Nearly Quadratic Asynchronous Distributed Key Generation from Recursive Consensus

Ittai Abraham[1], Renas Bacho[*2], Julian Loss[†2], and Gilad Stern[3,4]

[1]a16z
[2]Ruhr University Bochum
[3]Tel Aviv University
[4]Hebrew University

June 19, 2026

Abstract

Distributed key generation (DKG) is a Byzantine fault tolerant protocol among a set of n parties that enables generation of a public-secret key pair for threshold cryptosystems without a trusted third party. It serves as a building block for many decentralized protocols, such as randomness beacons, threshold encryption and signing, consensus, and general multi-party computation. Of particular importance are *asynchronous* DKG protocols with low communication and round complexity. However, all existing constructions for asynchronous DKG require cubic or higher communication costs in the number of parties (including setup costs) to terminate. This severely limits their practicality, especially in large-scale systems such as the internet or blockchains, where communication and round complexity are the main bottlenecks to efficiency.

In this paper, we present the first asynchronous DKG protocol that overcomes this limitation. Our protocol has the following desirable properties. It is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties. Parameterized by a natural number $1 \leq k \leq \log n$, it has an expected communication cost of $O(\lambda^2 n^{2+1/k})$ bits and terminates in expected $O(k)$ rounds, where λ denotes the security parameter. Furthermore, for setup, it requires a plain public key infrastructure (PKI) and random oracles, which is standard in the literature on DKG protocols. From a conceptual perspective, our main contribution is to adapt the classical recursive paradigm (for distributed protocols) from the synchronous to the asynchronous model and to apply it to asynchronous DKG.

1 Introduction

Distributed Key Generation (DKG) [Ped91, BK25] allows a set of n parties $\{1, \ldots, n\}$ to jointly generate a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$, where the secret key is shared between the parties via a threshold secret sharing. Cryptographic operations that require the secret key can then be performed by a certain threshold number of parties but not by fewer. Key applications of DKG protocols are threshold encryption and threshold signature schemes. In fact, many state-of-the-art Byzantine fault tolerant (BFT) distributed protocols [YMR+19, GKKS+22] use threshold signatures to reduce communication complexity or threshold encryption to prevent censorship and reduce front running. Over the last four decades, an extensive amount of DKG protocols have been proposed [Ped91, CGJ+99, DXKKR23]. However, most of these protocols assume a *synchronous* network and provide no security guarantees once the network becomes unstable, an event that is not uncommon on the internet. A more recent line of work [KKMS20, AJM+21] focuses on

[*]This work was done while this author was at CISPA.
[†]This work was done while this author was at CISPA.

solving DKG in the *asynchronous* network model (referred to as *asynchronous DKG*, or *ADKG* for short), where message delivery may have unbounded delay [LSP82].

Existing constructions for asynchronous DKG[1] require cubic or higher communication costs (in the number of involved parties). This severely limits their practicality, especially in large-scale systems such as the Internet or blockchains, where even just reading a cubic number of bits is impractical.

In concurrent and independent work, [FT25, FGL+25] has recently shown how to obtain an ADKG protocol with just $\tilde{O}(n^2)$ total network writes and $\tilde{O}(n^3)$ total network reads. This is achieved assuming secure erasures and a setup phase that allows each party to post $O(n)$ bits on the public bulletin board, in addition to a global $O(n)$-sized structured reference string (SRS)[2]. This setup assumption is significantly more powerful than a standard/plain PKI model, in which each party is only allowed to post $\tilde{O}(1)$ bits on the public bulletin board. Moreover, this setup requires a total of $\tilde{O}(n^3)$ network reads from the bulletin board[3].

A natural question remains:

Can we design an optimally-resilient, adaptively secure asynchronous DKG protocol with subcubic communication complexity for both reads and writes (including from the bulletin board), from just a plain PKI setup?

1.1 Our Contribution

We address this question by presenting an (optimally-resilient) asynchronous DKG protocol with subcubic communication cost from just a plain PKI setup, random oracles, and secure erasures. Our protocol has the following desirable properties:

- **Security:** Our protocol is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties. This is optimal for asynchronous DKG.
- **Efficiency:** For any $1 \leq k \leq \log n$, our protocol has an expected communication cost of $O(\lambda^2 n^{2+1/k})$ bits and terminates in expected $O(k)$ rounds, where λ is the security parameter.

Conceptually, our main contribution is to adapt the classical synchronous recursive paradigm of [CW92, BGP92] to the asynchronous model and to apply it to asynchronous DKG. For a comparison with existing asynchronous DKG protocols, see Table 1 below. We discuss more related work on DKG and consensus protocols in Appendix A, and refer to [BK25] for a systematization of the extensive DKG literature.

We believe that even if better ADKG protocols will eventually be found, there is a conceptual value in this work as it highlights ways in which the classical synchronous recursive paradigm of [CW92, BGP92] can be applied to the asynchronous model.

2 Technical Overview

We give an overview of the key challenges encountered in designing a subcubic asynchronous DKG, and our novel ideas to address them.

2.1 Common Approach for ADKG

We first review the common approach that underpins almost all existing asynchronous DKG protocols. Recall that the goal of a DKG protocol among a set of n parties, where up to f of them can be malicious,

[1] We note that all protocols are resilient to a Byzantine adversary that can corrupt up to $f < n/3$ parties, which is optimal for asynchronous DKG [BCLZL23, FLM85]. In this work, we also focus on optimally-resilient asynchronous DKG.

[2] This SRS itself requires $O(\lambda n^3)$ communication and $O(\log n)$ rounds in asynchrony [DXR24].

[3] We note that there may be other setup assumptions in which [FT25, FGL+25] obtain subcubic reads, but those require a trusted party that can read the bulletin board, compute, and then post $O(n)$ back to the bulletin board. We believe that this setup assumption is too strong and focus our work on the plain PKI setup assumption.

Table 1: Comparison of existing asynchronous DKG protocols.

Protocol	Resilience	Communication	Rounds	Setup Cost	Setup	Additional Notes
Kokoris et al. [KKMS20]	$f < n/3$	$O(\lambda n^4)$	$O(n)$	$O(\lambda n^2)$	RO, PKI	without pairings
Abraham et al. [AJM+21]	$f < n/3$	$O(\lambda n^3 \log n)$	$O(1)$	$O(\lambda n^2)$	RO, PKI	$O(1)$-round MVBA
Gao et al. [GLL+22a]	$f < n/3$	$O(\lambda n^3)$	$O(1)$	$O(\lambda n^2)$	RO, PKI	variant of [AJM+21]
Das et al. [DYX+22]	$f < n/3$	$O(\lambda n^3)$	$O(\log n)$	$O(\lambda n^2)$	RO, PKI	verifiable encryption
Das et al. [DXKKR23]	$f < n/3$	$O(\lambda n^3)$	$O(\log n)$	$O(\lambda n^2)$	RO, PKI	without pairings
Zhang et al. [ZDL+23]	$f < n/3$	$O(\lambda n^4)$	$O(1)$	None	Auth. Channels	coll-resistant hashes
Abraham et al. [AJM+23]	$f < n/3$	$O(\lambda n^3)$	$O(1)$	$O(\lambda n^3)$	RO, PKI, PoT	optimal packed AVSS
Feng et al. [FT25, FGL+25]	$f < n/3$	$O(\lambda^2 n^2)$	$O(1)$	$O(\lambda n^3)$	RO, PKI, Silent	silent setup, SNARKs
Our ADKG	$f < n/3$	$O(\lambda^2 n^{2+1/k})$	$O(k)$	$O(\lambda n^2)$	RO, PKI	$k \leq \log n$, SNARKs

Note. Communication and round complexity are given in expected numbers, as all these protocols are randomized. On **Setup Cost**: The silent setup in [FT25] requires each party to register an $O(\lambda n)$-bit sized public key and thus read $\Omega(\lambda n^2)$ bits during setup, yielding $\Omega(\lambda n^3)$ total setup communication cost. Moreover, the silent setup requires a powers-of-tau (PoT) setup, which itself requires $O(\lambda n^3)$ communication and $O(\log n)$ rounds in asynchrony [DXR24]. We discuss [FT25] and the "silent setup" in more detail in Appendix A. The other protocols (with plain PKI) register $O(\lambda)$-sized public keys during setup. Our protocol and [FT25, FGL+25] require erasures only for adaptive security.

is to generate a secret key sk that is shared among all parties through a threshold secret sharing scheme, so that each party i obtains a share sk_i of the secret key. For the following discussion, we denote the degree of the polynomial interpolating these key shares by t. We note that asynchronous DKG protocols can have a dual-threshold $t \in [f, n-f)$, while synchronous protocols typically have $t = f < n/2$.

High-level structure. Abstractly speaking, existing protocols for ADKG[4] typically follow the same two-phase structure consisting of a *sharing phase* and a *consensus phase*. In the initial sharing phase, each party $i \in [n]$ locally chooses a random secret s_i (in the space of secrets) and shares it among all parties via an asynchronous verifiable secret sharing (AVSS) scheme [CGMA85, CKLS02], where s_i lies on some polynomial f_i of degree t. From the AVSS of party i, each other party $j \in [n]$ should then obtain a share $s_{i,j} := f_i(j)$ of the secret s_i. However, because the network is asynchronous, an AVSS instance may have completed at some honest parties but not yet at all of them, as some may experience delays. (However, we are still guaranteed that if it completes at one honest party, it will *eventually* complete at all honest parties.) Therefore, in an ADKG, each party i waits for $f+1$ (or sometimes $n-f$) AVSS instances to complete locally and stores the corresponding indices of the dealers in a set $T_i \subseteq [n]$. In particular, each party might have a different set T_i of locally completed AVSS instances. Therefore, in the subsequent consensus phase, parties agree on a subset $\mathcal{Q} \subseteq [n]$ of at least $f+1$ parties whose AVSS instances completed. This set $\mathcal{Q}$ of qualified parties specifies which secrets should be included in the secret key. Concretely, the secret key is then (implicitly) given by the sum of their individual secrets $\mathsf{sk} := \sum_{i \in \mathcal{Q}} s_i$, while each party i's secret key share is the sum of shares of their individual secrets $\mathsf{sk}_i := \sum_{j \in \mathcal{Q}} s_{j,i}$. It is crucial that the secret key includes at least $f+1$ individual secrets (i.e., $|\mathcal{Q}| \geq f+1$), ensuring that an adversary corrupting f parties does not know all the secrets included in the key.

Inherent lower bounds for AVSS. To recap, we see that the sharing phase consists of n parallel instances of an AVSS scheme, where each of the n parties acts as a dealer in one of these instances. However, well-known results on AVSS show an inherent lower bound of $\Omega(n^2)$ message complexity in communication per instance [ACD+23, BDK13].This yields an inherent lower bound of $\Omega(n^3)$ messages for the initial sharing phase only. But what about the consensus phase?

FLP impossibility for asynchronous consensus. The consensus phase is typically implemented via a multivalued validated Byzantine agreement (MVBA) protocol [CKPS01], also called *validated asynchronous Byzantine agreement (VABA)* in some works [AMS19]. In an MVBA protocol, each party $i \in [n]$ has a

[4]Synchronous DKG protocols have the same high-level structure. However, there are crucial differences between the two network settings, which we will point out later.

value v_i as input and all parties agree on a common output value v satisfying the external validity condition. However, all existing constructions for MVBA either assume a trusted dealer at setup that threshold-shares a secret key between the parties (used to implement non-interactive threshold signatures and strong common coins) [AMS19, LLTW20, GLL+22b] or have a communication cost of $\Omega(\lambda n^3)$ bits [AJM+23, AJM+21, DDL+24, DXKKR23], or both. In particular, all prior MVBA protocols *without* a threshold key setup have a communication cost of $\Omega(\lambda n^3)$ bits. We note that the protocols in [AMS19, LLTW20, GLL+22b], which assume a threshold key setup, achieve the optimal $O(n^2)$ message complexity [ACD+23, DR85] and could implement the non-interactive threshold signatures with a transparent setup [ACR21, Fu25]. But still, these protocols critically rely on strong common coins, which require a threshold key setup.

The main difficulty in the design of asynchronous consensus protocols stems from the famous FLP impossibility result by Fischer, Lynch, and Paterson [FLP85], which states that reaching consensus in asynchronous networks necessitates randomness. While the randomness can be generated locally as in [Bra84], this results in an inefficient protocol with $O(2^n)$ multiplicative overhead in communication and rounds. Therefore, to obtain an efficient protocol, the randomness is typically generated in the form of common randomness. But common randomness is what a threshold key setup provides, which itself requires an ADKG (or a trusted dealer). We stress that this issue is not inherent to the synchronous setting, where deterministic consensus exists [DS83, MR21]. In Appendix A.1, we recall the most efficient way from the literature to resolve this problem.

2.2 Recent Progress in Synchrony

The abstract structure of DKG protocols in the synchronous network is identical: a sharing phase and a consensus phase. The sharing phase of almost all existing synchronous DKG protocols is executed by n parallel instances of a verifiable secret sharing (VSS) scheme, one instance per party. Again, existing results show an inherent lower bound of $\Omega(n^2)$ message complexity in communication per instance [ACD+23, DR85], yielding $\Omega(n^3)$ total message complexity. In a recent major advance, two concurrent works [BLL+24, FLT24a] overcome this barrier and present (synchronous) DKG protocols with *subcubic* communication complexity. Concretely, the protocol in [BLL+24] has $O(\lambda n^2 \log n)$ bits of communication, while [FLT24a] has $O(\lambda n^{2.5})$ bits of communication. To achieve this, these works use aggregatable PVSS [GJM+21, BL23] in combination with a recursive structure of the protocol. We briefly recall the design in [BLL+24] and explain why their approach fails in the asynchronous setting. This will serve as an important starting point for our ideas.

The approach in [BLL+24]. Recall that a publicly verifiable secret sharing (PVSS) scheme [Sta96] allows to share a secret s among a set of parties via a single transcript Trx, such that anyone can verify that the secret was shared correctly (it requires only a PKI setup). From the transcript Trx, each party i can then extract its share s_i of the secret. Further, an aggregatable PVSS scheme [GJM+21, BL23] additionally allows to aggregate multiple transcripts $\mathsf{Trx}_1, \ldots, \mathsf{Trx}_l$ into a single transcript Trx (while maintaining public verifiability). PVSS schemes allow for simple and modular constructions of DKG protocols: First, each party $i \in [n]$ generates a transcript Trx_i for a random secret s_i and sends it to all parties. Then, the parties agree on a set of at least $f+1$ valid transcripts, from which they can directly derive their secret key share (by extracting its share from each transcript and summing them). This can be viewed as each party i proposing a secret s_i (encoded in Trx_i), after which the parties agree on $f+1$ such proposals to use for the secret key. Since agreement on a single proposal requires $\Omega(n^2)$ messages [ACD+23, DR85], this yields a total of $\Omega(n^3)$ messages. But what if there are only two proposals (i.e., PVSS transcripts) to begin with?

