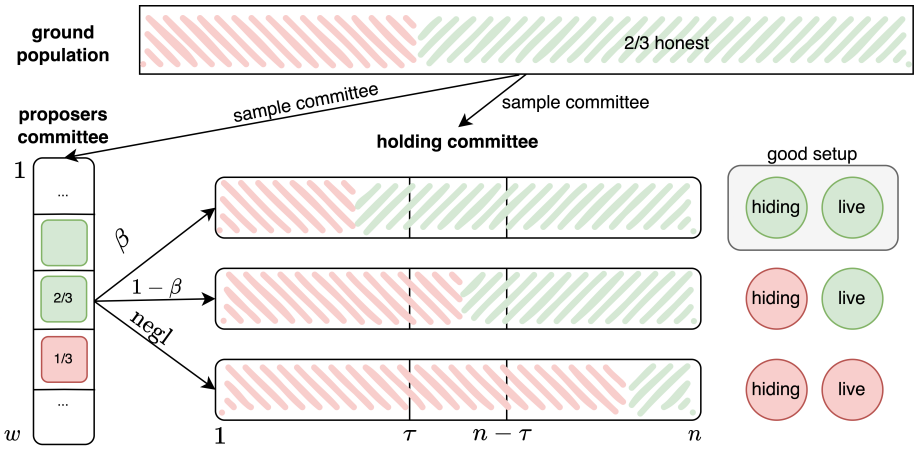

Figure 1 The high level idea of our protocols. A *proposers committee* is elected, and we wait until w proposers broadcast a setup. Assuming $2/3$ honesty in the ground population, a proposer is honest with probability $2/3$. Each proposer is assigned a *holding committee* of size n and creates an (n, τ) threshold setup for it. A committee is *hiding* if it contains at most τ corrupted parties, and *live* if it contains at most $n - \tau - 1$ corrupted parties. A setup is *good* if its proposer in honest and its holding committee is hiding and live. We set w so as to have enough honest proposers, and n and τ so that each holding committee is hiding with constant probability β and live with all but negligible probability. As a result, we get good setups with a constant probability γ.

keep the ATOB running. To maintain practical efficiency the crucial step is to avoid FHE. We achieve this by generating weaker setups than Blum *et al.* [5], nonetheless still strong enough for the continued execution of the ATOB.

A crucial observation is that coins consumed by Byzantine agreement do not need to be perfect, i.e., always unpredictable and agreed upon [12, 31]. Hence, instead of generating a single, perfect threshold setup, we generate several candidate setups, such that some *constant* fraction of them are good. Many DKG protocols can be seen as doing this as their first step, but their next step is to combine them into a single perfect setup. In order to be combinable, the setups must be of a particular form, and the committee that holds the setup must be *good* (that is, contain less than a threshold corruptions) except with negligible probability. As our setups are not combined and our committees only need to be good with a constant probability, our protocols are simpler and more efficient, and use smaller committees.

Both seed, our seed-generation protocol, and wMDCF, our common-coin protocol, follow the approach depicted in Figure 1. They elect a *proposers committee*, each member of which is expected to create a *setup* (a *VSS setup* or a *coin setup*, for seed and wMDCF, respectively). Each elected proposer is assigned a *holding committee*, for which it creates the threshold setup. For this, the proposer acts as a dealer, encrypts the private setup material under the keys of the holding committee, and broadcasts these encryptions and the required verifications keys with a single message on the ATOB. We use a VRF-based lottery to determine both the proposers and the holding committees, where each party is elected with probability proportional to its stake. To open a seed value in the seed protocol, each of the holding committees reveals its shares and these are all added together. To flip a coin cid in the wMDCF protocol we first open a new seed value and then hash cid with the seed to obtain a pointer to one of the published setups, which is used to obtain the value of cid.

As we show in Section 7, we can have a proposers committee of size 653 and holding committees of size 359, resulting in approx. $85K$ encrypted values posted on Ledger. For 60 bits of security and assuming optimal corruption $1/3$ in the ground population, our protocols

are live with all but negligible probability, and our common-coin protocol is unpredictable and agreed-upon with probability approx. 31.8%. It is instructive to compare these results against previous literature, particularly against the approach that runs the randomness-generation protocols in committees with honest supermajority. Algorand [26, Figure 3] requires a committee of size approx. 2000, assuming corruption 0.2 in the ground population, and larger than 4000, assuming corruption 0.24, to get good committees with probability $5 \cdot 10^{-9}$, or approx. 28 bits of security. Extending this approach to a ground population with corruption 0.3, which is still sub-optimal, and 60 bits of security, the authors of GearBox [22, Table 1] show that committees of size 16037 are needed. We remark that asynchronous distributed key generation protocols, the state-of-the-art approach for threshold-setup generation, require honest supermajority, hence one would require a committee of similar sizes and sub-optimal resilience in the ground population.

Organization. The rest of this paper is organized as follows. Section 2.1 presents the formal model used in the schemes and Section 2.2 presents the primitives used in our schemes. Then, each of the seed-generation and common-coin protocols are presented in modular way, in two steps. Section 3 presents *wVSS*, a weak verifiable secret sharing scheme, which is then used in Section 4 to build the *seed-generation* protocol. Section 5 presents wHDCF, a weak honest-dealer coin-flip protocol, which is then used in Section 6 to build the wMDCF common-coin protocol. All of these schemes are parameterized over committee sizes and thresholds, and secure bounds for these are computed in Section 7.

2 Preliminaries

2.1 Model

We assume a model with asynchronous authenticated point-to-point channels. In addition we assume an asynchronous persistent total-order broadcast channel. We denote by Ledger the totally-ordered sequence of messages that have been delivered on the channel. We point out that if a blockchain has a distinction between final and non-final messages, then Ledger denotes the final messages. We assume that when a protocol is started all the parties taking part in the protocol agree on a session identifier sid and an existing point on the ledger, $p \leq |\mathsf{Ledger}|$. We think of p as the *starting point of the protocol*, which gives consensus on the context of the protocol like stake distribution and lottery as discussed below. Protocols can have *public output* which might not be explicitly posted on the ledger, but will have a well-defined value and virtual point p at which they happened.

▶ **Definition 1** (Public output). *We say that* PubOutF *is a public output function if it computes a public output from a ledger* Ledger *and a session identifier* sid*, where either* $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) = y \in \{0,1\}^*$ *or* $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) = \perp$*. We require that if* $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) \neq \perp$ *then* $\mathsf{PubOutF}(\mathsf{Ledger}\|m, \mathsf{sid}) = \mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid})$ *for all* m*. We say that* sid *gave public output* y *at position* p *if* $|\mathsf{Ledger}| \geq p$ *and* $\mathsf{PubOutF}(\mathsf{Ledger}[1, p-1], \mathsf{sid}) = \perp$ *and* $\mathsf{PubOutF}(\mathsf{Ledger}[1, p], \mathsf{sid}) = y$*. Unless multiple* sid*'s are in scope, we omit the* sid *parameter. Finally, we informally say that some protocol gives public output* PubOutF *when additionally the ledger is implicit or when it is an eventual property of the ledger.*

Dynamic Stake. We consider proof-of-stake defined via the ledger. For each Ledger there is a stake distribution $\Sigma(\mathsf{Ledger}) : \mathbb{P} \to \mathbb{R}_0$ which may change as the ledger grows, can be computed in poly-time, and which gives for each party P its stake $\Sigma(\mathsf{Ledger})(\mathsf{P})$. For each point p there is also a stake distribution Σ_p, which is the stake distribution used by protocols with p as starting point. It may be different from $\Sigma(\mathsf{Ledger}[1, p])$, as discussed below.

Lotteries. In PoS based protocol it is common that parties are selected at random for carrying out a role in the protocol, like serving on a committee or producing the next block in a blockchain. To keep the model simple we assume that this is done via a random oracle. To keep the model simple we assume that for each point p on the ledger there is a random oracle $\Gamma_p : \{0,1\}^* \to \{0,1\}^\lambda$. We assume that Γ_p is sampled and made available to the parties at some point *after* Σ_p can be computed from Ledger. This ensures that Γ_p is independent of Σ_p. If Γ_p was made available before Σ_p was fixed, then corrupted parties would be able to update Σ_p based on Γ_p (e.g., by moving their stake to parties "lucky" in Γ_p). We implement this by iteratively generating random and unpredictable seeds, appearing as public outputs, with our seed protocol. Then, for a given point p, let seed_p be the latest seed on $\mathsf{Ledger}[1,p]$, let $\Gamma_p(x) = R(\mathsf{seed}_p, x)$, for a random oracle R, let $p' < p$ be the latest point where seed_p was unpredictable, and let $\Sigma_p = \Sigma(\mathsf{Ledger}[1,p'])$.