That is exactly the idea in [BLL+24] and where the recursion comes in. The goal is to establish a single PVSS transcript Trx that recursively aggregates all (valid) individual transcripts $\mathsf{Trx}_1, \ldots, \mathsf{Trx}_n$ from the n parties. Their protocol first partitions the set $\mathcal{P}$ of n parties into two smaller sets $\mathcal{P}_1$ and $\mathcal{P}_2$ of roughly the same size (i.e., $\mathcal{P}_1 \cup \mathcal{P}_2 = \mathcal{P}$). Then, it runs the protocol recursively on each child $\mathcal{P}_i$, which generates a PVSS transcript $\mathsf{Trx}[\mathcal{P}_i]$ (at this point, the transcript is known only to the parties in $\mathcal{P}_i$ and it is itself formed by calling its children, etc.). Subsequently, each child $\mathcal{P}_i$ proposes its transcript to its parent node $\mathcal{P}$ (i.e., to all parties in $\mathcal{P}$). And the parent then runs two instances of a deterministic consensus protocol [MR21] to establish consensus among all parties in $\mathcal{P}$ on both proposals (one instance per child). Finally, the parent

keeps the valid transcripts and aggregates them. Importantly, the authors use a *deterministic* consensus protocol [MR21] with optimal communication cost of $O(\lambda n^2)$ bits and $O(n)$ rounds.[5]

Why does it work in synchrony? Note that their DKG protocol assumes $f < n/2$ (i.e., there is an honest majority). That means that the parent node $\mathcal{P}$ in the first recursion step has an honest majority and thus behaves correctly (we say *the parent is "good"*). Therefore, the consensus protocols run by $\mathcal{P}$ ensure agreement among all parties on the two proposals $\mathsf{Trx}[\mathcal{P}_1]$ and $\mathsf{Trx}[\mathcal{P}_2]$. Subsequent steps are deterministic (i.e., checking validity of each transcript and their aggregation), which preserves the consistency among all parties. Thus, given that $\mathcal{P}$ has an honest majority, the recursion step ensures that, upon termination, all parties output a common (aggregated) transcript Trx. Further, termination is guaranteed because all steps are deterministic and communication is synchronous: in the first round, each child proposes its value; in the second round, the parent deterministically reaches consensus on both proposals; in the third round, the parent aggregates the valid proposed transcripts, and then proceeds to the next recursion level.

Why does the aggregated transcript at the top-level have honest contribution? This follows by an inductive argument, based on the key observation that, regardless of which f parties the adversary (adaptively) corrupts, there is always at least one honest branch in the recursion tree that successfully completes every step of the recursion (at the base level of recursion, each party generates its own PVSS transcript). In more detail: Since $\mathcal{P}$ is good (i.e., has an honest majority), at least one of its two children $\mathcal{P}_1$ and $\mathcal{P}_2$ must also be good, say $\mathcal{P}_1$. The same reasoning then applies recursively to $\mathcal{P}_1$ and its children $\mathcal{P}_1 = \mathcal{P}_{1,1} \cup \mathcal{P}_{1,2}$, and so on down the recursion tree. At the base level of recursion, each child corresponds to a single party. Since both are honest at this level, their initially sampled PVSS transcripts are guaranteed to be included in every higher-level aggregate (a good parent will always accept a good child's proposal), and thus in the final top-level aggregated transcript. In particular, the final output aggregated PVSS transcript will always have an honest contribution, ensuring it remains secret from the adversary.

2.2.1 Why Does It Fail in Asynchrony?

There are several points where the previous approach fails in the asynchronous setting, which we will explain here. Given that we have $f < n/3$ in asynchrony, we call a parent (resp. child) *"good"* if it has an honest super-majority (i.e., the fraction of honest parties is at least $\frac{2}{3}$).

Malicious branch succeeds. First, given that $\mathcal{P}$ is good, we are only guaranteed that one of its children, say $\mathcal{P}_1$, is good - as in the synchronous setting. The other child, $\mathcal{P}_2$, may be malicious and remain silent, proposing nothing. In synchrony, parties have synchronized clocks, so silent parties can be detected. In asynchrony detection is not possible: a child may simply be slow and experience delays. Thus, a parent waiting for both proposals may get stuck indefinitely and never receive a proposal from the malicious child. Therefore, to make progress and maintain liveness, the parent can only wait for at most one child proposal. But this enables a subtle, devious attack: The adversary can follow the protocol correctly but delay the honest child's proposal, causing its own proposal to be accepted - it can make a *malicious branch* succeed. As a consequence, the final aggregate transcript would contain no honest contributions, enabling the adversary to learn the encoded secret (i.e., the secret key), which violates the secrecy property of ADKG.

No deterministic asynchronous consensus. Second, the parent must run a consensus protocol to ensure agreement among all parties on the two proposals. Even if both children could be forced to propose (and we sidestep the above problem), a malicious child could still propose multiple transcripts, requiring the parent to agree on one. In synchrony, this is handled by an optimal-communication *deterministic* consensus protocol. In asynchrony, however, the FLP impossibility rules out deterministic consensus. Therefore, randomness is necessary, and to avoid exponential blow-ups in cost, the randomness should be somehow shared among parties rather than purely local. Thus, the key question is how to obtain the required randomness. Previous approaches in asynchrony typically run n parallel instances of an AVSS scheme and derive random ranks for parties from these. This allows to efficiently generate a weak (rather than strong) common coin, which

[5] We note that the protocol in [MR21] uses non-interactive threshold signatures. But as observed and used in [BLL+24], these can be implemented with a transparent setup [ACR21, Fu25].

is sufficient for building an asynchronous consensus protocol using the Gather framework of Abraham et al. [AJM+21]. But again, the n instances of an AVSS require $\Omega(n^3)$ communication. To make things worse, the Gather protocol in [AJM+21] itself has $O(\lambda n^3)$ communication cost.

How to break the log-round barrier? Third, the recursion has a depth of $\log n$, which already implies $\Omega(\log n)$ total rounds - even if we had a constant-round asynchronous consensus (and resolved the above problem). A potential idea, which we will indeed adopt later, is to generalize the binary tree into a tree of depth k and degree $\ell = n^{1/k}$ for any $1 \leq k \leq \log n$, so that each parent now has ℓ children. With only k levels, it seems hopeful to go below the $\Omega(\log n)$ round barrier. Let us do the concrete calculation. In the binary tree, a parent must run 2 instances of the consensus protocol, one per child. Thus, in an ℓ-ary tree, the parent would need to run ℓ instances of the consensus protocol, one per child. Even if we assume a hypothetical asynchronous consensus protocol that terminates in an expected constant number of rounds, running ℓ such instances leads to an expected round complexity of $O(\log \ell) = \frac{1}{k} O(\log n)$ to terminate all instances. Going over all k levels, linearity of expectation gives a total round complexity of $k \cdot \frac{1}{k} O(\log n) = O(\log n)$, which is again logarithmic in n and we are back where we started.

We next outline our techniques, structured into those for the sharing phase and those for the consensus phase. Similar to prior work, our consensus phase also relies on our sharing primitive to generate the randomness required for asynchronous consensus. Unlike previous protocols, which implement sharing via n parallel AVSS instances (or some closely related procedure), incurring an inherent $\Omega(n^3)$ message cost, our approach avoids this overhead.

2.3 Our Ideas I: Committee-Based Sharing

As outlined above, a key problem is that the adversary can make a malicious branch succeed, which yields a final aggregate transcript with no honest contribution (thus, the adversary knows the secret key). We observe that the root cause for this is that each party generates its own input (i.e., the PVSS transcript) to the recursion protocol. Our idea is to start differently: We let other (randomly chosen) parties generate each party's input. Specifically, we want each party's input to be an aggregated PVSS transcript whose secret remains hidden from the adversary (we call such a transcript *good*). This way, even if a malicious branch in the recursion protocol succeeds, the final aggregate will also be good (and the secret key remains hidden from the adversary). Our sharing protocol achieves exactly that and is inspired by random-committee techniques, as used extensively in the literature on communication-efficient distributed protocols [ACG+25, KRY22]. Notably, our approach *inverts* this technique: Specifically, we associate with each party i a random hidden committee $\mathcal{L}_i$ of $O(\lambda)$ parties that "collectively generate" an aggregated PVSS transcript for party i. In more detail, this works as follows.

Description. Each party $i \in S$ locally generates a PVSS transcript Trx_i for a randomly sampled secret s_i, and erases s_i after that. It then sends Trx_i to a random set of $O(\lambda)$ other parties, where the set is derived using a verifiable random function (VRF) to ensure near-uniform selection. Once a party j has received "sufficiently many" (around $\frac{2}{3}$-fraction of the expected number) PVSS transcripts from other parties, it aggregates them into $\tilde{\mathsf{Trx}}_j$ and disperses the aggregate among all parties via a *provable AVID*[6] protocol. It then sends a signed proof π_j that it succeeded in doing so (i.e., proof of dispersal), and parties forward these proofs to make sure that all parties know of at least one successful dispersal. Such a proof constitutes a *validated input*. The idea is to generate succinct validated inputs for all parties that can be used for subsequent steps in the protocol (specifically, the consensus protocol).

Security. If S has an honest super-majority, then we can guarantee that any validated aggregated PVSS transcript $\tilde{\mathsf{Trx}}_j$ has a secret hidden from the adversary. For this, by a Chernoff bound, we know Trx_j contains a transcript from an honest party that erased its PVSS secret at the time of generation. Hence, the aggregate remains unpredictable even if that party is corrupted later. Further, again by a Chernoff bound, we can

[6] A provable asynchronous verifiable information dispersal protocol [LLTW20] enables a sender to disperse data among parties together with a succinct proof that enough correct fragments were delivered to allow later reconstruction.

guarantee *liveness*: There exists at least one forever-honest party that succeeds in forming such an aggregate. In particular, all parties eventually receive a validated input and terminate.

For the VRF (implementable using BLS signatures), we can rely on a plain PKI setup and *do not* need a trusted setup using common techniques [Ram23]. Specifically, we can use a random oracle H to derive a random seed $\mathsf{seed} \in \{0,1\}^\lambda$ for a list of registered public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ as $\mathsf{seed} := \mathsf{H}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. This seed is then used by parties as an additional input to the VRF evaluation algorithm. While the adversary can influence the seed, this advantage is only polynomial in λ for a computationally-bounded adversary (as it can only generate $\mathsf{poly}(\lambda)$ lists of public keys, or make only $\mathsf{poly}(\lambda)$ calls to the random oracle H). Thus, the overwhelming success probability $1 - \mathsf{negl}(\lambda)$ of our protocol absorbs the adversary's polynomial advantage.

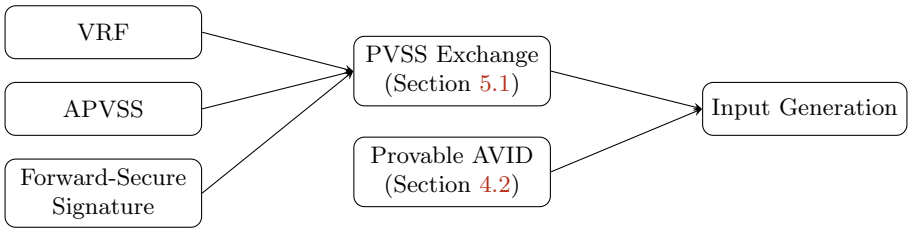


Figure 1: Overview of the sharing protocol.

2.4 Our Ideas II: Recursive Consensus

At a high level, our ADKG protocol proceeds in two steps. First, parties run the sharing protocol to generate validated inputs. Then, they invoke a validated asynchronous Byzantine agreement (VABA) protocol to agree on a single input. The respective aggregated PVSS transcript is retrieved and parties terminate.

2.4.1 VABA from Weak Coin

The VABA protocol we use is a variant of No-Waitin-Hotstuff [AJM+21], which we call *Succinct-Knowledge-NWH (SK-NWH)*. It leverages succinct proofs to reduce the communication cost to quadratic per view. Each view runs a proposal election protocol, which relies on a weak coin for liveness. Concretely, parties first commit their proposals via a provable AVID, and the weak coin then determines which proposal to output. Unlike previous protocols, our proposal election protocol also handles the case where the weak coin selects parties that did not commit any input. Notably, the core of our VABA protocol is a recursive protocol for the weak coin, which we explain next.

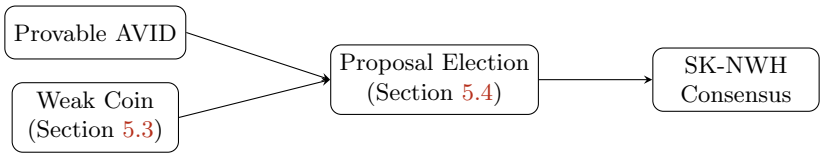


Figure 2: Overview of the SK-NWH protocol.

2.4.2 Weak Coin Generation

We now describe how our weak coin generation protocol works. For this, we partition the set of n parties recursively into $\ell = n^{1/k}$ smaller sets of roughly the same size at each step. This hierarchical partition forms a tree T of degree ℓ and depth $k \in [\log n]$.

Description. The protocol for S proceeds as follows. First, validated inputs for all parties are generated via our sharing protocol (see previous section). Next, SK-NWH is run on each child $k \in [\ell]$, outputting a

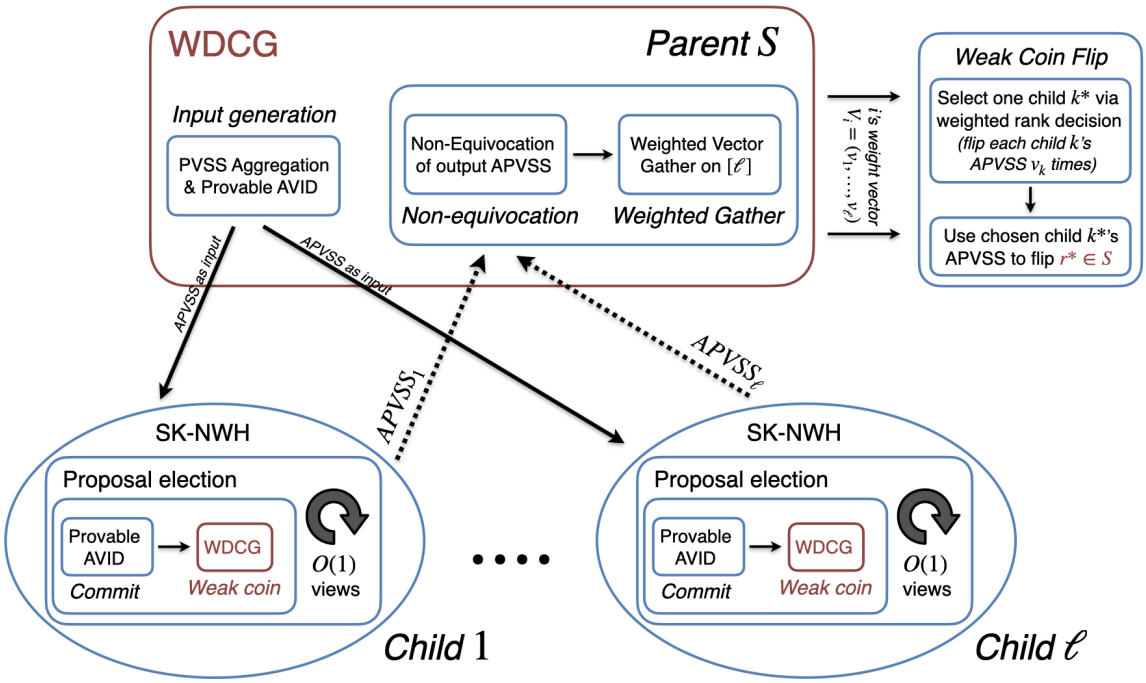


Figure 3: Our weak coin protocol for a set S. The protocol is split into the weak distributed coin generation (WDCG) protocol and the weak coin flip protocol. We present a recursive view for the WDGC protocol, where the top is a WDCG on S and the bottom WDCG are from the recursion on the children $1, \ldots, \ell$ of S.

provable aggregated PVSS transcript[7] π_k, where the randomness for this consensus is obtained recursively from that child's own children. Since a malicious child may generate many valid outputs from its SK-NWH, we run a *non-equivocation* protocol per child, guaranteeing at most one valid output per child. Consequently, there are at most $\ell = n^{1/k}$ possible aggregate PVSS transcripts to choose from. Since there is at least one good child (assuming S is good itself), every party will see at least one child's transcript that passed the non-equivocation phase. (Note that different parties may receive their transcripts from different children). Using that child's aggregate transcript, parties then run a *Weighted Gather* protocol.[8] This is similar to a regular Gather protocol [AJM+21], but instead of recording party identities, it records the number of parties supporting each child's transcript. This compresses a vector of length n into a "weight vector" of length $\ell = n^{1/k}$, with each weight at most $\log n$ bits. The resulting output vector $V_i = (v_1, \ldots, v_\ell)$ from each honest party $i \in S$ is guaranteed to be a superset of a large core vector V^* (i.e., $v_k \geq v_k^*$ for all $k \in [\ell]$) of total weight at least $n - f$. (We also have the guarantee of a cover vector with total weight at most n.) After outputting the weight vector, parties retrieve the aggregated PVSS transcripts Trx_k, $k \in [\ell]$.