Our protocols include steps where a party samples a committee cid of size n. We model this as a function $\mathsf{SampleCommittee}_p(\mathsf{cid}, n) \to \left(\mathsf{H}_i\right)_{i\in[n]}$ that uses Γ_p to sample (with replacement) n parties from $\mathbb{P}$ with probability proportional to the stake Σ_p. As the input is public, the output can be verified by a function $\mathsf{VerifyCommittee}_p(\mathsf{cid}, \left(\mathsf{H}_i\right)_{i\in[n]})$ that reruns $\mathsf{SampleCommittee}_p(\mathsf{cid}, n)$ and verifies that it matches $\left(\mathsf{H}_i\right)_{i\in[n]}$. We assume $\mathsf{SampleCommittee}_p(\mathsf{cid}, n)$ is locally computable by every party. Using our lottery abstraction this could be implemented by calling $\Gamma_p(\mathsf{cid}, i)$, for some committee cid and for $i \in [n]$, to obtain a number $r_i \in \{0, 2^\lambda - 1\}$, and then deterministically mapping r_i to a party $P_i \in \mathbb{P}$ based on Σ_p. Observe that a party with relatively large stake can appear multiple times in the committee.

2.2 Primitives

Our schemes make use of the following primitives.

2.2.1 Public-Key Encryption with Full Decryption

There are keys $(\mathsf{dk}_i, \mathsf{ek}_i)$, for all $\mathsf{P}_i \in \mathbb{P}$, for an IND-CPA encryption scheme with full decryption, PKE. Encrypting a message $m \in \mathsf{PKE}.\mathcal{M}$ using randomness $r \in \mathsf{PKE}.\mathcal{R}$ results in a ciphertext $c = \mathsf{Enc}_{\mathsf{ek}_i}(m; r) \in \mathsf{PKE}.\mathcal{C}$. Given a ciphertext $c \in \mathsf{PKE}.\mathcal{C}$ the decryption algorithm $\mathsf{Dec}_{\mathsf{dk}_i}(c)$ returns both $m \in \mathsf{PKE}.\mathcal{M}$ and $r \in \mathsf{PKE}.\mathcal{R}$. The triple (m, r, c) can then be verified by anyone holding ek_i by checking if $\mathsf{Enc}_{\mathsf{ek}_i}(m; r) = c$. Given an invalid ciphertext a zero knowledge proof that the ciphertext is invalid can be obtained using the secret key.

Construction using El Gamal. We first show that we can obtain the properties above in the random oracle model, as long as only encryptions of random messages are needed. This can then be lifted to a complete encryption scheme by symmetrically encrypting the message under a freshly sampled random key.

To encrypt a random value r, use El Gamal with $H(r)$ as randomness. I.e. if $\mathsf{dk} = x$ and $\mathsf{ek} = h = g^x$, then you encrypt r as $c = (A, B) = (g^{H(r)}, r \cdot h^{H(r)})$. To decrypt you first compute $r = B/(A^x)$, then check if re-encrypting using $H(r)$ as randomness gives back c. If verification checks out you can simply send r as proof. If the re-encryption does not match, you provide a proof that r was obtained by decrypting c. Note that (A, B) decrypts to r (under (g, h)) iff $\mathrm{DL}_g(h) = \mathrm{DL}_A(B/r)$, so this proof can be constructed using the Fiat-Shamir transform of the Σ-protocol for equality of discrete logarithms.

In the full scheme, in order to encrypt m using randomness r, you encrypt r as above and additionally include a symmetric encryption of m using r as key. To decrypt you first use regular El Gamal decryption to obtain r and verify it by re-encrypting. If it was encrypted correctly you use it to decrypt m and return (m, r), otherwise return $(\perp, r)$.

2.2.2 Threshold Coin Flip

We use a (n, t)-threshold coin-flip (CF) scheme, where n is the total number of parties, t is the corruption threshold, and the reconstruction threshold is $t + 1$. The scheme has the following interface.

- $\mathsf{Setup}(n, t) \to (\mathsf{vk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$: The dealer generates a verification key vk and secret key shares sk_i of P_i. The secret keys can be used to create coin shares of multiple coins.
- $\mathsf{VerifyKeyShare}(\mathsf{vk}, i, \mathsf{sk}_i) \to b \in \{0, 1\}$: Given the verification keys vk, it verifies sk_i.
- $\mathsf{Flip}(\mathsf{sk}_i, \mathsf{coin}) \to (\mathsf{s}_i, \mathsf{w}_i)$: Given a coin identifier coin and secret key sk_i, it returns a coin share s_i for coin and potentially a correctness proof w_i, i.e., a proof that the coin share has been computed correctly using sk_i.
- $\mathsf{VerifyCoinShare}(\mathsf{vk}, \mathsf{coin}, \mathsf{s}_i, \mathsf{w}_i) \to b \in \{0, 1\}$: It verifies coin share s_i for coin identifier coin using the correctness proof w_i and verification key vk.
- $\mathsf{Combine}(\mathsf{coin}, \{\mathsf{s}_{i_j}\}_{j \in [t+1]}) \to \mathsf{s} \in \{0, 1\}^\lambda$: Given $t + 1$ valid coin shares s_{i_j}, for $j \in [t + 1]$, it returns the value s of the coin identifier coin.
- $\mathsf{VerifyCoin}(\mathsf{vk}, \mathsf{coin}, \mathsf{s}) \to b \in \{0, 1\}$: It verifies s as the value of coin identifier coin using the verification key vk.

Security properties. Assuming an honest dealer, i.e., that $\mathsf{Setup}()$ is correctly executed, and that there are no more than t corrupted parties, the scheme satisfies the following.

Completeness If the dealer is honest then all key shares generated with $\mathsf{Flip}(\mathsf{sk}_i, \mathsf{coin})$ will verify with $\mathsf{VerifyCoinShare}$.

Agreement For any $t + 1$ valid key shares the value $\mathsf{Combine}(\mathsf{coin}, \{\mathsf{s}_{i_j}\}_{j \in [t]})$ is the same, which define *the* value $\mathsf{s}_{\mathsf{coin}}$.

Unpredictability The value $\mathsf{s}_{\mathsf{coin}}$ is unpredictable without honest shares, i.e., for a set $C = \{\mathsf{P}_{i_j}\}_{j \in [t+1]}$ of corrupted parties, if a poly-time adversary has been given vk and sk_i for $\mathsf{P}_i \in C$ for a random setup and has not been given $\mathsf{Flip}(\mathsf{sk}_i, \mathsf{coin})$ for $\mathsf{P}_i \notin C$, then it cannot guess $\mathsf{s}_{\mathsf{coin}}$ better than at random. This holds even if it has access to an oracle giving $\mathsf{Flip}(\mathsf{sk}_i, \mathsf{coin}')$ for all honest P_i for all $\mathsf{coin}' \neq \mathsf{coin}$.

Instantiation. Scheme CF can be instantiated with any non-interactive unique threshold signature scheme, such as BLS threshold signatures [8, 6]. The dealer picks a random secret key sk and shares it among all n parties using a polynomial $\phi(X) = \sum_{k=0}^{t} \phi_k X^k$, such that $\phi_0 = \mathsf{sk}$. The only difference from threshold BLS is in $\mathsf{Setup}()$: it runs the key generation algorithm of the threshold signature scheme, but it does not return the verification keys in the form $g_2^{\mathsf{sk}_i} \in G_2$, where $i \in [n]$ and g_2 is the generator of G_2, as in the original scheme. Instead, it returns a vector $(V_0, \ldots, V_t)$, where $V_k = g^{\phi_k} \in G_2$, for $k \in \{0, \ldots, t\}$, i.e., it returns Feldman commitments [24] to the coefficients of ϕ. This allows us to implement $\mathsf{VerifyKeyShare}()$, so P_i can verify that its key share sk_i is indeed a point on polynomial ϕ by checking whether

$$g_2^{\mathsf{sk}_i} \stackrel{?}{=} \prod_{k=0}^{t} (V_k)^{i^k}. \tag{1}$$

Observe that the original verification keys can still be obtained using (1) with input vk and i, hence $\mathsf{VerifyCoinShare}()$ and $\mathsf{VerifyCoin}()$ need no modification. Algorithm $\mathsf{Flip}()$ returns a signature share s_i on message coin using the key share sk_i of party P_i. Algorithm $\mathsf{Combine}()$ creates the threshold signature s from $t + 1$ valid signature shares, which can then be hashed

to get a value in $\{0, 1\}$. Algorithms $\mathsf{VerifyCoinShare}()$ and $\mathsf{VerifyCoin}()$ invoke the signature verification algorithm, which, in the case of BLS, only takes as input the message coin and a signatures share s_i or signature s, i.e., $\mathsf{w}_i = \perp$, and uses a pairing function. Alternatively, one can use the common-coin scheme of Cachin, Kursawe, and Shoup [10], but $\mathsf{VerifyCoin}()$ would additionally need as input the $t + 1$ valid coin shares and proofs $\{\mathsf{s}_{i_j}, \mathsf{w}_{i_j}\}_{j \in [t+1]}$.

2.2.3 Secret sharing

We require a secret-sharing scheme TSS with threshold t with the following interface.

1. $\mathsf{Share}(s; r) \to (s_1, \ldots, s_n)$: It shares a secret s using randomness r to n secret shares $(s_1, \ldots, s_n)$.
2. $\mathsf{Reconstruct}(\{s_{i_j}\}_{j=1}^{t}) \to s'$: Given t shares it reconstructs some secret s'.

The hiding property says that the joint distribution of t shares s_i is independent of s. We can instantiate TSS with Shamir's secret sharing scheme [37].

2.2.4 Digital Signature

Finally, there are keys $(\mathsf{sk_P}, \mathsf{vk_P})$, for all $\mathsf{P} \in \mathbb{P}$, for a digital signature scheme DS with unique signatures.

3 Weak Verifiable Secret Sharing

In this section we define a weak VSS protocol. It is weak in the sense that it is sometimes not hiding. But it is always binding and live (allows reconstruction). There is a designated dealer D, which is one of the participating parties. We assume D is given as part of session identifier, $\mathsf{sid} = (\mathsf{D}, \mathsf{sid}')$, and hence is known by all parties when the instance is created.

Syntax. The syntax of wVSS is as follows.

Commit On input $(\textsc{commit}, \mathsf{sid}, m)$ to D it starts running the commitment protocol and may as a result help produce a public output (see Definition 1) $\mathsf{PubOutVSSCommit}$. On input $(\textsc{commit}, \mathsf{sid})$ to a participating party $\mathsf{P} \neq \mathsf{D}$ it starts running the commitment protocol and may as a result help produce a public output $\mathsf{PubOutVSSCommit}$.

Open On input $(\textsc{open}, \mathsf{sid})$ after $\mathsf{PubOutVSSCommit}(\mathsf{sid}, \mathsf{Ledger_P}) \neq \perp$, a party P starts running the open protocol and may as a result output $(\textsc{done-open}, \mathsf{sid}, m_\mathsf{P}, \pi)$, where π is a proof that m_P is the output. The proof can be checked by any party P' for which $\mathsf{PubOutVSSCommit}(\mathsf{sid}, \mathsf{Ledger_{P'}}) \neq \perp$ using $\mathsf{wVSSVerify}(\pi, m)$.

Security. The security properties of wVSS are as follows.

Termination

(1) If D is honest and gets input $(\textsc{commit}, \mathsf{sid}, m)$, and all other honest parties get input $(\textsc{commit}, \mathsf{sid})$ then eventually there is a public output $\mathsf{PubOutVSSCommit}$.

(2) If $\mathsf{PubOutVSSCommit}$ occurred and all honest parties get input $(\textsc{open}, \mathsf{sid})$ then eventually all honest parties give an output $(\textsc{done-open}, \mathsf{sid}, \cdot)$.

(3) If any honest party gives an output $(\textsc{done-open}, \mathsf{sid}, \cdot)$ then eventually all honest parties give output $(\textsc{done-open}, \mathsf{sid}, \cdot)$.

Validity If D is honest and had input $(\textsc{commit}, \mathsf{sid}, m)$ and some honest P gave output $(\textsc{done-open}, \mathsf{sid}, m_\mathsf{P}, \pi)$, then $m_\mathsf{P} = m$ and $\forall m' : \mathsf{wVSSVerify}(\pi, m') = \top \Leftrightarrow m' = m$.

Binding If D is corrupted, then the following holds. When the first honest party observes public output PubOutVSSCommit then one can in poly-time compute from the view of the adversary up to this point, a message m such that if later an honest party P gives output $(\text{DONE-OPEN}, \mathsf{sid}, m_{\mathsf{P}}, \pi)$ then $m_{\mathsf{P}} = m$ and $\forall m' : \mathsf{wVSSVerify}(\pi, m') = \top \Leftrightarrow m' = m$.

β-Weak Hiding If D is honest then for each session sid it holds with probability $\beta > 0$ at a point in time t before any honest party got input $(\text{OPEN}, \mathsf{sid})$ that m is hidden in the view of the adversary at time t, i.e., if $m = m_b$ for (m_0, m_1) picked by the adversary and a uniformly random bit b, then the adversary cannot guess b better than at random.

Construction. The central idea of our wVSS construction is to have a dealer choose a secret seed σ and secret share it unto a random holding committee and put on the ledger an encryption of each of the shares under the public key of the holder, this vector of encryptions is called the setup. If this was done correctly any $t+1$ honest parties can decrypt their shares and use them to reconstruct σ. All randomness for the secret sharing and the encryption will be generated from σ using a PRG. This allows the committee to rerun the setup procedure and check consistency with the published setup after reconstruction. This ensures that if anyone $t+1$ parties can reconstruct to some value σ, then all shares are correct, and therefore all subsets of $t+1$ shares reconstruct to the same σ. We also use randomness derived from σ to encrypt m and include the encryption in the setup. We cannot let the dealer pick the holding committee as we need enough honest parties on it to avoid deadlock of reconstruction. Therefore the holding committee is sampled pseudorandomly from the session identifier sid.

We use n_{VSS} and τ_{VSS} to define the size of the holding committee sampled by the dealer, and the reconstruction threshold in the holding committee, respectively. To ensure weak hiding these parameters should be chosen such that the sampled committee has at most τ_{VSS} corruptions with constant probability at least β, and to ensure liveness less than $n_{\text{VSS}} - \tau_{\text{VSS}}$ should be corrupted except with negligible probability. The scheme makes use of a signature scheme DS (Section 2.2.4), an encryption scheme PKE with full decryption (Section 2.2.1), a threshold secret-sharing scheme TSS (Section 2.2.3), a pseudorandom generator PRG, and a hash function H modelled as a random oracle.

Algorithm 1 Scheme wVSS, algorithm Commit, where an instance sid of wVSS is created at point p on Ledger. Code for process P_i.

```
1: function commit_value(σ, C)
2:     ρ = PRG(H(σ))
3:     m_mask ←$ρ {0,1}^λ
4:     (s_1, ..., s_{n_VSS}) ←$ρ TSS.Share(σ)
5:     for j in C do
6:         r_j ←$ρ {0,1}^λ;   e_j ← PKE.Enc_{ek_j}(s_j, r_j)
7:     return ((e_1, ..., e_{n_VSS}), m_mask)

8: upon input (COMMIT, sid, π, m) where sid = (D, sid') and P_i = D do      // only dealer D
9:     (H_1, ..., H_{n_VSS}) ← SampleCommittee_p(sid, n_VSS)
10:    σ ← DS.Sign_{sk_D}(sid)
11:    ((e_1, ..., e_{n_VSS}), m_mask) ← commit_value(σ, (H_1, ..., H_{n_VSS}))
12:    broadcast (sid, π, (e_1, H_1) ..., (e_{n_VSS}, H_{n_VSS}), m ⊕ m_mask) on Ledger
```