This setup, which we call *Weak Distributed Coin Generation (WDCG)*, enables a *weak coin flip* protocol that, with constant probability, outputs a common uniformly random value $r^* \in S$ each time it is invoked.[9] This works as follows. From the WDCG setup, each party outputs a weight vector along with the aggregate transcripts. These transcripts are then used to derive "weight-many" ranks for each child. Specifically, the weight v_k of a child $k \in [\ell]$ determines how many ranks from its transcript are generated for it: its transcript Trx_k first provides a random value s_k, which is then used to derive v_k many ranks $r_k[i] := \mathsf{H}(i, (s_k, k))$ for $i \in [v_k]$, where H is a random oracle. Finally, the child $k^* \in [\ell]$ with the highest rank is selected, and its

[7] By this, we mean a succinct proof showing the existence of a good aggregated PVSS transcript Trx_k, dispersed among the parties in S and retrievable later if needed.

[8] The protocol also has the *binding cover* property [DDL+24].

[9] Thus, a single call to WDCG suffices to generate all the randomness required for a consensus protocol.

aggregate PVSS is used once more to flip a single coin $r^* \in S$, which is the protocol output. With constant probability, the highest rank corresponds to a child PVSS that all parties flip, since every coin derived from the core is flipped by all. In that case, all parties select the same transcript to generate the final coin r^* and thus agree on the output. Finally, to bootstrap the recursion (base case), parties call any constant-round VABA protocol with cubic communication [AJM+21, AJM+23].

Message blow-up within malicious child? While the above construction works, there is one notable concern that we have not yet addressed. Namely that a malicious child may cause its honest parties to send too many messages during its SK-NWH execution. We overcome this by simply limiting the number of views in each child SK-NWH execution to at most a value in $\Theta(\lambda)$. The probability of non-termination is then negligible.

2.4.3 Efficiency

We analyze the communication and round complexity of our protocol. We denote $O_\lambda := O(\mathsf{poly}(\lambda))$; in our protocols, this polynomial is at most quadratic. Let S be one of the $\ell = n^{1/k}$ children. The weak distributed coin generation protocol runs in $O_\lambda(|S|^2 n^{1/k})$ communication. The SK-NWH and the weak coin flip run in $O_\lambda(n^2)$ and $O_\lambda(n^{2+1/k})$ communication per view, respectively, and the weak coin flip has a constant success probability per view whenever S is a good child (i.e., less than a $\frac{1}{3}$-fraction of corrupt parties). After an expected constant number of rounds and $O_\lambda(|S|^2 n^{1/k})$ expected communication, the VABA protocol on S will complete with a valid aggregated PVSS transcript for its parent. Even when S is a bad child, we bound the number of views so that the total communication cost remains $O_\lambda(|S|^2 n^{1/k})$. Overall, each level requires, in expectation, $O(1)$ rounds and $O_\lambda(|S|^2 n^{1/k})$ communication. Since there are $O(k)$ levels, by linearity of expectation, the total number of expected rounds is $O(k)$ and the total expected communication is $O_\lambda(n^{2+1/k})$. For $k = \log n$, we obtain $O_\lambda(n^2)$ communication and $O(\log n)$ rounds. And for $k = \frac{1}{\varepsilon}$ with a fixed $\epsilon \in (0, 1]$, we obtain $O_\lambda(n^{2+\varepsilon})$ communication and $O(\frac{1}{\varepsilon})$ rounds. An interesting question is whether DKG protocols can achieve $O_\lambda(n^2)$ communication (including setup costs) in $o(\log n)$ rounds.

2.4.4 Recalling Section 2.2.1

We now briefly explain how our approach resolves the three issues that caused the synchronous recursive approach of [BLL+24] to fail in the asynchronous setting.

- *Malicious branch succeeds.* We address this problem by generating the input directly within the parent set, whereas in their protocol the input is provided by the children. Stated differently, their protocol adopts a purely bottom-up approach, whereas ours follows an up–bottom-up strategy.
- *No deterministic asynchronous consensus.* We address this problem by setting up the required randomness recursively from the children, whereas their protocol relies on a deterministic (synchronous) consensus protocol.
- *Breaking the log-round barrier.* We address this problem by employing a generalized recursion tree and by having parties wait only for the fastest child. Using the constant-round Weighted Gather, we can then amplify that single input into a common core (weight vector) among all parties and flip weak coins with constant success probability.

3 Preliminaries and Model

In this section, we define the model and preliminaries for our paper.

General notation. Let λ denote the security parameter. We assume that the public parameters of all underlying cryptographic primitives, which are implicitly parameterized by λ, are fixed in advance and known to all parties. For two integers $a \leq b$, we define the set $[a, b] := \{a, \ldots, b\}$; if $a = 1$, we write this set as $[b]$, and if $a = 0$, we write it as $[\![b]\!]$. We use $\mathbb{N}$ to denote the set of positive integers, and

$\mathbb{N}_0$ the set of non-negative integers. We use the following vector notations. Let $v = (v_1, \ldots, v_k)$ be a vector in $\mathbb{N}_0^k$. We let $v[i] := v_i$ for every $i \in [k]$. Furthermore, for $v, u \in \mathbb{N}_0^k$, we write $v \leq u$ if for all $i \in [k]$, $v_i \leq u_i$. Similarly, for $v_1, \ldots, v_m \in \mathbb{N}_0^k$, $\max\{v_1, \ldots, v_m\}$ is the vector u such that for all $i \in [k]$, $u[i] = \max\{v_j[i] \mid j \in [m]\}$. Finally, for $v \in \mathbb{N}_0^k$, we let $|v| := \sum_{i=1}^{k} v[i]$. We use standard cryptographic terminology such as negligible, overwhelming, and PPT (probabilistic polynomial-time). We measure the communication cost of our distributed protocols in bits, and assume that all stated protocol properties hold except with negligible probability in λ.

Adversarial and network model. We consider a complete network of n parties $\mathcal{P} := \{1, \ldots, n\}$ connected by pairwise authenticated channels. We consider an *asynchronous* network, where any message can be delayed arbitrarily under the constraint that messages sent between honest parties must eventually be delivered. We assume a Byzantine adversary who can corrupt up to $f < n/3$ parties and may cause them to deviate from the protocol arbitrarily. Furthermore, the adversary is *strongly adaptive*: it can corrupt a party at any time during the protocol execution, and delete or substitute any undelivered message that this party sent while being honest.

Public key infrastructure. As common in this line of work on distributed cryptographic protocols [DY81], we assume that parties have established a bulletin board *public key infrastructure* (PKI) before the protocol execution. Concretely, we assume that every party $i \in [n]$ has a public-secret key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for a digital signature scheme DS, for an aggregatable PVSS scheme PVSS, for a verifiable random function VRF, and for a forward-secure digital signature scheme KDS (see Appendix B for definitions). For improved readability, we assume that $(\mathsf{pk}_i, \mathsf{sk}_i)$ includes the keys of all these underlying primitives. In the bulletin board PKI model (also called *plain* PKI model), each party generates its keys locally (where corrupt parties may choose their keys arbitrarily) and then makes its public keys known to everybody using a public bulletin board. Throughout, we will use $\vec{\mathsf{pk}} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ to denote the list of all public keys.

3.1 Cryptographic Tools

We introduce the tools used in the paper (see Appendix B for formal definitions).

Verifiable random function. A verifiable random function (VRF) [MVR99] allows to produce a pseudorandom string u along with a proof of correctness π.In more detail, on input a message m and a secret key sk, a verifiable random function VRF produces a pseudorandom string $u = \mathsf{VRF}(\mathsf{sk}, m)$ along with a proof π. Using the proof π and the public key pk (corresponding to sk), anyone can verify that u was computed correctly without revealing any information about sk. Looking ahead, we will make use of VRFs for local committee-sampling, and we note that this use does not require a global source of randomness. Notably, we assume only a plain PKI setup and do not rely on any trusted VRF setup; the VRF can be instantiated using [DY05].

Forward-secure digital signature. In a forward-secure digital signature scheme KDS [BM99, KO17], each party i has a base public-secret key pair $(\bar{\mathsf{pk}}_i, \bar{\mathsf{sk}}_i)$, where the public key is fixed but the secret key can sequentially be updated (in our case, after the VRF evaluation to determine the set of elected parties). For this purpose, there exists an update function Update that on input a secret key $\bar{\mathsf{sk}}_i$ and an epoch number $r \in \mathbb{N}$ outputs the updated secret key $\bar{\mathsf{sk}}_i$ for the next epoch $r + 1$. Note that, by convention, the pair $(\bar{\mathsf{pk}}_i, \bar{\mathsf{sk}}_i)$ is included in $(\mathsf{pk}_i, \mathsf{sk}_i)$. Looking ahead, we will make use of forward-secure digital signatures for local committee-sampling; the scheme can be instantiated using [KO17].

Aggregatable publicly verifiable secret sharing. In a verifiable secret sharing (VSS) scheme [Fel87], a dealer distributes shares of a secret among a group of parties, ensuring the secret can only be reconstructed if a threshold number of these parties collaborate. Publicly verifiable secret sharing (PVSS) [Sta96] extends this concept by allowing any external entity to verify the correctness of the secret sharing via a single transcript. We focus on PVSS schemes that enable the aggregation of multiple secret sharings while maintaining public verifiability, also referred to as *aggregatable PVSS* (APVSS) schemes; the scheme can be instantiated using [GJM+21, BL23, BLL+24]. Furthermore, the works [GJM+21, BLL+24] demonstrate

how an aggregated PVSS transcript can be used to construct a one-round distributed randomness beacon protocol.

Non-interactive zero-knowledge proofs. In our protocols, we use non-interactive zero-knowledge (NIZK) proofs [BFM88]. Informally, a non-interactive proof system for an NP relation $\mathcal{R}$ with respect to a random oracle H is a pair of PPT algorithms $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVerify})$ such that (i) $\mathsf{PProve}^{\mathsf{H}}$ takes a statement x and a witness w with $(x, w) \in \mathcal{R}$ as input and outputs a proof π, and (ii) $\mathsf{PVerify}^{\mathsf{H}}$ takes the statement x and the proof π as input and outputs a decision bit 0 (accept) or 1 (accept). Regarding security properties, completeness requires that honestly computed proofs for $(x, w) \in \mathcal{R}$ are always accepted. Soundness requires that no malicious prover can find an accepting proof for a false statement x, i.e., a statement such that $(x, w) \notin \mathcal{R}$ for all w. Finally, zero-knowledge requires that there is a simulator that can simulate proofs without knowing the witness w by programming the random oracle H appropriately. For improved readability, we will omit the random oracle associated to PS in our descriptions hereafter. Furthermore, our NIZKs can be instantiated using succinct non-interactive arguments of knowledge (SNARKs) with $O(\lambda)$ proof size and transparent setup [Fu25, AGL+23].

4 Helper Primitives

In this section, we present the auxiliary primitives that serve as tools for our main building blocks.

4.1 Non-Equivocation Rounds

We construct two protocols for non-equivocation, AbandonableNonEquiv and NonEquiv.In both, every party has an input m_i and a *tag* that can be thought of as a round identifier. Every party can output a proof π, which signifies that this is the unique value that it input into the protocol. Parties can verify this proof using NEVerify in the case of the NonEquiv protocol and ANEVerify in the case of the AbandonableNonEquiv protocol. The AbandonableNonEquiv protocol has the additional property that parties can "abandon" the round and cause it to halt. By that we mean that after $f + 1$ honest parties abandon the protocol, no more values can be input into the protocol, meaning that no new values will receive valid proofs. Note that the NonEquiv protocol is similar to the provable broadcast primitive of [AMS19], and the AbandonableNonEquiv is similar to the locked broadcast primitive described in [AS22]. Looking ahead, we use the abandonability property in order to get the binding cover property for our weighted gather protocol (cf. Section 5.2).

4.1.1 Definition

We start by defining a non-equivocation protocol. In this protocol, each party i receives an input m_i and a tag tag, and may output a proof π_i.

Definition 1. *A non-equivocation protocol* $(\mathsf{NonEquiv}, \mathsf{NEVerify})$ *has the following properties, assuming all honest parties call the protocol with the same tag:*

- ***Validity.*** *If an honest party i outputs π_i, then* $\mathsf{NEVerify}(i, m_i, \pi_i, tag) = 1$.
- ***Non-Equivocation.*** *The adversary cannot generate two tuples (j, m, π) and (j, m', π') such that $m \neq m'$ and* $\mathsf{NEVerify}(j, m, \pi, tag) = \mathsf{NEVerify}(j, m', \pi', tag) = 1$.
- ***Liveness.*** *Every honest party eventually outputs a proof.*

An abandonable non-equivocation protocol is a non-equivocation protocol with the additional *abandonability* property, indicating that parties can abandon the protocol, while disallowing new outputs. Formally, we define this as follows.

Definition 2. *An abandonable non-equivocation protocol* $(\mathsf{AbandonableNonEquiv}, \mathsf{ANEVerify})$ *has the following properties, assuming all honest parties call the protocol with the same tag:*

- ***Validity.*** *If an honest party i outputs π_i, then* $\mathsf{ANEVerify}(i, m_i, \pi_i, tag) = 1$.
- ***Non-Equivocation.*** *The adversary cannot generate two tuples (j, m, π) and (j, m', π') such that $m \neq m'$ and* $\mathsf{ANEVerify}(j, m, \pi, tag) = \mathsf{ANEVerify}(j, m', \pi', tag) = 1$.
- ***Abandonability.*** *At the time $f+1$ honest parties abandon the protocol, the following holds: for every $i \in [n]$, there is a value m_i such that the adversary cannot generate a tuple (m'_i, π'_i) such that $m'_i \neq m_i$ and* $\mathsf{ANEVerify}(i, m'_i, \pi'_i, tag) = 1$.
- ***Liveness.*** *Either every honest party eventually outputs a proof, or some honest party abandons the protocol.*

4.1.2 Construction

The NonEquiv protocol has two logical rounds. In the first round, every party sends a "help" message with its input m_i to all parties. After receiving this message, parties reply with a signature on the first message received from any party in a "sig" message. Parties wait to receive $n - f$ such messages, and then output a proof that they have received this many signatures. Since any set of $n - f$ parties is a quorum, any two sets of $n - f$ parties will contain a common honest party that is only willing to sign one message per sender, guaranteeing non-equivocation.