We implement Commit in Algorithm 1. In order to commit to a chosen value m, D first pseudorandomly samples a holding committee of size n_{VSS} (line 9). We say that the committee is "assigned" to D, as D cannot influence it without getting rejected as public output. The

dealer then computes a signature σ on sid, obtaining a unique and unpredictable value. Then D commits to σ by secret-sharing it to the committee. This logic is extracted in an auxiliary function commit_value. It computes a random tape $\rho = \mathsf{PRG}(H(\sigma))$. This random tape is used in all subsequent steps that require randomness. Specifically, in line 3 a random message m_{mask} is sampled, in line 4 the value σ is secret-shared to the members of the holding committee $\mathcal{C}$ using an $(n_{\text{VSS}}, \tau_{\text{VSS}})$-TSS, and in lines 5– 6 the shares of σ are encrypted to the committee members. Each of these values are sampled *pairwise independently* from ρ. Finally, D broadcasts its *VSS setup* on Ledger (line 12). This VSS setup serves as a public output signalling that the message is committed and can at this point only be opened to some unique value–which could be $\perp$. We define the function $\mathsf{PubOutVSSCommit}(\mathsf{Ledger}, (\mathsf{D}, \mathsf{sid}'))$ as the earliest (in Ledger) message $\big((\mathsf{D}, \mathsf{sid}'), \pi, (e_1, \mathsf{H}_1) \ldots, (e_{n_{\text{VSS}}}, \mathsf{H}_{n_{\text{VSS}}}), m\big)$ which is signed by D, where $\mathsf{VerifyCommittee}((\mathsf{D}, \mathsf{sid}'), \mathsf{H}_1, \ldots, \mathsf{H}_{n_{\text{VSS}}}) = 1$. If no such message exists in Ledger, then $\mathsf{PubOutVSSCommit}(\mathsf{Ledger}, \mathsf{sid}) = \perp$.

Algorithm 2 Scheme wVSS, algorithm Open, where an instance sid of wVSS is created at point p on Ledger. Code only for process P_i is in the committee of instance sid, i.e., P_i is one of the H_j in the VSS-setup $(\mathsf{sid}, (e_1, \mathsf{H}_1) \ldots, (e_{n_{\text{VSS}}}, \mathsf{H}_{n_{\text{VSS}}}), m_{masked})$ published on Ledger.

```
State:
13:     validShares[sid] ← []

14: upon input (OPEN, sid) such that PubOutVSSCommit(Ledger_{P_i}, sid) ≠ ⊥ do
15:     let ((e_1, H_1) ..., (e_{n_VSS}, H_{n_VSS}), m_masked) = PubOutVSSCommit(Ledger_{P_i}, sid)
16:     let C = {H_1, ..., H_{n_VSS}}
17:     (s'_i, r'_i) ← Dec_{dk_i}(e_i)
18:     e'_i ← Enc_{ek_i}(s'_i, r'_i)
19:     if e'_i = e_i then
20:         send (SHARE, s'_i, r'_i) to parties in C
21:     else
22:         create zk-proof W_i that e_i decrypts to (s'_i, r'_i)
23:         send (COMPLAINTENCRYPTION, sid, W_i, s'_i, r'_i) to parties in C

24: upon deliver (SHARE, s_j, r_j) from P_j do
25:     if e_j = Enc_{ek_j}(s_j, r_j) then
26:         append s_j to validShares[sid]

27: upon |validShares[sid]| = τ_VSS + 1 do
28:     let (s_{j_1}, ..., s_{j_{τ_VSS+1}}) = validShares[sid]
29:     σ' ← TSS.Reconstruct({s_{j_k}}_{j∈[τ_VSS+1]})
30:     if DS.Ver_{vk_D}(σ', sid) = 0 then
31:         output (DONE-OPEN, sid, ⊥, validShares[sid])
32:     ((e'_1, ..., e'_{n_VSS}), m_mask) ← commit_value(σ')
33:     if (e'_1, ..., e'_{n_VSS}) ≠ (e_1, ..., e_{n_VSS}) then
34:         output (DONE-OPEN, sid, ⊥, validShares[sid])
35:     else
36:         output (DONE-OPEN, sid, m_mask ⊕ m_masked, σ')

37: upon deliver c = (COMPLAINTENCRYPTION, sid, W_j, s_j, r_j) do
38:     e'_j ← PKE.Enc_{ek_j}((s_j, r_j))
39:     if e'_j ≠ e_j and W_j is valid then
40:         output (DONE-OPEN, sid, ⊥, c)
```

We implement Open in Algorithm 2. On input OPEN, and after PubOutVSSCommit(Ledger, sid) $\neq \perp$, a party in the holding committee of the sid instance parses the output as a VSS setup. Party P_i then decrypts e_i to get its share and the randomness used for encryption, (s_i, r_i) (line 17). It then re-encrypts (s_i, r_i) (line 18) to verify that the encryption was done correctly (line 19). If the encryption is valid they send (s_i, r_i) to the other parties, otherwise it sends a verifiable complaint (lines 22–23). The complaint includes a that e_i decrypts to (s'_i, r'_i).

Upon receiving a share from P_j (line 24), party P_i verifies that the share sent by P_j indeed corresponds to the value e_j published on Ledger. If this is the case, the share is considered valid. Observe that the share the dealer created for P_j might be wrong in the first place. This is detected upon reconstruction. Specifically, once $\tau_{\text{VSS}} + 1$ valid shares are received (line 27), P_i runs the reconstruction of TSS to get back some σ', which should be a signature on sid computed by D. Party P_i first verifies the signature and, if valid, it repeats the steps performed by D to secret-share σ' (line 32). Observe that, given σ', all steps in commit_value() are deterministic. Hence, if the reconstructed σ' is the same as the σ computed by the dealer, then commit_value(σ') will return the same values as the ones posted on Ledger by D or we detect that the dealer cheated and output $\perp$. This is checked in line 33. Finally, upon delivering a complaint, sent by some party P_j, party P_i verifies the complaint and, if valid, outputs $\perp$. Note that no party can produce a valid complaint if the check in line 33 goes through. The wVSSVerify check can simply be implemented by a function that when given an encryption complaint checks if it is valid as in line 37, and when given a set of shares checks that they are all valid and treats them as input to the activation rule in line 27 to see that the same output is obtained. When the output of wVSS needs to be distributed to the full set of parties, each party on the committee simply forwards their (DONE-OPEN, sid, m_{P}, π) message to the remaining parties. Note that even though the proof of the outputs can differ, an outside party only needs to receive one. Hence, in gossiping networks the output messages can be deduplicated by only forwarding the first valid one to lower communication complexity.

4 Generating an Unpredictable Seed

In this section we define a seed-generation protocol seed. A seed can be thought of as a perfect coin flip: there is agreement on the output and its value is unpredictable before the protocol starts.

Syntax. The syntax of seed is as follows:

Commit On input (SEED, sid) in a session with session identifier sid a party starts running the commit protocol and may as a result public output PubOutSeedCommit.

Open On input (SEED-OPEN, sid), which must be given after public output PubOutSeedCommit, in a session with id sid a party starts running the opening protocol and may as a result output (DONE-SEED, sid, c), for $c \in \{0,1\}^\lambda$.

Security. The security properties of seed are as follows:

Termination If all honest parties get inputs (SEED, sid) then eventually all honest parties get public output PubOutSeedCommit.

If all honest parties get correct inputs (SEED-OPEN, sid) then eventually all honest parties give an output (DONE-SEED, sid, $\cdot$).

Agreement If two honest parties have outputs $(\text{DONE-SEED}, \mathsf{sid}, c_{\mathsf{P}})$ and $(\text{DONE-SEED}, \mathsf{sid}, c_{\mathsf{Q}})$ then $c_{\mathsf{P}} = c_{\mathsf{Q}}$. Call the common value c_{sid}.

Unpredictability For each session sid it holds that c_{sid} is unpredictable before the first honest party gets input $(\text{SEED-OPEN}, \mathsf{sid})$.

Algorithm 3 Scheme seed, algorithm Commit, where an instance sid of seed is created at some point p on Ledger. Code for process P_i.

```
41: upon input (SEED, sid) do
42:     C ← SampleCommittee(sid, m_SEED)
43:     for j ∈ [m_SEED] such that C[j] = P_i do
44:         r ←$ {0,1}^λ
45:         wVSS(COMMIT, ((P_i, j), sid), r)
```

Algorithm 4 Scheme seed, algorithm Open, where an instance sid of seed is created at some point p on Ledger. Code for process P_i.

```
46: State:
47:     openings[sid] ← []

48: upon input (SEED-OPEN, sid) such that PubOutSeedCommit(sid, Ledger) ≠ ⊥ do
49:     setups ← PubOutSeedCommit(sid, Ledger)
50:     for j ∈ [w_SEED] do
51:         sid_j ← setups[j]
52:         wVSS(OPEN, sid_j)

53: upon deliver (DONE-OPEN, sid_j, r, π) do
54:     if j ∈ [w_SEED] ∧ wVSSVerify(π, r) then
55:         append m to openings[sid]

56: upon |openings| = w_SEED
57:     output (DONE-SEED, sid, ⨁_{r∈openings[sid]} r)
```

Construction. The protocol uses parameters m_{SEED} and w_{SEED}. The idea is to sample m_{SEED} parties in $\mathbb{P}$ to contribute a wVSS setup, asynchronously wait for the first w_{SEED} setups and use the XOR of them as a seed. We discuss in Section 7 how to set these parameters, such that at least one good setup (that is, from an honest proposer and with a committee with at most τ_{VSS} corrupted members) appears on the ledger, except with negligible probability.

The protocol is started at some starting point p of Ledger, with associated stake Σ_p and committee sampling mechanism $\mathsf{SampleCommittee}_p()$. We implement Commit in Algorithm 3 and Open in Algorithm 4. A party that is elected to contribute a wVSS setup (line 42) picks a random r and starts an instance of wVSS to share r (lines 44–45). Once w_{SEED} wVSS protocols with session identifiers $\mathsf{sid}_j = (\mathsf{P}_j, k, \mathsf{sid})$, where $\mathcal{C}[k] = \mathsf{P}_j$, have given public output PubOutVSSCommit on Ledger, then we define PubOutSeedCommit to be the ordered tuple of the session identifiers of the first w_{SEED} such outputs. After this point, the value of the nonce is implicitly defined by the state of the ledger, and on input $(\text{SEED-OPEN}, \mathsf{sid})$, parties start running the Open algorithm on these w_{SEED} instances of wVSS (lines 48–52). By design, the holding committee of each of these instances has enough honest members for wVSS to terminate. The final seed value is defined as the XOR of the values output by each Open (line 57).

5 Weak Honest-Dealer Coin-Flip

In this section we define the weak honest-dealer coin-flip (wHDCF) protocol. In wHDCF there is a designated dealer D, which is one of the participating parties. We assume D is given as part of session identifier, $\mathsf{sid} = (\mathsf{D}, \mathsf{sid}')$, and hence is known by all parties when the instance is created. The scheme is *weak* in the sense that parties may output $\perp$ as the value of the coin, but if two honest parties output a value in $\{0, 1\}$, then it will be the same. It is *honest-dealer* as the coin value becomes predictable for a corrupted D. The scheme makes use of a committee verification mechanism $\mathsf{SampleCommittee}_p()$ proportional to stake at point p (Section 2.1), an encryption scheme with full decryption PKE (Section 2.2.1), and an $(n_{\text{COIN}}, \tau_{\text{COIN}})$-threshold weak coin flip scheme CF (Section 2.2.2). Here n_{COIN} and τ_{COIN} are protocol parameters, for which we choose specific values in Section 7.

Syntax. The syntax of weak honest-dealer coin-flip is as follows:

Deal On input $(\text{DEAL}, \mathsf{sid})$ a participating party starts running the dealing protocol of CF and may as a result produce a public output PubOutSingleDeal.

Flip On input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$, for coin identifier cid, after $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, a party starts running the flip protocol of CF and outputs $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, s, \pi)$, where $s \in \{\perp\} \cup \{0, 1\}^\lambda$ and π is a proof that s is the output of the coinflipping protocol. The proof can be checked by any party P' for which $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}_{\mathsf{P}'}) \neq \perp$ using $\mathsf{wHDCFVerify}(\pi, m)$.

Security. The security properties of wHDCF are as follows.

Termination

(1) If D is honest and all honest parties get input $(\text{DEAL}, \mathsf{sid})$, then eventually $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$.

(2) If, after $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, all honest parties get input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$, then eventually all honest parties give output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, \cdot)$, except with negligible probability.

Weak Agreement If two honest parties output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{P}}, \pi)$ and $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{Q}}, \pi)$, such that $c_{\mathsf{P}} \neq \perp$ and $c_{\mathsf{Q}} \neq \perp$, then $c_{\mathsf{P}} = c_{\mathsf{Q}}$, except with negligible probability. The same holds if $c_{\mathsf{Q}} \neq \perp$ and $\mathsf{wHDCFVerify}(\pi, c_{\mathsf{Q}}) \neq \perp$. Moreover, if D is honest, then no honesty party P outputs $c_{\mathsf{P}} = \perp$.

Honest-Dealer β-Unpredictability If dealer D of session sid is honest, then each coin flip cid is independently unpredictable with some constant probability $\beta > 0$, where β is defined when $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$ and is independent of cid.

In the full version of this work [3] we formalize the *Honest-Dealer β-Unpredictability* property and show the proofs.

Construction. In a high level, the scheme works as follows. Dealer D is assigned a *coin-holding committee* of size n_{COIN} and creates a *coin setup* for an $(n_{\text{COIN}}, \tau_{\text{COIN}})$-threshold coin scheme CF for this committee. Termination is achieved by appropriately setting the parameters and from the pseudorandom nature of the committee: if the dealer completes the setup, there are at least $\tau_{\text{COIN}} + 1$ honest parties in the committee, except with negligible probability. The weak agreement property is achieved by *verifiable* complaints against a corrupted dealer. Upon receiving a valid complaint, a party terminates the Flip protocol outputting $\perp$. If, additionally, D is honest, then our protocol guarantees unpredictability with constant probability β, defined as the probability of having at most τ_{COIN} corruptions in the committee, and depending only on n_{COIN} and τ_{COIN}.

Algorithm 5 Scheme wHDCF, algorithm Deal, where an instance sid of wHDCF is created at point p on Ledger. Code for process P_i.

```
58: upon input (DEAL, sid) where sid = (D, sid') and P_i = D do          // only dealer D
59:     (H_1, ..., H_{n_COIN}) ← SampleCommittee_p(sid, n_COIN),
60:     (vk, sk_1, ..., sk_{n_COIN}) ← CF.Setup(n_COIN, τ_COIN)
61:     for j ∈ [n_COIN] do
62:         r_j ←$ {0,1}^λ;   e_j = PKE.Enc_{ek_j}((sk_j, r_j))
63:     broadcast (sid, vk, (H_1, e_1), ..., (H_{n_COIN}, e_{n_COIN})) on Ledger
```

In Algorithm 5 we implement Deal. The dealer first (line 59) samples the coin-holding committee of size n_{COIN} and then (line 60) uses CF to create a coin setup for it. The coin setup includes secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_{n_{\text{COIN}}}$ and verification key vk. Each secret key sk_i is encrypted to party's P_i long term private key ek_i using a fresh randomness r_i (lines 61–62). The coin setup is broadcast on Ledger. When a coin-setup is included in Ledger we define the public output PubOutSingleDeal(sid, Ledger) as $(\mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}}))$ if the included committee verifies using VerifyCommittee. Otherwise the output is $\perp$.

In Algorithm 6 we implement Flip. Only parties in the coin-holding committee run it. When P_i gets input (FLIP, sid, cid) and $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}_{\mathsf{P}_i}) \neq \perp$, it first reads the coin setup and tries to decrypt e_i to obtain its key share (line 70). Scheme PKE returns sk'_i and the randomness r'_i that D is supposed to have used at encryption time. Party P_i checks whether D has indeed done so by re-encrypting (sk'_i, r'_i) and checking the result against e_i. If it is different, P_i sends a COMPLAINTENCRYPTION message that includes a zero-knowledge proof that e_i decrypts to (sk'_i, r'_i) (lines 71–73) and stops handling the Flip event. Otherwise, P_i can prove correct decryption of e_i in a complaint message by sending (sk'_i, r'_i). Party P_i then verifies its key share against the verification vector vk published in the coin setup, and, if it is invalid, sends a COMPLAINTKEYSHARE message to $\mathcal{C}$ (lines 74–75) and returns. If the check passes, it creates a coin share using the threshold-coin scheme CF (line 76) and sends to the committee $\mathcal{C}$. All complaints are verifiable: COMPLAINTENCRYPTION is valid if the zk-proof W_j, proving that the published e_j decrypts to (sk_j, r_j)), is valid, and the re-encryption of (sk_j, r_j) gives something different from e_j (lines 81–84). COMPLAINTSHARE is valid if the re-encryption of (sk_j, r_j) gives the published e_j and the key share sk_j is deemed invalid by the CF scheme (lines 85–88). Party P_i outputs in two cases, whichever comes first. First, upon collecting $\tau_{\text{COIN}} + 1$ valid coin shares (line 89), in which case the value of the coin is reconstructed using the underlying CF scheme. Second, upon receiving a valid complaint (line 94), in which case a $\perp$ value is output. The wHDCFVerify check can be implemented by a function that when given a complaint checks if it is valid according to the activation rules in line 81 or line 85, and when given a set of shares checks that they are valid and reruns the activation rule in line 89.