The AbandonableNonEquiv protocol starts with the same two rounds, and then adds two identical rounds. Concretely, after receiving $n - f$ "sig" messages, parties send a "lock" message with their input m_i and the generated proof. Parties then reply with a "lock" message including signatures. Finally, parties output a proof that they have received enough "lock" messages. These additional rounds allow us to achieve the abandonability property by adding a barrier for entering the protocol. If $f + 1$ parties abandon the protocol, they will not send any new "lockSig" messages. This means that only values for which they have already received non-equivocation proofs might get enough "lockSig" messages, and parties cannot change their values if they already sent a non-equivocation proof.

For our constructions, we define the following non-interactive proof system. $\mathsf{PS}_{\mathsf{TS}} = (\mathsf{PProve}_{\mathsf{TS}}, \mathsf{PVerify}_{\mathsf{TS}})$ for the relation

$$\mathcal{R}_{\mathsf{TS}} := \left\{ \left(\vec{\mathsf{pk}}, m; S, (\sigma_i)_{i \in S} \right) \;\middle|\; |S| = n - f,\ \forall i \in S : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, m, \sigma_i) = 1 \right\}.$$

Essentially, this proof system is a threshold signature, proving knowledge of $n - f$ signatures on the same message m from different parties. We present the non-equivocation protocol as pseudocode in Algorithms 1 and 2, and the abandonable non-equivocation protocol in Algorithms 3 and 4.

4.1.3 Analysis

We defer the proofs to Appendix C.1.

Theorem 1. *The pair* (NonEquiv, NEVerify) *(cf. Algorithms 1 and 2)is a non-equivocation protocol resilient to $f < n/3$ Byzantine parties. The pair* (AbandonableNonEquiv, ANEVerify) *(cf. Algorithms 3 and 4) is an abandonable non-equivocation protocol resilient to $f < n/3$ Byzantine parties. Furthermore, both protocols run in $O((\lambda + m)n^2)$ communication and $O(1)$ rounds, where m denotes the size of the input.*

4.2 Provable AVID

A *Provable Asynchronous Verifiable Information Dispersal (AVID)* protocol consists of two sub-protocols Disperse and Retrieve along with a DisperseVerify function. Parties also have access to an external validity function valid. In the Disperse protocol, a designated sender holds as input an externally valid message m such that $\mathsf{valid}(m) = 1$. The sender may output a proof of dispersal (checked via the DisperseVerify algorithm), which indicates that the Retrieve protocol will succeed. In the Retrieve protocol, all parties can then jointly

reconstruct the committed message. In particular, parties should only output externally valid values from the retrieval. Note that provable AVID has totality for retrieve, but only provability for the dispersal.

Our primitive makes use of linear erasure codes [RS60] and cryptographic accumulators [BdM94], defined as follows (see Appendix B for formal definitions).

Linear erasure codes. We use standard (q, b)-Reed-Solomon codes [RS60]. These codes enable the encoding of b data symbols into a codeword of q symbols (using the Encode algorithm), such that any subset of b symbols from the codeword is sufficient to recover the original data (using the Decode algorithm). We will use Reed-Solomon codes with parameters $(q, b) = (n, f + 1)$. This choice ensures robust decoding while preserving security against Byzantine parties.

Cryptographic accumulator. A cryptographic accumulator [BdM94] is a primitive that allows elements from a set D to be "accumulated" into a single accumulation value z, using the $\mathsf{Eval}_{\mathsf{Acc}}$ algorithm. Further, for any element in D, it provides a compact proof of membership, called a witness, which can be generated using the $\mathsf{Witness}_{\mathsf{Acc}}$ algorithm and then verified using the $\mathsf{Verify}_{\mathsf{Acc}}$ algorithm. The standard security notion for accumulators is *collision-resistance*, which ensures that an adversary cannot create invalid membership proofs. We use accumulators with proof and accumulation value sizes of $O(\lambda)$ [BBF19].

4.2.1 Definition

Formally, we define a provable AVID protocol as follows.

Definition 3. *In a provable AVID protocol* $(\mathsf{Disperse}, \mathsf{DisperseVerify}, \mathsf{Retrieve})$*, a designated sender s has an input m, and all parties have access to an external validity function* valid*. Assuming that* $\mathsf{valid}(m) = 1$*, the protocol has the following properties:*

- ***Binding.*** *At the time the first honest party completes the* $\mathsf{Disperse}$ *protocol, there exists a unique committed value m'.*
- ***External Validity.*** *For the m' defined in the binding property,* $\mathsf{valid}(m') = 1$*. Further, if the sender is honest, then the committed value is its input $m' = m$.*
- ***Agreement.*** *Any two honest parties that output a value from the* $\mathsf{Retrieve}$ *protocol, output the same value m' (defined in the binding property above).*
- ***Termination.*** *If s is honest, then it completes the* $\mathsf{Disperse}$ *protocol and outputs a proof π.*
- ***Provability.*** *If s is honest and it outputs a proof π, then* $\mathsf{DisperseVerify}(s, \pi) = 1$*.*
- ***Totality.*** *If an honest party receives a proof π such that* $\mathsf{DisperseVerify}(s, \pi) = 1$ *and all honest parties call* $\mathsf{Retrieve}$*, then they all complete* $\mathsf{Retrieve}$*.*

4.2.2 Construction

For our protocol, we assume an external validity function valid that parties have access to. Further, we make use of the following non-interactive proof system. $\mathsf{PS}_{\mathsf{acc}} = (\mathsf{PProve}_{\mathsf{acc}}, \mathsf{PVerify}_{\mathsf{acc}})$ for the relation

$$\mathcal{R}_{\mathsf{acc}} := \left\{ \left(\vec{\mathsf{pk}}, \mathsf{acc}, j; m, \pi_{\mathsf{TS}}\right) \;\middle|\; \begin{array}{l} \mathsf{acc} = \mathsf{Eval}_{\mathsf{Acc}}(ak, \mathsf{Encode}(m; n, f+1)), \\ \mathsf{valid}(m) = 1,\ \mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); \pi_{\mathsf{TS}}) = 1 \end{array} \right\}.$$

This proof system is essentially just a threshold signature proving knowledge of $n - f$ signatures on an accumulation value acc for a message m with proof of external validity for m.

We describe our provable AVID protocol, which uses linear erasure codes and cryptographic accumulators. We begin with the dispersal phase, where a designated sender $s \in [n]$ holds an input message m that it wants to disperse among all parties. Then, we proceed with the retrieval phase, where all parties jointly reconstruct the message m. In the dispersal phase, the sender first encodes its message m into n code words $M' := (m'_1, \ldots, m'_n)$ via an $(n, f + 1)$-encoding function Encode. It then accumulates these code words

M' into an accumulation value acc. Next, it collects $n - f$ signatures on acc from other parties to form a non-equivocation proof π_{TS} for acc. Additionally, it produces a proof π_{acc} showing that acc corresponds to a message m such that $\mathsf{valid}(m) = 1$ (i.e., a proof that acc is the accumulation value for an encoding of m and m is externally valid). In the next phase, it distributes shares of m with validity proofs by sending $(\mathsf{acc}, \pi_{acc}, m'_i, w_i)$ to each party i, where w_i is a membership proof of m'_i with respect to acc. Upon receiving such a valid tuple (which also guarantees uniqueness of acc and external validity of the encoded m), other parties reply with signatures on the senders index s. With that, the sender generates a final proof of dispersal π_{tot} with $n - f$ of these signatures and terminates with output π_{tot}. The function $\mathsf{DisperseVerify}$ is then just verifying whether π_{tot} is a verifying threshold signature of weight $n - f$. For the retrieval phase, each party simply sends its received share along with membership proof (m'_i, w_i) to all parties. Upon receiving $f + 1$ such valid shares for the same acc with proof π_{tot}, a party can then reconstruct the message m via the decoding function Decode of the underlying erasure code. We present it as pseudocode in Algorithms 5 to 7.

4.2.3 Analysis

We defer the proofs to Appendix C.2.

Theorem 2. *The pair* $(\mathsf{Disperse}, \mathsf{DisperseVerify}, \mathsf{Retrieve})$ *(cf. Algorithms 5 to 7)is a provable AVID protocol resilient to* $f < n/3$ *Byzantine parties. The* $\mathsf{Disperse}$ *protocol runs in* $O(\lambda n + m)$ *communication and* $O(1)$ *rounds, where* m *denotes the size of the input. And the* $\mathsf{Retrieve}$ *protocol runs in* $O(\lambda n^2 + mn)$ *communication and* $O(1)$ *rounds.*

5 Building Blocks

In this section, we describe the main building blocks for our DKG protocol.

5.1 PVSS Exchange

In this section, we construct a PVSS exchange protocol. The goal of the protocol is to output an aggregated PVSS transcript that can be proven to have at least one honest contribution with high probability.

Each party generates a PVSS transcript and sends it to a $\Theta(\lambda)$ number of parties. Parties then output an aggregated transcript if they get a "large enough" number of contributions to know that at least one honest transcript is included with high probability.

5.1.1 Definition

Parties call a PVSS exchange protocol with a *tag* as input, and may eventually output an aggregated transcript and a proof. Parties can also run the $\mathsf{ExchangeVerify}$ algorithm in order to verify that an output includes at least one honestly generated transcript with high probability.

Definition 4. *A PVSS exchange protocol* $(\mathsf{PVSSExchange}, \mathsf{ExchangeVerify})$ *has the following properties if all honest parties call the* $\mathsf{PVSSExchange}(tag)$ *protocol:*

- ***Verifiability.*** *If an honest party outputs* $(\mathsf{trans}_i, \pi_i)$*, then* $\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_i, tag) = 1$*. Furthermore, if the adversary provides a pair* (trans, π) *such that* $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi, tag) = 1$*, then* trans *includes at least one contribution generated by an honest party that erased its contents without being corrupted.*
- ***Termination.*** *At least one forever-honest party completes the protocol and outputs a value.*

5.1.2 Construction

In this construction, we use a VRF that receives a key, an index, and a tag, and outputs a boolean flag and a proof. The VRF outputs *true* with probability $\frac{\lambda c}{n}$ for some chosen constant $c \geq 10$ and $false$ with the remaining probability. In addition, parties wait to hear from "around $\frac{2}{3}$" of the expected number of messages, with a slack of ϵ, meaning that parties wait for $\frac{2}{3} - \epsilon$ of the expected number of messages. The parameter ϵ can be chosen to be any agreed-upon number strictly between 0 and $\frac{1}{50}$. After hearing from $(\frac{2}{3} - \epsilon)\lambda c$ parties with correct VRF proofs, parties aggregate the received transcripts, and output the aggregated transcript and a proof that it contains enough transcripts. Intuitively, in order to show that the output is secure, we show any set of "enough" random parties should contain an honest party with high probability. In order to prove that at least one forever-honest party terminates we show that for any given set of corrupted parties, there is an extremely low probability that no honest party terminates. In fact, the probability is so low that even considering all possible choices of corrupted parties, there is still a high probability of at least one forever-honest party terminating.

Further, we make use of the following non-interactive proof system $\mathsf{PS}_{\mathsf{exch}} = (\mathsf{PProve}_{\mathsf{exch}}, \mathsf{PVerify}_{\mathsf{exch}})$ for the relation

$$\mathcal{R}_{\mathsf{exch}} := \left\{ \left(\vec{\mathsf{pk}}, \mathsf{trans}, j, tag; S, \{(\sigma_i, \mathsf{trans}_i, \pi_i)\}_{i \in S}\right) \;\middle|\; \begin{array}{l} |S| = (\frac{2}{3} - \epsilon)\lambda c, \ \forall i \in S : \\ \mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}_i, \vec{\mathsf{pk}}) = 1, \\ \mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}_i, true, j, tag, \pi_i) = 1, \\ \mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}_i, (j, \mathsf{trans}_i, tag), \sigma_i) = 1 \end{array} \right\}.$$

Essentially, the proof system shows that a party received $(\frac{2}{3} - \epsilon)\lambda c$ valid PVSS transcripts from other parties with corresponding valid signatures along with verifying VRF proofs. We present the protocol as pseudocode in Algorithms 8 and 9.

5.1.3 Analysis

We defer the proofs to Appendix D.1.

Theorem 3. *The pair* $(\mathsf{PVSSExchange}, \mathsf{ExchangeVerify})$ *(cf. Algorithms 8 and 9) is a PVSS Exchange protocol resilient to* $f < n/3$ *Byzantine parties. Furthermore, it runs in* $O(\lambda^2 n^2)$ *communication and* $O(1)$ *rounds.*

5.1.4 On the VRF Setup

One common way to perform a VRF setup in a distributed setting with a random oracle H is to allow every party to choose a public key pair for a signature scheme with unique signatures and posting it in a bulletin board setup, and only then calling a randomness beacon to get a common random seed $\mathsf{seed} \in \{0,1\}^\lambda$. In order to compute the function VRF on a given value v, parties sign v to produce the unique signature σ, and then output the value $r := \mathsf{H}(\mathsf{seed}, \sigma)$ and the proof σ. Parties can check the correctness of the output by simply verifying the signature and checking that r is indeed computed correctly.

To remove the call to a randomness beacon, thus making the setup simply a bulletin board PKI setup, we can employ the techniques of [Ram23]. Instead of computing the seed seed as the output of a randomness beacon, we set $\mathsf{seed} := \mathsf{H}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. This allows the adversary to influence the seed seed by picking its public keys. However, a computationally bounded adversary can only generate $\mathsf{poly}(\lambda)$ sets of public keys (or make $\mathsf{poly}(\lambda)$ calls to the random oracle H). Note that the probability of failure in Lemmas 1 and 2 is negligible in λ, and thus from the union bound, the probability of the adversary being able to generate a set of public keys for which the claims do not hold is at most $\mathsf{poly}(\lambda) \cdot \mathsf{negl}(\lambda)$, which is still negligible in λ.

5.2 Weighted Gather

The *Weighted Gather* protocol is a modification of the Verifiable Gather protocol [AJM+21].

In a regular gather protocol, the input is an element in $[n]$ and the output is a binary vector of length n. The key property is that there is a core vector c with $n - f$ elements such that all output vectors contain the vector c. In weighted gather, the input is an element in $[\ell]$ with $\ell < n$ and the output is a *weighted* vector of length ℓ. The key property is that there is a core vector c whose sum of weights is $n - f$ such that all output vectors dominate the vector c in each coordinate.

For the purposes of this work, the weighted gather protocol needs to be a cover gather [DDL+24], and does not need to be verifiable. A "regular" gather protocol only guarantees that there exists a large *common core set* such that all parties' outputs contain it. A cover gather also guarantees that there exists a *common cover set* such that all parties' outputs are included in it. In order to achieve this additional property, we use an abandonable non-equivocation round, that allows parties to block new inputs from being considered before terminating. Furthermore, for this work, the protocol does not even require a validity property other than external validity.

Looking ahead, the verifiability of the weighted gather will be implicitly checked via a quorum of parties in the Proposal Election protocol.

5.2.1 Definition

Every party i enters the protocol with a pair of inputs (x_i, π_i) such that $x_i \in [\ell]$ for some commonly-known ℓ. Every party outputs a vector $v_i \in \mathbb{N}_0^\ell$. Parties also have access to an external validity function valid that receives a pair (x_i, π_i) and outputs either 1, indicating that the pair is valid, or 0, indicating that it is not. Every honest party is assumed to have a valid input pair (x_i, π_i). In the definition, recall that the *weight of a vector*, $|v|$, is the sum of all the entries in v, and that $u \geq v$ indicates that u is greater than or equal to v in each entry.

Definition 5. *A Weighted Gather protocol has the following properties, assuming all honest parties participate with valid inputs:*

- ***Binding Core.*** *At the time the first honest party completes the protocol, there exists a vector $v \in \mathbb{N}_0^\ell$ such that $|v| \geq n - f$ and $v_i \geq v$ for the output v_i of any honest i.*
- ***Binding Cover.*** *At the time the first honest party completes the protocol, there exists a vector $u \in \mathbb{N}_0^\ell$ such that $|u| \leq n$ and $u \geq v_i$ for the output v_i of any honest i.*
- ***External Validity*** *For every $j \in [\ell]$ such that $u[j] > 0$, it is possible to extract a proof π_j such that $\mathsf{valid}(j, \pi_j) = 1$ at the time the first honest party completes the protocol.*
- ***Termination.*** *All parties output a value and complete the protocol.*

5.2.2 Construction

The weighted gather protocol proceeds in several near-identical logical rounds. In the first round, parties call an abandonable non-equivocation protocol on their inputs, which are indices in $[\ell]$. Following that, they send their input in a "val" message, along with the proof of non-equivocation. After receiving $n - f$ such messages, parties construct a vector vec, which simply counts the number of times each index was received. In the second logical round, parties call a non-equivocation protocol on their vectors and forward them in a "vec" message along with proofs that they were correctly constructed. After Receiving $n - f$ vectors, parties aggregate them to a vector agg by computing the maximum of the vectors in each entry (logically, this is similar to a relaxed union of the original sets). Again, parties call a non-equivocation protocol on the aggregated vectors and send them in an "agg" message along with proofs of correct computation. In the final round, parties wait until they receive $n - f$ aggregated vectors and compute an output from them. They then send that output to all parties in an "out" message along with a proof, and abandon the abandonable non-equivocation protocol. Every party that receives a correct output adopts that value, forwards it and abandons the abandonable non-equivocation protocol. Finally, after receiving $n - f$ "out" messages, parties terminate.

Having honest parties send their inputs makes sure that every party eventually receives $n-f$ messages and constructs a vector. Following a standard counting argument, as shown in Lemma 3, in any set of $n-f$ correctly aggregated agg values at least $f+1$ include the same vec from the previous round. This vec will then be included in every output, and thus can act as a common core. On the other hand, since parties only terminate after receiving $n-f$ "out" messages, they know that at least $f+1$ parties abandoned the abandonable non-equivocation protocol. This means that at that point in time, no new values can be introduced into the protocol, and thus the values that have already been input will form a cover for the outputs of the protocol.

We make use of the following non-interactive proof systems. First, $\mathsf{PS}_{\mathsf{cnt}} = (\mathsf{PProve}_{\mathsf{cnt}}, \mathsf{PVerify}_{\mathsf{cnt}})$ for the relation

$$\mathcal{R}_{\mathsf{cnt}} := \left\{ \begin{array}{r} (\vec{\mathsf{pk}}, \mathsf{vec}, (m_i)_{i\in[\ell]}; \\ (S_i)_{i\in[\ell]}, \{(\sigma_{i,j}, \pi_{i,j})\}_{i\in[\ell], j\in S_i} \end{array} \;\middle|\; \begin{array}{l} |\mathsf{vec}| = n-f,\ \forall i \in [\ell] : |S_i| = \mathsf{vec}[i], \\ \forall i \in [\ell], j \in S_i : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, m_i, \sigma_{i,j}) = 1, \\ \forall i \in [\ell], j \in S_i : \mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, m_i; \pi_{i,j}) = 1 \end{array} \right\}.$$

This proof system essentially counts how often each message m_i from a known set of messages $\{m_i\}_{i\in[\ell]}$ was received by other parties. For all such messages, this count number is then stored in a vector vec of length ℓ, where $\mathsf{vec}[i]$ tells how often message m_i was received. Roughly, this can be thought of as a generalized threshold signature for a list of messages with distinct weights.

Second, a proof system, which takes the maximum from a set of received vectors. Concretely, we define the proof system $\mathsf{PS}_{\mathsf{max}} = (\mathsf{PProve}_{\mathsf{max}}, \mathsf{PVerify}_{\mathsf{max}})$ for the relation

$$\mathcal{R}_{\mathsf{max}} := \left\{ \begin{array}{r} (\vec{\mathsf{pk}}, \mathsf{agg}, (m_i)_{i\in[\ell]}; \\ S, \{(\mathsf{vec}_i, \pi_{\mathsf{cnt},i})\}_{i\in S} \end{array} \;\middle|\; \begin{array}{l} |S| = n-f,\ \mathsf{agg} = \max\{\mathsf{vec}_i \mid i \in S\}, \\ \forall i \in S : \mathsf{PVerify}_{\mathsf{cnt}}(\vec{\mathsf{pk}}, \mathsf{vec}_i, (m_i)_{i\in[\ell]}; \pi_{\mathsf{cnt},i}) = 1 \end{array} \right\}.$$

We present the protocol as pseudocode in Algorithm 10.

5.2.3 Analysis

We defer the proofs to Appendix D.2.

Theorem 4. *The protocol* WeightedGather *(cf. Algorithm 10) is a Weighted Gather protocol resilient to $f < n/3$ Byzantine parties. Furthermore, it runs in $O((\lambda + \ell \log n)n^2)$ communication and $O(1)$ rounds.*[10]

5.3 Weak Coin

To construct a consensus protocol, we require a Proposal Election protocol. As a core building block for the Proposal Election protocol, we construct a Weak Coin protocol that allows it to randomly sample an index $i^* \in [n]$. The coin is weak in the sense that parties might not agree on the output. However, with constant probability, all parties output the same uniformly sampled index. The weak coin construction is split into two parts: a setup phase called Weak Distributed Coin Generation (WDCG), and a protocol for flipping the coin (Flip). The WDCG protocol runs once to set up polynomially many coin flips. Parties may only call the Flip protocol after completing the WDCG protocol, using the WDCG protocol's output and state internally. Each call to Flip also requires a unique tag to differentiate different coin flips. It should be noted that our entire call stack is recursive, keeping track of a pair of indices $i_{\min}, i_{\max}$ indicating that only parties $\{i_{\min}, i_{\min}+1, \ldots, i_{\max}\}$ are running the protocol. These inputs are implicit in all protocols except the WDCG protocol, as this is the only part of the stack that makes recursive calls and needs to use these indices explicitly.

All parties enter the WDCG protocol with no input (other than the indices $i_{\min}, i_{\max}$) and output a value v_i. Parties enter the Flip protocol with the same value v_i and a *tag*, and output an index $i^* \in [n]$.

[10] We note that $\log n < \lambda$. Otherwise, the adversary would have exponential computational power. Therefore, the communication cost is upper bounded by $O(\lambda \ell n^2)$.

5.3.1 Definition

We formally define a weak coin protocol next.

Definition 6. *A Weak Coin protocol* (WDCG, Flip) *has the following properties, assuming all honest parties call* Flip(v_i, tag) *with* v_i *being their output from* WDCG*:*

- α***-Uniformity.*** *Each time* Flip *is called with a given tag, there is an event* E *that occurs with probability* α *or greater such that given* E*, all honest parties output the same uniformly sampled* i^* *from* Flip*. Furthermore, if* E *takes place, the adversary's view is independent of* i^* *before some honest party calls* Flip *with tag.*
- ***Termination.*** *If all honest parties participate in the* WDCG *protocol, they complete it. If all honest parties participate in the* Flip *protocol with the same tag, they all complete it.*

5.3.2 Construction

We construct a weak coin protocol next.

Weak distributed coin generation protocol. The WDCG protocol is a recursive protocol for setting up a weak coin. For a known parameter ℓ, the participating parties are recursively split into ℓ nearly equal sets in each step of the recursion.

Informally, the protocol proceeds in three logical steps:

1. Input generation: Parties start by attempting to get an aggregate transcript by calling the PVSSExchange protocol. Any party that manages to do so disperses the transcript using a provable AVID protocol, and then sends a proof that it succeeded in doing so. Parties forward these proofs in order to make sure that all parties know of at least one successful dispersal. Note that different parties may have different proofs for different provable aggregate PVSS transcripts.

2. Reducing the number of provable aggregate PVSS transcripts to a small set of at most ℓ options:

   In the base of the recursion (i.e., when there are very few parties), parties simply call a cubic consensus protocol, BaseConsensus so trivially there is just one transcript.

   This cannot be done when there are many parties, so parties instead aim to have at most one transcript per child. This is done by doing the following three step protocol for each child:

   (a) Recursively call WDCG for this child.

   (b) Call a consensus protocol RecConsensus for this child, using the above WDCG for randomness. Parties get their inputs for the RecConsensus protocol from the input generation phase described above.

   (c) Run a "child non-equivocation" procedure on the outcome of the consensus protocol above. This is done since some of the children might have a dishonest majority.

3. Weighted gather: After receiving a proof of non-equivocation from at least one child, parties forward that proof, and call a weighted gather protocol with that child's index as an input. Finally, parties output the vector that they received in the weighted gather protocol. In the background, parties also retrieve any transcript that was agreed upon by any of the children, after seeing a non-equivocation proof. This means that they only need to retrieve $O(\ell)$ many transcripts.

We present the protocol as pseudocode in Algorithm 11.

Technical notes. In the construction, all of the code described in ChildNonEquiv is run in the background and is not a separate building block that is called by the WDCG protocol. This sub-protocol is a slight adaptation of a non-equivocation protocol, allowing each child node to provide only one input to the next

stages. Note that since this is a specialized use case, the properties of this sub-protocol are proven directly as part of the coin flip protocol.

The partition algorithm receives three arguments $i_{\min}, i_{\max}$ and ℓ. The algorithm then outputs an array child that partitions all parties with indices between $i_{\min}$ and $i_{\max}$ to ℓ nearly equal parts. In particular, the first $(i_{\max} - i_{\min}) \mod \ell$ parts are of size $\lceil (i_{\max} - i_{\min})/\ell \rceil$ and the rest are of size $\lfloor (i_{\max} - i_{\min})/\ell \rfloor$. All other entries are $\perp$. If $\lfloor (i_{\max} - i_{\min})/\ell \rfloor < 4$, all entries between $i_{\max}$ and $i_{\min}$ are 1 instead.

We call two validated consensus sub-protocols (a base case protocol and a protocol for the recursion) in this construction. Both constructions need to be secure against a strongly adaptive adversary controlling less than $\frac{1}{3}$ of the total parties. In addition to the regular properties of consensus, requiring all honest parties to output the same externally valid value, we require parties to be able to output a proof that the output is indeed correct. We note that in any protocol in which parties generate commit certificates, the certificates can act as such a proof. See Definition 16 for a formal definition of the protocol.

The first of the consensus protocols, BaseConsensus, is called in the base case of the recursion. For this protocol, we use the No-Waitin' Hotstuff (NWH) protocol of [AJM+21, AJM+23]. The protocol has $O(\lambda n^3)$ expected bit complexity and $O(1)$ expected round complexity. The second consensus protocol, RecConsensus is called in every other level of the recursion. The protocol is a slightly adapted version of the NWH protocol, with two changes from the original. Instead of using the PE protocol constructed in [AJM+21, AJM+23], we use the PE protocol constructed in Section 5.4. Secondly, instead of sending vectors of $n - f$ signatures in each round, we send proofs that $n - f$ such signatures have been collected, using $\mathsf{PProve}_{\mathsf{TS}}$ and $\mathsf{PVerify}_{\mathsf{TS}}$ (as defined in Section 4.1). This reduces the costs of every part of the protocol other than the PE to only expected $O(\lambda n^2)$ per view.

To limit communication cost in the case of a dishonest majority, we only run each call to RecConsensus or BaseConsensus for $(\lambda c \cdot \log^{-1}(\frac{9}{7})) < 3\lambda c$ views. Note that this introduces a negligible probability of non-termination. In a call to RecConsensus or to BaseConsensus with less than a $\frac{1}{3}$-fraction of corrupt parties and a PE protocol with α-Binding, there is at most a $(1-\alpha)^r$ probability of not terminating by round r. Using $\alpha = \frac{2}{9}$ (BaseConsensus has $\frac{1}{3}$-Binding) and $r = \lambda c \cdot \log^{-1}(\frac{9}{7})$, we get a failure probability of at most

$$(1-\alpha)^r = (7/9)^{\lambda c \cdot \log^{-1}(\frac{9}{7})} = 2^{-\lambda c} \in \mathsf{negl}(\lambda),$$

in which case even the calls to the consensus protocols with an honest super-majority might not terminate.

Note that in the No-Waitin' Hotstuff protocol parties only terminate after receiving a commit certificate, which means that these protocols do indeed have verifiable outputs. Syntactically, we call the BaseConsensus and RecConsensus protocols with three arguments: $x, i_{\min}, i_{\max}$. Here, x is the input to the protocol, and $i_{\min}, i_{\max}$ signify that parties $i_{\min}, \ldots, i_{\max}$ are participating in the protocol. The Verify algorithm takes an output y, a proof π and two indices $i_{\min}, i_{\max}$, and checks whether y was the output from the call with parties $i_{\min}, \ldots, i_{\max}$.

Randomness beacon from APVSS. The works [GJM+21, BLL+24] demonstrate how to use an aggregated PVSS transcript to derive a one-round distributed randomness beacon protocol. To this end, each party i derives a partial coin on a given message using its secret key sk_i via the PartialCoin algorithm, and the result is verified using the VerifyPartial algorithm with input pk_i. Any set of $n - f$ valid partial coins can then be combined into a coin value using the AggregateCoin algorithm. We make use of this procedure in our flip protocol, which we describe next.

Flip protocol. The Flip protocol is called after running the WDCG protocol. The WDCG protocol's output and state are used in the Flip protocol. In the Flip protocol, parties use the transcripts during the WDCG protocol, along with the vectors they got from the weighted gather protocol to generate a random coin. They do this by using each transcript to flip a single random value, for a total of $O(\ell)$ flips. They then extract many pieces of randomness from that random value using a random oracle. Every honest party i extracts $v_i[j]$ pieces of randomness from the j'th coin, where v_i is its output from the weighted gather protocol. These pieces of randomness are used as *ranks* for the transcripts, and parties choose the transcript with the single highest associated rank. Finally, parties extract one more piece of randomness from the winning transcript and use it as a common coin. Since the weighted gather has a large binding core, there

is fairly high probability ($\frac{2}{3}$) that the top rank is one that all parties will see and thus they will output the same value. In addition, since the weighted gather is a cover gather, the adversary cannot wait to see which transcript will win and then make slow parties include it in their gather vectors retroactively. We present the protocol as pseudocode in Algorithm 15.

5.3.3 Analysis

We defer the proofs to Appendix D.3.