6 Weak Multiple-Dealer Coin-Flip

In this section we define the weak multiple-dealer coin-flip (wMDCF) protocol. It is *weak* as it inherits the agreement property from wHDCF: parties may output $\perp$, but if two honest parties output a value in $\{0, 1\}$, then it will be the same. It is called *multiple-dealer* as there are multiple dealers, forming a *proposers committee*, selected pseudorandomly using $\mathsf{SampleCommittee}_p()$. The protocol uses parameters m_{wMDCF} and w_{wMDCF}. Parameter m_{wMDCF} refers to the size of the proposers committee, i.e., the number of parties that are

Algorithm 6 Scheme wHDCF, algorithm Flip (cid), where an instance sid of wHDCF is created at point p on Ledger. Code for process P_i, P_i is one of the H_j in the coin-setup $(\mathsf{sid}, \mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}}))$ published on Ledger.

```
State:
    validShares[sid][cid] ← [], for each sid and cid
    justifiedComplaint[sid][cid] ← ⊥, for each sid and cid
    terminated[sid][cid] ← 0, for each sid and cid

upon input (FLIP, sid, cid) such that PubOutSingleDeal(sid, Ledger) ≠ ⊥ do
    (vk, (H_1, e_1), ..., (H_{n_COIN}, e_{n_COIN})) ← PubOutSingleDeal(sid, Ledger)
    let C = {H_1, ..., H_{n_COIN}}
    (sk'_i, r'_i) = PKE.Dec_{dk_i}(e_i)
    if e_i ≠ PKE.Enc_{ek_i}((sk'_i, r'_i)) then
        create zk-proof W_i that e_i decrypts to (sk'_i, r'_i)
        send (COMPLAINTENCRYPTION, sid, cid, W_i, sk'_i, r'_i) to parties in C; return
    if CF.VerifyKeyShare(vk, i, sk'_i) = 0 then
        send (COMPLAINTKEYSHARE, sid, cid, sk'_i, r'_i) to parties in C; return
    s_i = CF.CreateShare(sk'_i, cid)
    send (COINSHARE, sid, cid, s_i) to parties in C

upon deliver (COINSHARE, sid, cid, s_j) from P_j do
    if CF.VerifyCoinShare(vk, cid, s_j) = 1 then
        append s_j to validShares[sid][cid]

upon deliver c = (COMPLAINTENCRYPTION, sid, cid, W_j, sk_j, r_j) do
    e'_j ← PKE.Enc_{ek_j}((sk_j, r_j))
    if e'_j ≠ e_j and W_j is valid then
        justifiedComplaint[sid][cid] ← c

upon deliver c = (COMPLAINTKEYSHARE, sid, cid, sk_j, r_j) do
    e'_j ← PKE.Enc_{ek_j}((sk_j, r_j))
    if e'_j = e_j and CF.VerifyKeyShare(vk, j, sk_j) = 0 then
        justifiedComplaint[sid][cid] ← c

upon |validShares[sid][cid]| = τ_COIN + 1 and terminated[sid][cid] = 0 do
    let (s_{j_1}, ..., s_{j_{τ_COIN+1}}) = validShares[sid][cid]
    s ← CF.Combine(cid, {s_{j_k}}_{k∈[τ_COIN+1]})
    terminated[sid][cid] ← 1
    output (DONE-FLIP, sid, cid, s, validShares[sid][cid])

upon justifiedComplaint[sid][cid] ≠ ⊥ and terminated[sid][cid] = 0 do
    terminated[sid][cid] ← 1
    output (DONE-FLIP, sid, cid, ⊥, justifiedComplaint[sid][cid])
```

selected to act as a dealer in an instance of wMDCF. Parameter w_{wMDCF} refers to the number of parties in the proposers committee we asynchronously wait for. In Section 7 we show how to set these parameters, such that at least one good setup appears on the ledger, except with negligible probability, and a constant rate γ of the setups are good.

Syntax. The syntax of weak Multiple-Dealer Coin-Flip (wMDCF) is as follows:

Deal On input (DEAL, sid) a participating party starts running the dealing protocol and may as a result help produce a public output PubOutMultiDeal.

Flip On input $(\textsc{flip}, \mathsf{sid}, \mathsf{cid})$ for a coin identifier cid, after $\mathsf{PubOutMultiDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, a party starts running the coin-flip protocol and outputs $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, s)$, where $s \in \{\perp\} \cup \{0,1\}^\lambda$.

Security. The security properties of honest-dealer coin-flip are as follows. For the *agreement* and *unpredictability* properties we use a probability $\gamma > 0$, called the *good-setup probability*, which depends on the parameter w_{wMDCF} and on the hiding probability β of wHDCF, and is constant and independent of sid and cid.

Termination

(1) If all honest parties get input $(\textsc{deal}, \mathsf{sid})$ then eventually there is public output $\mathsf{PubOutMultiDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, except with negligible probability.

(2) If all honest parties get input $(\textsc{flip}, \mathsf{sid}, \mathsf{cid})$ then eventually all honest parties give an output $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, \cdot)$, except with negligible probability.

γ-Agreement For each session sid and coin identifier cid it holds that, if two honest parties output $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, c_\mathsf{P})$ and $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, c_\mathsf{Q})$, such that $c_\mathsf{P} \neq \perp$ and $c_\mathsf{Q} \neq \perp$, then $c_\mathsf{P} = c_\mathsf{Q}$, except with negligible probability. Moreover, with probability γ it holds that no honest party outputs $\perp$ as the value of the coin. All together, this means that, if two honest parties have outputs $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, c_\mathsf{P})$ and $(\textsc{done-flip}, \mathsf{sid}, \mathsf{cid}, c_\mathsf{Q})$, then $c_\mathsf{P} = c_\mathsf{Q} \neq \perp$ with probability γ.

γ-Unpredictability For each session sid and coin identifier cid it holds that the value of coin cid is unpredictable with probability γ.

In the full version of this work [3] we formalize the *agreement* and *unpredictability* properties and show the proofs.

Algorithm 7 Scheme wMDCF, algorithm Deal, where an instance sid of wMDCF is created at some point p on Ledger. Code for process P_i.

```
State:
97:     setups[w_wMDCF] ← []

98: upon input (DEAL, sid) do
99:     C ← SampleCommittee(sid, m_wMDCF)
100:    for j ∈ [m_wMDCF] such that C[j] = P_i do
101:        wHDCF(Deal, ((P_i, j), sid))

102: upon w_wMDCF setups PubOutMultiDeal(((P_j, k), sid), Ledger) ≠ ⊥ where C[k] = P_j
103:    Let setups contain the identifiers which gave public output sorted deterministically
104:    seed(SEED, sid)
```

Construction. On Algorithm 7 we implement Deal. On input $(\textsc{deal}, \mathsf{sid})$, a protocol instance is created with some starting point p. For each time P_i is sampled to be a dealer in a wHDCF instance (line 99), it creates a new instance of wHDCF and runs the Deal algorithm. Every party waits for w_{wMDCF} instances of the wHDCF protocol (started by the dealers sampled in line 99) to give public output on the Ledger. When this happens, parties run an instance of the seed protocol (line 104). This seed will be later used in the Flip algorithm of wMDCF to pseudorandomly choose one of the w_{wMDCF} setups. We define $\mathsf{PubOutMultiDeal} = \mathsf{PubOutSeedOpen}$, so the output of the seed protocol signals the end of the dealing phase.

Algorithm 8 Scheme wMDCF, algorithm Flip (cid), where an instance sid of wMDCF is created at some point p on Ledger. Code for process P_i.

```
105: upon input (FLIP, sid, cid) such that PubOutMultiDeal(sid, Ledger) ≠ ⊥ do
106:     j ← H(sid, cid, seed)
107:     wHDCF(FLIP, setups[j], cid)

108: upon deliver (DONE-FLIP, sid, cid, s, π)
109:     if wHDCFVerify(π, s) then
110:         output (DONE-FLIP, sid, cid, s)
```

In Algorithm 8 we implement Flip. On input (FLIP, sid, cid) and after observing public output PubOutMultiDeal every party P_i uses a cryptographic hash function H, to hash (sid, cid, seed) into $j \in \{1, \ldots, w_{\mathsf{wMDCF}}\}$ (line 106). Then, the algorithm Flip of the wHDCF_j instance is used to compute the value of coin cid. We assume that each party on the committee of the selected wHDCF instance disseminate the output to the ground population.

7 Setting the Parameters

▶ **Definition 2** (Binomial distribution). *Let X a random variable counting the number of successes out of n trials, where success happens with probability p. Then X follows the binomial distribution, i.e., $X \sim \mathcal{B}(n, p)$ and the probability that exactly k successes happen is*

$$\Pr[X = k] = \Pr[\mathcal{B}(n, p) = k] = \binom{n}{k} p^k (1-p)^{(n-k)}. \tag{2}$$

7.1 Sampling a holding committee for wVSS and wHDCF

Let n denote the size of a holding committee and $\tau < n/2$ denote a number, such that the holding committee has at most τ corruptions with a constant probability β, and more than $n - \tau$ corruptions only with a negligible probability $\epsilon = 2^{-\lambda}$, where λ is the security parameter. The idea is the following. If we use a (n, τ)-secet-sharing or common-coin scheme in the committee, then the committee is *hiding* with probability β and *live* with probability $1 - \epsilon$. These capture the parameters of both the wVSS and the wHDCF schemes. In wVSS we have $n \triangleq n_{\text{VSS}}$ and $\tau \triangleq \tau_{\text{VSS}}$, and in wHDCF we have $n \triangleq n_{\text{COIN}}$ and $\tau \triangleq \tau_{\text{COIN}}$.

As discussed earlier, we model a committee-election mechanism as a black-box function SampleCommittee(), which samples parties with probability proportional to their stake at some well-defined point on the ledger. As SampleCommittee() does sampling with replacement, it can be modelled with a binomial distribution. Using (2) we have that $\beta = \sum_{k=0}^{\tau} \Pr[\mathcal{B}(n, 1-p) = k]$ and $\epsilon = \sum_{k=n-\tau+1}^{n} \Pr[\mathcal{B}(n, 1-p) = k]$, for $p = 2/3$. In Table 1 we show various combinations for n and τ, such that $\epsilon \leq 2^{-\lambda}$ for $\lambda = 60$, and the resulting hiding probability β.

7.2 Sampling a proposer committee for seed and wMDCF

In protocols seed and wMDCF parties have a chance to participate in the *proposers committee*, i.e., to win the right to become a dealer in a wVSS or wHDCF instance, respectively. Parties are again sampled using SampleCommittee (Section 7.1), which returns a committee of size

m, but the protocols only wait for the first w setups to appear on Ledger and only use those. In seed, we have $m \triangleq m_{\text{SEED}}$ and $w \triangleq w_{\text{SEED}}$, and in wMDCF we have $m \triangleq m_{\text{wMDCF}}$ and $w = w_{\text{wMDCF}}$.

Necessary conditions. As before, we need to make sure that, except with negligible probability $\epsilon = 2^{-\lambda}$, there are at least w honest parties on the committee to ensure termination. This is bounded as ϵ in Section 7.1 but with n and τ replaced by m and $w - 1$ respectively. But now we additionally need to make sure that, except with negligible probability at least one of the w setups that appear on Ledger is a *good setup*, that is, from an honest party who sampled a committee with less than τ corruptions. This condition corresponds exactly to the setup in Section 7.1 being hiding, but with the probability p changed to account for the fact that we are interested in the probability of an honest party *who provided a good setup*. Since an honest dealer has a β (which depends on the parameters of the subprotocol) probability of providing a good setup, we set $p = \beta \cdot 2/3$ and require $\sum_{k=0}^{w-1} \Pr[\mathcal{B}(m, 1-p) = k] \geq 1 - 2^{-\lambda}$.

Good-setup probability. Finally, specifically for wMDCF, we calculate the probability γ, defined in Section 6, that a setup published on Ledger is good, i.e., the probability of getting an unpredictable and agreed upon value in each coin flip. We derive this from the expected number of bad setups, which (by linearity of expectation) is $m \cdot (1 - \beta \cdot \frac{2}{3})$, and from the fact that the adversary can schedule the order of messages, causing all bad setups and, hence, only $w - m \cdot (1 - \beta \cdot \frac{2}{3}))$ good setups, to appear on Ledger. This gives us the fraction of good setups that in expectation appear on the ledger as

$$\gamma = \frac{w - (m \cdot (1 - \beta \cdot \frac{2}{3}))}{w}. \tag{3}$$

Putting it all together. We show the resulting parameters with $\lambda = 60$ bits of security in Table 1. As an example, for a holding committee with size $n = 259$ and reconstruction threshold $\tau = 103$, we get hiding probability $\beta = 98.7\%$. Then we can sample a proposers committee of size $m = 653$ and wait for $w = 327$. This results in $84,693$ encrypted shares being posted on the Ledger, and for wMDCF it gives a good-setup probability $\gamma = 31.8\%$.

8 Analysis of Communication Complexity

To demonstrate the power of being able to sample concretely small committees, we analyze the concrete complexity of our protocols. Note that a purely asymptotic analysis would not show any gains over simply using a state of the art ADKG protocol with subset sampling and near optimal resilience. We give all sizes in *bits*, but for simplicity we treat group and field elements as λ bits. For instance, we use 3λ as the size of an encrypted share, which (using Section 2.2.1) consists of 2 group elements and a symmetrically encrypted share of a secret of size λ. This would not be precise for concrete instantiations, but it would only change our estimates by a small constant factor which depends, for example, on the concrete curves being employed.

We define ATOB complexity as the cost of including a message of a given size in Ledger. In the following "broadcast" refers to broadcasting through the ATOB and "multicast" refers to a party sending a message to all parties. As the communication cost of a broadcast and multicast depend on the implementation, we keep these costs opaque and report results as a number of broadcasts and multicasts required. For inter-committee communication we assume point-to-point channels and give results in total number of bits sent.

Table 1 This table shows possible values (subject to conditions in Section 7.1) for the *holding committee* parameters, n and τ, and the resulting hiding probability β. For each obtainable β, it shows possible values (subject to conditions in Section 7.2) for the *proposers committee* parameters, m and w, and the resulting good-setup probability γ. In both seed and wMDCF schemes, each of the w dealers encrypts keys for a committee of size n, which gives a total of $m * w$ encryptions.

n	τ	β	m	w	γ	$n \cdot w$
653	320	$> 1 - 2^{60}$	653	321	32.2%	$209.6K$
300	125	99.9%	653	322	32.3%	$96.6K$
280	114	99.6%	653	323	32.1%	$90.4K$
275	111	99.4%	653	324	32.0%	$89.1K$
271	109	99.3%	653	325	32.0%	$88.1K$
265	106	99.0%	653	326	31.9%	$86.4K$
261	104	98.8%	653	327	31.9%	$85.3K$
259	103	98.7%	653	327	31.8%	$84.7K$
257	102	98.6%	659	330	31.6%	$84.8K$
256	101	98.3%	672	337	31.3%	$86.3K$
254	100	98.1%	682	342	31.1%	$86.9K$
252	99	98.0%	692	347	30.8%	$87.4K$

The wVSS protocol has an ATOB complexity of 1 message of size $n_{\text{VSS}} \cdot 3\lambda + \lambda$ with the encrypted shares and masked message in the setup. To distribute the output either the secret of size λ is multicast or a complaint of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$ (in case of correctly encrypted shares reconstructing an inconsistent setup) is multicast. A priori, every member of the committee needs to multicast the output, giving a multicast complexity of n_{VSS} messages of size λ (or of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$, in which case the dealer can be slashed). The remaining interaction consists of n_{VSS} committee members sending one message with a decrypted share or a complaint of total size at most 4λ to the rest of the committee, resulting in a total message complexity of at most ${n_{\text{VSS}}}^2 \cdot 4\lambda$.