Theorem 5. *The pair* (WDCG, Flip) *(cf. Algorithms 11 and 15) is a Weak Coin protocol with* $\alpha = \frac{2}{3}$ *and resilient to* $f < n/3$ *Byzantine parties. The* WDCG *protocol runs in* $O(\ell\lambda^2 n^2)$ *communication and* $O(\log_\ell n)$ *rounds. And the* Flip *protocol runs in* $O(\ell\lambda n^2)$ *communication and* $O(1)$ *rounds.*

5.4 Proposal Election from Weak Coin

5.4.1 Definition

In a proposal election protocol, every honest party has an input prop_i and a *tag* signifying the PE instance. Parties have access to an external validity function valid, that can either output 1 on a given value, indicating that it is valid, or 0, indicating that it is not. We assume all honest parties start the protocol with valid inputs prop_i. Every honest party eventually outputs a value out_i and a proof π_i. Parties can verify the correctness of outputs and proofs with the PEVerify algorithm by computing $\mathsf{PEVerify}(\mathsf{out}, \pi, tag)$.

Definition 7. *A Proposal Election protocol has the following properties, assuming that honest parties call* $\mathsf{PE}(\mathsf{prop}, tag)$ *on externally valid values* prop*:*

- α***-Binding.*** *With probability at least* α*, at the time the first honest party completes the* PE *protocol, there exists some party* i^* *that was honest when starting the protocol such that the adversary cannot generate any pair of values* out, π *for which* $\mathsf{PEVerify}(\mathsf{out}, \pi, tag) = 1$ *and* $\mathsf{out} \neq \mathsf{prop}_{i^*}$*.*
- ***Completeness.*** *If an honest party outputs* out, π*, then* $\mathsf{PEVerify}(\mathsf{out}, \pi, tag) = 1$*.*
- ***External Validity.*** *If* $\mathsf{PEVerify}(\mathsf{out}, \pi, tag) = 1$*, then* $\mathsf{valid}(\mathsf{out}) = 1$ *or* $\mathsf{out} = \perp$*.*
- ***Termination.*** *If all honest parties start the* PE *protocol, they complete it.*

Note that combining α-Binding and completeness, we immediately find that with probability α or greater, all honest parties output the same value out. Further, since PEVerify is an algorithm (not a protocol with a state) in this construction, all honest parties output the same value for the same inputs. This slightly changes the properties of this PE protocol when compared to that of [AJM+21]. However, simply mapping the PEVerify protocol outputting 1 to it terminating and 0 to it not terminating (for example, by constructing a protocol that wraps PEVerify and does this exactly) results in the exact same properties as those of [AJM+21].

5.4.2 Construction

We construct a proposal election protocol from any weak coin protocol. Intuitively, parties would like to provide their proposals and then use the weak common coin to elect one of those proposals. Unfortunately, since parties can't wait to hear all proposals, they have to proceed after hearing at most $n - f$. This introduces four challenges:

1. First, parties might elect an index that hasn't provided a proposal. To solve this, parties tell each other if they've heard a proposal from the elected party. Any party that hears that $n - f$ parties haven't received a proposal can output $\perp$.
2. Second, parties might elect different indices with different proposals. Any party that hears two different proposals can also output $\perp$.

3. Third, in order to make sure that there is a "high enough" probability of success, parties output the elected proposal even if they hear it from only one of the $n - f$ parties. This is why parties make sure that a quorum of parties stored their proposal before proceeding to the coin flip.

4. Fourth, an adaptive adversary can wait to see who wins the election and corrupt it retroactively to equivocate its proposal. To overcome this, parties call a non-equivocation protocol before storing it in a quorum.

In more detail, after setting up the coin, parties call a non-equivocation protocol on their proposals prop, and send them with the corresponding proof in a "proposal" message. After receiving such a proof, parties store the proposal and non-equivocation proof and respond with a "stored" message. Every party waits to hear that $n - f$ parties stored its message and then sends a "ready" message indicating that it is ready to proceed to the coin flip. After receiving $n - f$ such "ready" messages, parties call the coin flip protocol and send "elected" messages informing each other of the random index r they output and of any stored value for party r. Parties then wait to receive "elected" messages from $n - f$ parties before computing their output. In the good case that all honest parties output the same index r of an honest party that sent a "ready" message, they will all send the same r in their "ready" messages. Moreover, each party will hear r's proposal from at least one party in the quorum. In addition, the adversary will not be able to report a different proposal, because it would have to produce a differing non-equivocation proof. Parties can then output that proposal, along with a proof that it was computed correctly. In any other case, the protocol provides liveness and parties can output $\perp$ and prove that there was either disagreement on the elected party r, or that many parties did not store its proposal.

We make use of the following non-interactive proof system, which essentially shows that the same value r was received at least $f + 1$ times from a total set of $n - f$ values from other parties. Concretely, we define the proof system $\mathsf{PS}_{\mathsf{elect}} = (\mathsf{PProve}_{\mathsf{elect}}, \mathsf{PVerify}_{\mathsf{elect}})$ for the relation

$$\mathcal{R}_{\mathsf{elect}} := \left\{ \left(\vec{\mathsf{pk}}, (r, tag); J, S, \{(r_i, m_i, \sigma_i)\}_{i \in S}\right) \;\middle|\; \begin{array}{l} |S| = n - f,\ |J| = f + 1,\ \forall j \in J : r_j = r, \\ \forall i \in S : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (r_i, m_i, tag), \sigma_i) = 1 \end{array} \right\}.$$

We also make use of a non-interactive proof system, which essentially shows that there was a unique winner, but it did not provide a correct output from the protocol. This means that enough parties either did not store its value (reported $\perp$), stored an invalid value, or did not store a correct proof of non-equivocation. Concretely, we define the proof system $\mathsf{PS}_{\mathsf{bad_win}} = (\mathsf{PProve}_{\mathsf{bad_win}}, \mathsf{PVerify}_{\mathsf{bad_win}})$ for the relation

$$\mathcal{R}_{\mathsf{bad_win}} := \left\{ \begin{array}{l} (\vec{\mathsf{pk}}, (r, tag); \\ J, S, \{(r_i, m_i, \sigma_i)\}_{i \in S}) \end{array} \;\middle|\; \begin{array}{l} |S| = n - f,\ |J| = f + 1,\ \forall j \in J : r_j = r, \\ \forall i \in J : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (r, tag), \sigma_i) = 1, \\ m_i = \perp \vee m_i = (\mathsf{prop}_i, \pi_i) : \mathsf{valid}(\mathsf{prop}_i) = 0 \\ \vee\ \mathsf{NEVerify}(r, \mathsf{prop}_i, \pi_i, (\text{“proposal”}, tag)) = 1 \end{array} \right\}.$$

We further make use of a non-interactive proof system, which essentially shows that no unique party was elected. This is shown by having $n - f$ parties report their preferred winner, without having any party with $f + 1$ supporters (as would be the case if a unique party were elected). Concretely, we define the proof system $\mathsf{PS}_{\mathsf{no_win}} = (\mathsf{PProve}_{\mathsf{no_win}}, \mathsf{PVerify}_{\mathsf{no_win}})$ for the relation

$$\mathcal{R}_{\mathsf{no_win}} := \left\{ \left(\vec{\mathsf{pk}}, tag; S, \{(r_i, m_i, \sigma_i)\}_{i \in S}\right) \;\middle|\; \begin{array}{l} |S| = n - f,\ \forall i \in S : \\ \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (r_i, m_i, tag), \sigma_i) = 1, \\ \forall i \in S : |\{j \in S : r_j = r_i\}| \leq f \end{array} \right\}.$$

We present the protocol as pseudocode in Algorithms 16 and 17.

5.4.3 Analysis

We defer the proofs to Appendix D.4.

Theorem 6. *The protocol* PE *(cf. Algorithms 16 and 17) is a Proposal Election protocol with* $\alpha = \frac{2}{9}$ *and resilient to* $f < n/3$ *Byzantine parties. It requires a single call to* WDCG *plus* $O((\lambda\ell + m)n^2)$ *communication and* $O(1)$ *rounds, where* m *denotes the size of the input.*

6 Asynchronous DKG

In this section, we describe a simple ADKG construction. In the construction, parties call the PVSSExchange protocol to generate a transcript, and then agree on the transcript using the RecConsensus protocol. The RecConsensus protocol is the same one as the one used in the WDCG protocol: the No-Waitin' Hotstuff protocol, using the PE protocol in Section 5.4. In addition to the regular input to the protocol, parties also input the indices of the minimal and maximal indices of parties participating in the protocol, in this case 1 and n, respectively.

6.1 Definition

Definition 8 (DKG Protocol)**.** *Let* Π *be a protocol executed by* n *parties* $\{1, \ldots, n\}$*, where for all* $i \in [n]$*, party* i *outputs a* secret key share tsk_i*, a vector of* public key shares $(\mathsf{tpk}_1, \ldots, \mathsf{tpk}_n)$*, and a* public key pk*. We require the following properties for* Π *which each holds with overwhelming probability in the presence of a strongly adaptive adversary* $\mathcal{A}$ *corrupting at most* t *parties:*

- ***Termination.*** *All honest parties terminate and output the same public key* pk *and the same vector of public key shares* $(\mathsf{tpk}_1, \ldots, \mathsf{tpk}_n)$*.*
- ***Correctness.*** *There exists a deterministic algorithm* Reconstruct *that on input any set of* $t + 1$ *secret key shares* $\{\mathsf{tsk}_i\}_{i \in I}$*, where* $I \subseteq [n]$*, outputs the same unique secret key* sk*. Further,* sk *is a valid secret key for* pk*.*
- ***Secrecy.*** *$\mathcal{A}$'s success probability in the following experiment is negligible:*
  - Offline Phase. *Initialize a corruption set* $\mathcal{C} := \emptyset$ *and let* $\mathcal{H} := [n] \setminus \mathcal{C}$*. Run* $\mathcal{A}$ *on input par.*
  - Corruption Queries. *At any point of the experiment,* $\mathcal{A}$ *may corrupt a party by submitting an index* $i \in \mathcal{H}$*. In this case, return the internal state of* i *and update* $\mathcal{C} := \mathcal{C} \cup \{i\}$*. Henceforth,* $\mathcal{A}$ *has full control over* i*.*
  - Online Phase. *Initiate an execution of* Π *with* $\mathcal{A}$ *having control over parties in* $\mathcal{C}$*. Let* $y := \mathsf{pk} \leftarrow \Pi$ *be the public key output by honest parties, and let* sk *be the respective secret key.*
  - Winning Condition. *Let* s^* *denote the output of* $\mathcal{A}$*. Then,* $\mathcal{A}$ *is considered successful if and only if* $|\mathcal{C}| \leq f$ *and* $s^* = \mathsf{sk}$*.*

6.2 Construction

We describe our DKG protocol in Algorithm 18. Conceptually, the ADKG protocol calls the first parts of the WDCG protocol, generating a transcript, dispersing it, agreeing on one transcript, and finally outputting it. In more detail, parties start by calling the PVSSExchange protocol, and any party that manages to output a transcript disperses it. After dispersing the transcript, the disperser informs other parties that it succeeded, along with a proof so that they can proceed to the consensus protocol. Parties forward this information and call a consensus protocol in order to agree on a dispersed transcript. After agreeing on the same transcript, parties retrieve it and output it.

6.3 Analysis

We present our main result for the ADKG protocol. We defer the proof of properties to Appendix E, but provide the efficiency analysis here.

Theorem 7. *The protocol* ADKG *(cf. Algorithm 18) is an Asynchronous Distributed Key Generation protocol resilient to $f < n/3$ Byzantine parties, assuming hardness of the OMDL problem (cf. Definition 17). Furthermore, it runs in an expected communication complexity of $O(\lambda^2 n^{2+1/k})$ bits and expected $O(k)$ rounds.*

Proof. We count the communication and round complexity of the protocol. For $\ell = n^{1/k}$, we show that it has an expected communication complexity of $O(\ell \cdot \lambda^2 n^2)$ and an expected round complexity of $O(\log_\ell n)$, which is exactly our claim in the theorem statement. The protocol consists of a single call to the PVSSExchange protocol costing $O(\lambda^2 n^2)$, a constant number of all-to-all rounds sending messages of size $O(\lambda)$, a single call to the WDCG protocol and a single call to the RecConsensus with inputs of size $O(\lambda)$. The WDCG protocol has an expected communication complexity of $O(\ell \cdot \lambda^2 n^2)$ and the consensus costs an expected $O(\lambda n^2)$. In total, the ADKG protocol has an expected cost of $O(\ell \cdot \lambda^2 n^2)$. Further, every part of the protocol requires a constant expected number of rounds, except the WDCG, with a round complexity of $O(\log_\ell n)$. □

Observing two extremes for the value of ℓ, we can evaluate the complexities of the protocol. One could choose a constant fan-out in the recursion, e.g., $\ell = 2$, resulting in a quadratic communication complexity of $O(\lambda^2 n^2)$, but a round complexity of $O(\log n)$. On the other extreme, setting the number of children in the recursion to be larger, e.g., $\ell = n^{\frac{1}{2}}$ (or any other $\ell = n^\epsilon$ for a constant $\epsilon \in (0, 1]$), results in a very low round complexity of $O(\log_\ell n) = O(1)$ (or more generally $O(1/\epsilon)$), but a larger communication complexity of $O(\lambda^2 n^{2.5})$ (or more generally $O(\lambda^2 n^{2+\epsilon})$). We further note that $\ell = n^{1/k}$ remains quite small even for moderately large values of n. For instance, for $n = 10.000$, choosing $k = 5$ already yields a small $\ell \approx 6$. Finally, an interesting direction for future work is whether DKG protocols can achieve $O_\lambda(n^2)$ communication (including setup costs) in $o(\log n)$ rounds.