The seed protocol does not add any interaction besides running m_{SEED} instances of wVSS. Only the first w_{SEED} of those to make it onto the ledger will result in interaction between committe members, so the communication complexity is at most $m_{\text{SEED}} \cdot {n_{\text{VSS}}}^2 \cdot 4\lambda$. The ATOB complexity of the deal phase of seed is m_{SEED} messages of size $n_{\text{VSS}} \cdot 3\lambda + \lambda$. To disseminate the outputs there is an additional $(w_{\text{SEED}} - s) \cdot n_{\text{VSS}}$ multicasts of size λ and $s \cdot n_{\text{VSS}}$ multicasts of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$, where s is the number of parties that can be slashed.

The wHDCF protocol has an ATOB complexity of 1 message of size $n_{\text{COIN}} \cdot 3\lambda + \lambda$ and no additional communication in the initial setup phase. The message complexity of each coin flip is at most ${n_{\text{COIN}}}^2 \cdot 4\lambda$ to reconstruct the coin (or $\perp$) in the committee, and then to disseminate the value to the ground population each committee member multicasts the reconstructed coin or a complaint of size at most 4λ, resulting in at most ${n_{\text{COIN}}}^2 \cdot 4\lambda$ bits communicated in addition to n_{COIN} multicasts of size 4λ. The deal phase of the wMDCF protocol has the same complexity as m_{wMDCF} deal phases of wHDCF and a single run of the seed protocol. That is, an ATOB complexity of m_{wMDCF} messages of size $n_{\text{COIN}} \cdot 3\lambda + \lambda$ and m_{SEED} messages of size $n_{\text{VSS}} \cdot 3\lambda + \lambda$, and a multicast complexity of $w_{\text{SEED}} \cdot n_{\text{VSS}}$ messages of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$, in addition to a communication complexity of $m_{\text{SEED}} \cdot {n_{\text{VSS}}}^2 \cdot 4\lambda$ bits. Whenever a coin needs to be flipped using wMDCF, the message complexity is that of running the selected wHDCF protcol.

To refresh the setup after the stake distribution has changed, one would need to first run an instance of the seed protocol and then the deal phase of the wMDCF protocol. Using the best parameters in Table 1 the concrete cost of refreshing the setup is 1959 messages of size 778λ, and $169,386$ multicasts of size at most 208λ. The communication complexity of flipping a coin and disseminating it to all parties is $259^2 \cdot 4\lambda$ in addition to 259 multicasts of 4λ bits. Employing the optimizations in Remark 3 reduces the multicast complexity of refreshing the setup to 654 messages of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$. Similarly the cost of distributing a coin becomes the same as 1 multicast of size 4λ.

If we were to assume $t < 0.3n$ in the paradigm of "subset sampling with almost optimal resilience" [5], and need a committee with honest supermajority with probability $1-2^{-60}$, then one would need to sample a committee with 16037 parties [22, Table 1]. If we then instantiate a state of the art ADKG protocol with $O(n^3\lambda)$ communication using the committee, assuming for the sake of an example the concrete cost is $n^3\lambda$, then we get a complexity of $> 4 \cdot 10^{12}\lambda$. It is then clear that our approach is far cheaper for all but extremely large values of n.

▶ Remark 3 (Deduplicating multicasts). Notice that each party only needs to receive a single proof of output for each of the w_{SEED} wVSS setups. Since large-scale P2P networks usually employ gossiping with deduplication of previously forwarded messages, each node can consider different output justifications from the same wVSS as identical for the purpose of decuplication. We conjecture that in most gossip-based P2P networks this results in a communication cost which is less than that of a single multicast, as it can be seen as a multicast from a single source which has gotten a headstart by being predistributed to $O(\lambda)$ nodes. With this instantiation the cost of disseminating the wVSS outputs from the seed protocol becomes the same as multicasting w_{SEED} messages of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$ over the gossip network. The same deduplication trick can be employed when disseminating the coin flips, reducing the cost of distributing the coin to the same as 1 multicast of size 4λ.

9 Conclusion

In this work we have presented protocols for generating randomness in an asynchronous PoS setting with dynamic participation. They are practical and concretely efficient, employ no trusted setup, and they make use of small committees. We have computed concrete numbers for the committees. Specifically, we can have a committee of $m = 653$ proposers, each generating a setup for $n = 359$ holders, resulting in approx. $85K$ encrypted values posted on Ledger. For $\kappa = 60$ bits of security and assuming optimal corruption $1/3$ in the ground population, our protocols are live with all but negligible probability. Our common-coin protocol is unpredictable and agreed-upon with probability approx. 31.8%, and, as it is based on threshold cryptography, the setup can be used for a flipping a polynomial number of coins. These committee sizes result from the fact that we require not all but only a constant factor of our setups to be good.

References

1 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptively secure packed asynchronous verifiable secret sharing and asynchronous distributed key generation. *IACR Cryptol. ePrint Arch.*, page 1759, 2022.

2 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *PODC*, pages 363–373. ACM, 2021.

3 Orestis Alpos, Christian Cachin, Simon Holmgaard Kamp, and Jesper Buus Nielsen. Practical large-scale proof-of-stake asynchronous total-order broadcast. *IACR Cryptol. ePrint Arch.*, page 1103, 2023.

4 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30. ACM, 1983.

5 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In *TCC (1)*, volume 12550 of *Lecture Notes in Computer Science*, pages 353–380. Springer, 2020.

6 Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003.

7 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788. Springer, 2018.

8 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

9 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

10 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

11 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *PODC*, pages 81–91. ACM, 2022.

12 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, pages 42–51. ACM, 1993.

13 Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *ACNS*, volume 10355 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2017.

14 Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. In *ASIACRYPT (3)*, volume 12493 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

15 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: super fast and partition resilient byzantine agreement. *IACR Cryptol. ePrint Arch.*, page 377, 2018.

16 Kevin Choi, Arasu Arun, Nirvan Tyagi, and Joseph Bonneau. Bicorn: An optimistically efficient distributed randomness beacon. *IACR Cryptol. ePrint Arch.*, page 221, 2023.

17 Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. *IACR Cryptol. ePrint Arch.*, page 728, 2023.

18 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement WHP. In *DISC*, volume 179 of *LIPIcs*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

19 Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *IEEE Symposium on Security and Privacy*, pages 2502–2517. IEEE, 2022.

20 Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *IEEE Symposium on Security and Privacy*, pages 2518–2534. IEEE, 2022.

21 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

22 Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy. In *CCS*, pages 683–696. ACM, 2022.

23 Drand. A distributed randomness beacon daemon, 2022. URL: `https://drand.love`.

24 Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–437. IEEE Computer Society, 1987.

25 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In *PODS*, pages 1–7. ACM, 1983.

26 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68. ACM, 2017.

27 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2):13:1–13:21, 2016.

28 Valerie King and Jared Saia. Correction to byzantine agreement in expected polynomial time, JACM 2016. *CoRR*, abs/1812.10169, 2018.

29 Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, page 366, 2015.

30 Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *FOCS*, pages 120–130. IEEE Computer Society, 1999.

31 Arpita Patra, Ashish Choudhury, and C. Pandu Rangan. Asynchronous byzantine agreement with optimal resilience. *Distributed Comput.*, 27(2):111–146, 2014.

32 Protocol Labs. Filecoin: A decentralized storage network. `https://filecoin.io/filecoin.pdf`, 2017.

33 Michael O. Rabin. Randomized byzantine generals. In *FOCS*, pages 403–409. IEEE Computer Society, 1983.

34 Mayank Raikwar and Danilo Gligoroski. Sok: Decentralized randomness beacon protocols. In *ACISP*, volume 13494 of *Lecture Notes in Computer Science*, pages 420–446. Springer, 2022.

35 David A. Wagner Ronald L. Rivest, Adi Shamir. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, 1996.

36 Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar R. Weippl. Hydrand: Efficient continuous distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 73–89. IEEE, 2020.

37 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

38 Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460. IEEE Computer Society, 2017.