Acknowledgments

Gilad Stern is supported in part by Israel Science Foundation 2338/23, AFOSR Award FA9550-23-1-0387, AFOSR Award FA9550-23-1-0312, Israel Science Foundation (Grant No. 1336/22), and by the European Union (ERC, FTRC, 101043243). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or AFOSR. Julian Loss is supported by the European Union, ERC-2023-StG-101116713, and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[AAPS23] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. Perfectly secure asynchronous agreement on a core set in constant expected time. *IACR Cryptol. ePrint Arch.*, page 1130, 2023. 32

[ACD+23] Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. *Distributed Comput.*, 36(1):3–28, 2023. 3, 4, 31

[ACG+25] Ittai Abraham, Eli Chouatt, Yossi Gilad, Gilad Stern, and Sophia Yakoubov. Asynchronous algorand: Reaching agreement with near linear communication and constant expected time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '25, page 28–38, New York, NY, USA, 2025. Association for Computing Machinery. 6

[ACR21] Thomas Attema, Ronald Cramer, and Matthieu Rambaud. Compressed sigma-protocols for bilinear group arithmetic circuits and application to logarithmic transparent threshold signatures. In *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part IV*, page 526–556, Berlin, Heidelberg, 2021. Springer-Verlag. 4, 5

[AGL+23] Arasu Arun, Chaya Ganesh, Satya Lokam, Tushar Mopuri, and Sriram Sridhar. Dew: A transparent constant-sized polynomial commitment scheme. In *Public-Key Cryptography – PKC 2023: 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7–10, 2023, Proceedings, Part II*, page 542–571, Berlin, Heidelberg, 2023. Springer-Verlag. 11

[AJM+21] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 363–373. ACM, 2021. 1, 3, 4, 6, 7, 8, 9, 15, 19, 20, 31, 32

[AJM+23] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Annual International Cryptology Conference*, pages 39–70. Springer, 2023. 3, 4, 9, 19, 31, 32

[AMS19] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 337–346. ACM, 2019. 3, 4, 11, 31

[AS22] Ittai Abraham and Alexander Spiegelman. Provable broadcast, 2022. 11

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I*, page 561–586, Berlin, Heidelberg, 2019. Springer-Verlag. 13

[BCLZL23] Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes (almost) for free in dkg and mpc. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part I*, page 71–106, Berlin, Heidelberg, 2023. Springer-Verlag. 2

[BDK13] Michael Backes, Amit Datta, and Aniket Kate. Asynchronous computational vss with reduced communication complexity. In *Proceedings of the 13th International Conference on Topics in Cryptology*, CT-RSA'13, page 259–276, Berlin, Heidelberg, 2013. Springer-Verlag. 3, 31

[BdM94] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. 13, 34

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery. 11, 35

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. *Bit Optimal Distributed Consensus*, pages 313–321. Springer US, Boston, MA, 1992. 2

[BHK+24] Fabrice Benhamouda, Shai Halevi, Hugo Krawczyk, Yiping Ma, and Tal Rabin. Sprint: High-throughput robust distributed schnorr signatures. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*, page 62–91, Berlin, Heidelberg, 2024. Springer-Verlag. 31

[BK25] Renas Bacho and Alireza Kavousi. SoK: Dlog-Based Distributed Key Generation . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 614–632, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. 1, 2

[BL23] Renas Bacho and Julian Loss. Adaptively secure (aggregatable) pvss and application to distributed randomness beacons. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1791–1804, New York, NY, USA, 2023. Association for Computing Machinery. 4, 10, 32, 33, 48

[BLL+24] Renas Bacho, Christoph Lenzen, Julian Loss, Simon Ochsenreither, and Dimitrios Papachristoudis. Grandline: Adaptively secure dkg and randomness beacon with (log-)quadratic communication complexity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 941–955, New York, NY, USA, 2024. Association for Computing Machinery. 4, 5, 9, 10, 19, 31

[BLSW24] Renas Bacho, Julian Loss, Gilad Stern, and Benedikt Wagner. Harts: High-threshold, adaptively secure, and robust threshold schnorr signatures. In *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings, Part III*, page 104–140, Berlin, Heidelberg, 2024. Springer-Verlag. 31

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 431–448, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 10, 34

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16:185–215, 2003. 36

[Bra84] Gabriel Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 154–162, New York, NY, USA, 1984. Association for Computing Machinery. 4

[CGJ+99] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 1, 30

[CGMA85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395, 1985. 3

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 88–97, New York, NY, USA, 2002. Association for Computing Machinery. 3

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, page 6, 2001. 3, 31

[CW92] Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, 1992. 2

[DDL+24] Sourav Das, Sisi Duan, Shengqi Liu, Atsuki Momose, Ling Ren, and Victor Shoup. Asynchronous consensus without trusted setup or public-key cryptography. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3242–3256, New York, NY, USA, 2024. Association for Computing Machinery. 4, 8, 16, 32

[DR85] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985. 4, 31

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. 4

[DWZ23] Sisi Duan, Xin Wang, and Haibin Zhang. FIN: practical signature-free asynchronous common subset in constant time. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 815–829. ACM, 2023. 32

[DXKKR23] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5359–5376, Anaheim, CA, August 2023. USENIX Association. 1, 3, 4, 31, 32

[DXR24] Sourav Das, Zhuolun Xiang, and Ling Ren. Powers of tau in asynchrony. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. 2, 3, 30

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, page 350–357, USA, 1981. IEEE Computer Society. 10

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 416–431, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 10

[DYX+22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew K. Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534, 2022. 3, 30

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438, Oct 1987. 10

[FGL+25] Hanwen Feng, Yingzi Gao, Yuan Lu, Qiang Tang, and Jing Xu. Practical asynchronous distributed key reconfiguration and its applications. *Cryptology ePrint Archive*, 2025. 2, 3, 30

[FLM85] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '85, page 59–70, New York, NY, USA, 1985. Association for Computing Machinery. 2

[FLMT24] Hanwen Feng, Zhenliang Lu, Tiancheng Mai, and Qiang Tang. Making hash-based MVBA great again. *IACR Cryptol. ePrint Arch.*, page 479, 2024. 31, 32

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. 4

[FLT24a] Hanwen Feng, Zhenliang Lu, and Qiang Tang. Dragon: Decentralization at the cost of representation after arbitrary grouping and its applications to sub-cubic dkg and interactive consistency. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 469–479, New York, NY, USA, 2024. Association for Computing Machinery. 4, 31

[FLT24b] Hanwen Feng, Zhenliang Lu, and Qiang Tang. Optimal adaptively secure hash-based asynchronous common subset. *IACR Cryptol. ePrint Arch.*, page 1710, 2024. 32

[FM88] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 148–161, New York, NY, USA, 1988. Association for Computing Machinery. 32

[FT25] Hanwen Feng and Qiang Tang. Asymptotically optimal adaptive asynchronous common coin and dkg with silent setup. In *Advances in Cryptology – CRYPTO 2025: 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–21, 2025, Proceedings, Part III*, page 642–675, Berlin, Heidelberg, 2025. Springer-Verlag. 2, 3, 30, 31

[Fu25] Shihui Fu. Improved constant-sized polynomial commitment schemes without trusted setup. *IACR Communications in Cryptology*, 2(2), 2025. 4, 5, 11

[GHMR18] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: Removing private-key generator from ibe. In *Theory of Cryptography: 16th International Conference, TCC 2018, Panaji, India, November 11–14, 2018, Proceedings, Part I*, page 689–718, Berlin, Heidelberg, 2018. Springer-Verlag. 30

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, January 2007. 30

[GJM+21] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, page 147–176, Berlin, Heidelberg, 2021. Springer-Verlag. 4, 10, 19, 30, 31

[GKKS+22] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan Garay, editors, *Financial Cryptography and Data Security*, pages 296–315, Cham, 2022. Springer International Publishing. 1

[GKPW24] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VII*, volume 14926 of *Lecture Notes in Computer Science*, pages 352–386. Springer, 2024. 30

[GLL+22a] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 246–257. IEEE, 2022. 3, 31

[GLL+22b] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous BFT closer to practice. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. 4, 31

[GS24] Jens Groth and Victor Shoup. Fast batched asynchronous distributed key generation. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*, page 370–400, Berlin, Heidelberg, 2024. Springer-Verlag. 31

[KKMS20] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1751–1767, New York, NY, USA, 2020. Association for Computing Machinery. 1, 3, 31

[KNR24] Jovan Komatovic, Joachim Neu, and Tim Roughgarden. Toward optimal-complexity hash-based asynchronous MVBA with optimal resilience. *IACR Cryptol. ePrint Arch.*, page 1682, 2024. 32

[KO17] Jihye Kim and Hyunok Oh. Forward-secure digital signature schemes with optimal computation and storage of signers. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors, *ICT Systems Security and Privacy Protection*, pages 523–537, Cham, 2017. Springer International Publishing. 10

[KRY22] Sebastian Kolby, Divya Ravi, and Sophia Yakoubov. Constant-round YOSO MPC without setup. Cryptology ePrint Archive, Paper 2022/187, 2022. 6

[LLTW20] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020. 4, 6, 31

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. 2, 31

[MR21] Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 4, 5

[MVR99] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 120, USA, 1999. IEEE Computer Society. 10, 34

[NRBB24] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. In Christina Pöpper and Lejla Batina, editors, *Applied Cryptography and Network Security*, pages 105–134, Cham, 2024. Springer Nature Switzerland. 30

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 522–526, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. 1, 30

[Ram23] Matthieu Rambaud. Adaptively secure consensus with linear complexity and constant round under honest majority in the bare PKI model, and separation bounds from the idealized message-authentication model. *IACR Cryptol. ePrint Arch.*, page 1757, 2023. 7, 15

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. 13, 33

[SBKN24] Nibesh Shrestha, Adithya Bhat, Aniket Kate, and Kartik Nayak. Synchronous distributed key generation without broadcasts. *IACR Communications in Cryptology*, 07 2024. 31

[Sta96] Markus Stadler. Publicly verifiable secret sharing. In *International Conference on the Theory and Application of Cryptographic Techniques*, 1996. 4, 10

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. 1

[ZDL+23] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. Practical asynchronous distributed key generation: Improved efficiency, weaker assumption, and standard model. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–581, 2023. 3

A Related Work

We discuss related work on DKG and consensus protocols.

Recent work on subcubic ADKG [FT25]. Very recently, Feng and Tang [FT25] constructed an asynchronous strong common coin protocol with a communication cost of $O(\lambda n^2)$ bits and $O(1)$ rounds. Their protocol is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$, which is optimal. Using a generic transformation from strong common coins to ADKG at the cost of $O(\lambda^2 n^2)$ added communication and reliance on secure erasures [FGL+25], this yields an asynchronous DKG protocol with a communication cost of $O(\lambda^2 n^2)$ bits and $O(1)$ rounds, assuming secure erasures. (We note that the work [FGL+25] actually gives $O(\lambda\kappa n^2)$ added communication, where κ is a statistical security parameter. For our protocol, we could also replace the factor λ^2 in the communication with $\lambda\kappa$. However, since the protocols are required to provide a strict λ-bit security, κ has to be in a linear relation with λ, and we therefore choose to avoid this subtle distinction between κ and λ in our work.) However, their strong common coin protocol crucially relies on the silent-setup threshold encryption scheme of [GKPW24]. And there are two critical points to this.

First, in a "silent setup", each party is required to register an $O(\lambda n)$ bit-sized public key on the public bulletin-board. In particular, during the setup, each party has to read $\Omega(\lambda n^2)$ bits, which leads to an overall communication cost of $\Omega(\lambda n^3)$ bits. This means that the whole protocol becomes cubic in communication, as the communication of the setup should also be taken into account. We note that the authors in [FT25] mention that this cost can be reduced by assuming a "key curator" as defined in [GHMR18]. This key curator reads the linear-sized silent-setup public keys of parties, verifies them against the powers-of-tau reference string, and then aggregates the public keys to a single $O(\lambda n)$-sized aggregate public key which is sufficient for the silent-setup threshold encryption scheme. In particular, parties only need to read the linear-sized aggregate public key, which incurs $O(\lambda n^2)$ total communication. But the key curator needs to be trusted for the computation. Further, we note that the definition of key curator in [GHMR18] assumes poly-logarithmic public parameters and keys, but the construction of Feng and Tang uses linear-sized public parameters (specifically, powers-of-tau reference string) and linear-sized keys of parties.

Second, the silent setup itself requires a powers-of-tau setup [NRBB24], which is a structured reference string (SRS) with a (uniformly random) trapdoor τ that must remain secret from anyone for the applications to be secure. Importantly, the powers-of-tau setup either requires a trusted third party that generates the string and then discards τ (along with other randomness), or it is generated through a distributed protocol. However, existing constructions for generating the powers-of-tau setup in asynchronous networks (without a trusted third party) [DXR24], in fact, first execute an ADKG protocol (specifically, the one in [DYX+22] with $O(\lambda n^3)$ communication cost) to produce threshold keys. These keys are then used in $O(\log n)$ sequential iterations of all-to-all communication to obtain the desired powers-of-tau reference string. In particular, without a subcubic ADKG to begin with, we cannot generate the powers-of-tau setup with subcubic communication cost.[11] (Worth mentioning, the $O(\log n)$ sequential iterations give $O(\log n)$ added round complexity.) In summary, the strong common coin protocol by Feng and Tang [FT25] either requires $O(\lambda n^3)$ communication cost and $O(\log n)$ rounds to establish their setup or a trusted third party (and additionally a key curator). We note that the powers-of-tau setup is universal in the sense that once it is set up, it can be used by any set of n parties. This still leaves our question open, as the setup needs to be generated in the first place.

DKG protocols. Starting with the seminal work of Pedersen [Ped91], numerous DKG protocols have been proposed. Most of them [CGJ+99, GJKR07, GJM+21, Ped91] assume a well-behaved synchronous network, where message delays are upper bounded by some known finite time Δ and the parties are assumed to have synchronized clocks. Abstractly speaking, existing DKG protocols[12] typically follow the same two-phase structure consisting of a sharing phase and a consensus phase. In the typical sharing phase, each party chooses a random secret (in the space of secrets) and shares it among all parties via a verifiable secret sharing (VSS)

[11] The distributed protocols for powers-of-tau in synchronous networks have similar complexities, as they require running $O(n)$ sequential Byzantine broadcasts on messages of size $O(\lambda n)$, once by each party - yielding an $\Omega(\lambda n^3)$ inherent lower bound.

[12] This also includes asynchronous DKG protocols.

scheme. In the consensus phase, parties then agree on a subset of at least $f+1$ parties[13] whose secrets should be included in the secret key (as sum of their individual secrets). Crucially, the sharing phase consists of n parallel instances of a Byzantine broadcast protocol. However, well-known results on Byzantine broadcast show an inherent lower bound of $\Omega(n^2)$ message complexity in communication per instance [ACD+23, DR85] (against a strongly adaptive adversary, even when a threshold key setup is assumed), which directly leads to $\Omega(n^3)$ message complexity for the sharing phase only. There are few recent works [SBKN24, GJM+21, BLL+24, FLT24a, FLMT24] that deviate from this blueprint, but [SBKN24, GJM+21] still require cubic communication cost and [FLMT24] assumes strong common coins.[14] The protocol [FLT24a] uses network partitioning (can be thought of recursion of depth one) and achieves a communication cost of $O(\lambda n^{2.5})$ bits and terminates in $O(n)$ rounds, while [BLL+24] uses recursion of depth $\log n$ and achieves a communication cost of $O(\lambda n^2 \log n)$ bits and terminates in $O(n)$ rounds. In our technical overview, we will elaborate on the design of [BLL+24] and also explain why recursion in asynchronous networks is highly non-trivial and seems to fail.

Starting with the work of Kokoris-Kogias et al. [KKMS20], asynchronous DKG protocols have attracted lots of attention in recent years [AJM+23, AJM+21, DXKKR23, GLL+22a, KKMS20]. In an asynchronous network, messages may experience arbitrary delays, with the only guarantee that messages sent between honest parties will eventually be delivered. In particular, the local clocks and processing speeds of honest parties are not necessarily synchronized, and parties may even start the protocol at different times. The main challenge in designing asynchronous protocols comes from the fact that it is, in general, not possible to distinguish a slow honest party from a silent Byzantine party. All existing asynchronous DKG protocols follow the previously stated two-phase structure: sharing phase and subsequent consensus phase. Crucially, their sharing phase, except the one in [FT25], is executed by n parallel instances of an asynchronous VSS (AVSS) protocol, where each of the n parties initiates one of these instances. However, well-known results on AVSS show an inherent lower bound of $\Omega(n^2)$ message complexity in communication per instance [BDK13], which directly leads to $\Omega(n^3)$ message complexity for the sharing phase only. Further, the consensus phase is typically implemented via a multivalued validated Byzantine agreement (MVBA) protocol [CKPS01], and all prior MVBA protocols *without* a threshold key setup[15] have a communication cost of $\Omega(\lambda n^3)$ bits. In particular, both the sharing and consensus phases suffer from cubic communication overhead. Some other recent works [BLSW24, BHK+24, GS24] design batched asynchronous DKG protocols that generate many secret keys at once using linear randomness extractors. These protocols assume an MVBA protocol, which have cubic communication cost (without assuming a threshold key setup in the first place) as stated before, and their sharing phase also has cubic communication cost.

Consensus protocols. Consensus protocols, both synchronous and asynchronous, have been the subject of extensive research for over 40 years [LSP82]. There are various types of consensus protocols, such as binary or multivalued Byzantine agreement, asynchronous common subset, and more. A consensus protocol allows a set of n parties, each starting with an input value v_i, to agree on a common output value v, with additional condition(s) depending on the type of consensus protocol. For instance, in Byzantine agreement, if all parties start with the same input value, then their output value v has to be that input value. A particularly useful consensus protocol is a multivalued *validated* Byzantine agreement (MVBA) [CKPS01], where the output value also have to be meaningful with respect to a predefined external validity function. For instance, in the context of DKG protocols, the validity function could check if the DKG transcript that parties output (i.e., the public keys and potentially auxiliary data) is correctly formed as a DKG, and not just any agreed-upon value. In recent years, several (asynchronous) MVBA protocols with $O(\lambda n^2)$ bits communication cost have been proposed [AMS19, GLL+22b, LLTW20], but all of these protocols require a trusted dealer to set up threshold keys for the parties (i.e., threshold shares of a secret key), which are used to implement threshold signatures as certificates and strong common coins. We stress that these works achieve the optimal $O(n^2)$ message-complexity [ACD+23, DR85] (and potentially also the optimal

[13] So that the adversary corrupting f parties does not know all the secrets that are included in the key.

[14] Strong common coins either require a trusted dealer or a DKG protocol.

[15] These protocols, in general, assume a trusted dealer at setup that threshold-shares a secret key between the parties. This threshold key setup is then used to implement (non-interactive) threshold signatures and strong common coins.

bit-complexity). These protocols and their reliance on a trusted dealer have spurred research into both ADKG and the closely related task of reducing the costs of MVBA *without* a threshold key setup. Recent works include [AAPS23, AJM+23, AJM+21, DDL+24, DXKKR23, DWZ23, FLMT24, FLT24b, KNR24], but all of these protocols either have $O(\lambda n^3)$ communication cost, assume strong common coins, or both.

A.1 More Background

Resolving FLP efficiently. We recall the Gather framework from Abraham et al. [AJM+21], which gives the most efficient way from the literature to resolve this problem.[16] The goal is to design a *weak common coin*[17] with constant success probability $\delta \in (0, 1]$, which can then be used to establish asynchronous consensus following Feldman and Micali [FM88]. Namely, given a weak common coin, such a consensus protocol proceeds in iterations of constant rounds, where parties exchange proposals and a weak common coin then retroactively selects one of the proposals. Crucially, if the coin selects an honest party's proposal, all parties can immediately output its proposal and terminate. Given that δ is constant and there is a $\frac{2}{3}$-fraction of honest parties in the system, the protocols terminates, in expectation, constant rounds.

Coming back to the weak common coin: First, each party $i \in [n]$ samples a random secret s_i and shares it via an AVSS. Then, it waits for $f+1$ AVSS instances to complete at it and stores the corresponding party indices in a set $T_i \subseteq [n]$. The idea is that the encoded secret $\tilde{s}_i$ in T_i (derived from the individual s_j over $j \in T_i$) will later be used to assign a *random rank* r_i to party i (e.g., from a random oracle as $r_i := \mathsf{H}(i, \tilde{s}_i)$). Thus, each party is assigned a random rank, which is hidden until reconstruction starts. Next, parties run a (binding cover) Gather protocol [DDL+24] with party i having input value T_i, and it outputs a set S_i of at least $n - f$ received input values. Recall that a (binding cover) Gather protocol does not solve consensus but instead guarantees the existence of some core set S (and cover set $\overline{S}$) of size at least $n - f$ such that $S \subseteq S_i$ (and $S_i \subseteq \overline{S}$) for all honest parties i. After the Gather, parties start reconstructing all secrets s_i, the ranks are derived, and each party i selected the party $\ell_i \in T_i$ with the highest rank as the elected leader. Intuitively, the party with maximal rank *among all parties* is in the core set S with probability $\frac{2}{3}$. Thus, the probability that this party is honest and in the core set is $\frac{1}{3}$. This gives a weak leader election with success probability $\delta = \frac{1}{3}$. Finally, hashing the elected leader's rank r_{ℓ_i} via another random oracle $\mathsf{H}(\ell_i, r_{\ell_i})$ yields the (weak) common coin with the same succes probability $\delta = \frac{1}{3}$.

B Formal Definitions

In this section, we give formal definitions for the primitives introduced in Section 3.1. Henceforth, let $\mathbb{G}$ be a cyclic group of prime order p with independent generators $g, h \in \mathbb{G}$.

Definition 9 (Aggregatable PVSS Scheme [BL23])**.** *A (t, n)-threshold aggregatable PVSS (APVSS) scheme over group $\mathbb{G}$ is a tuple of algorithms $\mathsf{APVSS} = (\mathsf{KGen}, \mathsf{Encr}, \mathsf{Decr}, \mathsf{Dist}, \mathsf{Agg}, \mathsf{Contr}, \mathsf{Ver}, \mathsf{Rec})$ such that:*

- $\mathsf{KGen}(par) \to (\mathsf{pk}, \mathsf{sk})$: *The randomized* key generation algorithm *takes as input system parameters par. It outputs a public encryption key pk and a secret decryption key sk.*
- $\mathsf{Encr}(\mathsf{pk}, m) \to c$: *The randomized* encryption algorithm *takes as input an encryption key pk and a message m. It outputs a ciphertext c. We may also write $\mathsf{Encr}_{\mathsf{pk}}(m)$ instead of $\mathsf{Encr}(\mathsf{pk}, m)$.*
- $\mathsf{Decr}(\mathsf{sk}, c) \to m$: *The deterministic* decryption algorithm *takes as input a decryption key sk and a ciphertext c. It outputs a message m (optionally with a proof of correct decryption). We may sometimes also write $\mathsf{Decr}_{\mathsf{sk}}(c)$ instead of $\mathsf{Decr}(\mathsf{sk}, c)$. Further, for all messages m and keys $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{KGen}(par)$, we require that $\Pr[\mathsf{Decr}_{\mathsf{sk}}(\mathsf{Encr}_{ek}(m)) = m] = 1$.*

[16] We note that concretely more efficient variants have recently been proposed [DDL+24], which also extend the Gather protocol with a valuable feature called *binding cover*. We will elaborate on that later.

[17] A δ-weak common coin allows parties to agree on a random value with some probability $\delta > 0$, where agreement is "oblivious" - parties do not know whether consensus was reached.

- $\mathsf{Dist}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \to (\vec{E}, \pi)$: *The randomized* secret sharing algorithm *takes as input a list of n encryption keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. It outputs a vector of encrypted shares $\vec{E} := (\mathsf{Encr}_{\mathsf{pk}_1}(S_1), \ldots, \mathsf{Encr}_{\mathsf{pk}_n}(S_n))$ along with a proof π, where $S_1, \ldots, S_n$ are (t, n)-threshold shares of a secret $S = S_0 \in \mathbb{G}$. We also refer to the tuple $T := (\vec{E}, \pi)$ as a* PVSS transcript*.*

- $\mathsf{Agg}(\{(\vec{E}_i, \pi_i)\}_{i\in[k]}) \to ((\vec{E}, \pi), \vec{w})$: *The deterministic* aggregation algorithm *takes as input PVSS transcripts $(\vec{E}_1, \pi_1), \ldots, (\vec{E}_k, \pi_k)$ for $k \in \mathbb{N}$. It outputs an* (aggregated) PVSS transcript *$T := (\vec{E}, \pi)$ along with a weight vector $\vec{w} \in \mathbb{N}_0^n$ (indication contributing parties and their weights).*

- $\mathsf{Contr}((\vec{E}, \pi), \vec{w}) \to I$: *The deterministic* contributor identifier algorithm *takes as input an (aggregated) PVSS transcript $T = (\vec{E}, \pi)$ with weight vector $\vec{w}$. It outputs a set of indices $I \subseteq [n]$ specifying the contributors.*

- $\mathsf{Ver}((\vec{E}, \pi), \vec{w}, (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)) \to 0/1$: *The deterministic* verification algorithm *takes as input an (aggregated) PVSS transcript $T = (\vec{E}, \pi)$ with weight vector $\vec{w}$, and a list of n encryption keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. It outputs 1 (valid transcript) or 0 (invalid transcript).*

- $\mathsf{Rec}(S_1, \ldots, S_{t+1}) \to S$: *The deterministic* reconstruction algorithm *takes as input $t + 1$ (decrypted) shares $S_1, \ldots, S_{t+1}$. It outputs a secret $S \in \mathbb{G}$. (If there are more inputs, choose the first $t + 1$ lexicographically.)*

Definition 10 (Aggregated Unpredictability [BL23])**.** *Let* APVSS *be a (t, n)-threshold aggregatable PVSS scheme. For any adversary $\mathcal{A}$ corrupting at most f parties, its probability of winning in the following security experiment* AggUnpred *is negligible in λ.*

- ***Setup Phase.*** *For all $i \in [n]$, run $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Keys}(\mathsf{par}, i)$. On input* par *and $\{\mathsf{pk}_i\}_{i\in[n]}$, the adversary $\mathcal{A}$ outputs an index set $\mathcal{C} \subseteq [n]$ with $|\mathcal{C}| \leq f$ of initially corrupted parties, together with updated public keys $\{\hat{\mathsf{pk}}_j\}_{j\in\mathcal{C}}$. For all $j \in \mathcal{C}$, set $\mathsf{pk}_j := \hat{\mathsf{pk}}_j$. Let $\mathcal{H} := [n] \setminus \mathcal{C}$ denote the set of honest parties. Let $\mathcal{L} := \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ be the public key list.*

- ***Transcript Oracle.*** *The adversary $\mathcal{A}$ is given oracle access to $\mathcal{O}_{\mathsf{PVSS}}$. On query $(\mathsf{givePVSS}, i)$ for some $i \in \mathcal{H}$, the oracle runs $\mathsf{ADist}(\mathsf{sk}_i, \mathcal{L})$, and returns the resulting transcript T to $\mathcal{A}$.*

- ***Corruption Oracle.*** *The adversary $\mathcal{A}$ is given oracle access to $\mathcal{O}_{\mathsf{corr}}$. On query $i \in \mathcal{H}$, the oracle returns party i's internal state, including the secret key. Update $\mathcal{C} := \mathcal{C} \cup \{i\}$ and $\mathcal{H} := \mathcal{H} \setminus \{i\}$.*

- ***Winning Condition.*** *The adversary outputs an aggregated transcript $T^* = ((\vec{E}^*, \pi^*), \vec{w}^*)$ and a value S^*. The adversary wins if and only if the following three conditions hold: $\mathsf{AVer}(\mathcal{L}, T^*) = 1$, $\mathsf{Contr}(T^*) \cap \mathcal{H} \neq \emptyset$, and $S^* = \mathsf{Rec}\big(\{\mathsf{Dec}_{\mathsf{sk}_i}(E_i^*)\}_{i\in\mathcal{H}}\big)$. Informally, these conditions require that the aggregated transcript is valid, that it contains at least one contribution from an honest party, and that S^* is exactly the secret encoded by the transcript.*

Definition 11 (Reed-Solomon Code [RS60])**.** *A Reed-Solomon code with parameters $(q, b) \in \mathbb{N} \times \mathbb{N}$ is a tuple of deterministic algorithms $\Sigma = (\mathsf{Encode}, \mathsf{Decode})$ with the following properties:*

- $\mathsf{Encode}(m) \to (s_1, \ldots, s_q)$: *The deterministic* encoding algorithm *takes as input a message m (which is internally split into b data symbols $m_1, \ldots, m_b$). It outputs a code word $(s_1, \ldots, s_q)$ of length q. Knowledge of any b elements of the code word uniquely determines the input message and the remaining of the code word.*

- $\mathsf{Decode}(s_1, \ldots, s_q) \to m$: *The deterministic* decoding algorithm *takes as input a code word $(s_1, \ldots, s_q)$ of length q. It outputs a decoded message m. This algorithm tolerates up to c errors and d erasures in a code word $(s_1, \ldots, s_q)$ if and only if $q - b \geq 2c + d$.*

Definition 12 (Cryptographic Accumulator [BdM94])**.** *A cryptographic accumulator scheme is a tuple of algorithms* $\Sigma = (\mathsf{Gen}_{\mathsf{Acc}}, \mathsf{Eval}_{\mathsf{Acc}}, \mathsf{Witness}_{\mathsf{Acc}}, \mathsf{Verify}_{\mathsf{Acc}})$ *with the following properties:*

- $\mathsf{Gen}_{\mathsf{Acc}}(n) \to ak$: *The randomized* accumulator key generation algorithm *takes as input an accumulation threshold* n*. It outputs a public accumulator key* ak*.*
- $\mathsf{Eval}_{\mathsf{Acc}}(ak, D) \to z$: *The deterministic* evaluation algorithm *takes as input an accumulator key* ak *and a list of* n *elements* $D := (d_1, \ldots, d_n)$*. It outputs an accumulation value* z *for* D*.*
- $\mathsf{Witness}_{\mathsf{Acc}}(ak, z, d, D) \to w/\perp$: *The possibly randomized* witness generation algorithm *takes as input an accumulator key* ak*, an accumulation value* z*, an element* d*, and a list of elements* D*. It outputs* $\perp$ *if* $d \notin D$*, and a witness* w *otherwise.*
- $\mathsf{Verify}_{\mathsf{Acc}}(ak, z, d, w, D) \to 0/1$: *The deterministic* verification algorithm *takes as input an accumulator key* ak*, an accumulation value* z*, an element* d*, a witness* w*, and a list of element* D*. It outputs 1 (accept) if* w *is a valid proof for membership* $d \in D$ *and 0 (reject) otherwise.*

Definition 13 (Key-Evolving Digital Signature Scheme [BM99])**.** *A key-evolving digital signature scheme is a tuple of algorithms* $\mathsf{KDS} = (\mathsf{KGen}_{\mathsf{KDS}}, \mathsf{Update}_{\mathsf{KDS}}, \mathsf{Sign}_{\mathsf{KDS}}, \mathsf{Verify}_{\mathsf{KDS}})$ *with the following properties:*

- $\mathsf{KGen}_{\mathsf{KDS}}(\lambda, T) \to (\mathsf{pk}, \mathsf{sk}_0)$: *The randomized* key generation algorithm *takes as input the security parameter* λ *and the total number of periods* $T \in \mathbb{N}$ *over which the scheme will operate. It outputs a base public key* pk *and corresponding base secret key* sk_0*.*
- $\mathsf{Update}_{\mathsf{KDS}}(\mathsf{sk}_r) \to \mathsf{sk}_{r+1}$: *The deterministic* secret key update algorithm *takes as input a secret key* sk_r *of some period* $r \in \mathbb{N}_0$*. It outputs an updated secret key* sk_{r+1} *for the period* $r + 1$*.*
- $\mathsf{Sign}_{\mathsf{KDS}}(\mathsf{sk}_r, m) \to \sigma$: *The possibly randomized* signature generation algorithm *takes as input the current secret key* sk_r *and a message* m*. It outputs a signature* σ_r *for the period* r*.*
- $\mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}, m, \sigma_r) \to 0/1$: *The deterministic* verification algorithm *takes as input the public key* pk*, a message* m*, and a signature* σ_r*. It outputs 1 (accept) if* σ_r *is a valid signature for period* r *and 0 (reject) otherwise.*

Definition 14 (Verifiable Random Function [MVR99])**.** *A verifiable random function is a tuple of algorithms* $\mathsf{VRF} = (\mathsf{Gen}_{\mathsf{VRF}}, \mathsf{Eval}_{\mathsf{VRF}}, \mathsf{Verify}_{\mathsf{VRF}})$ *with the following properties:*

- $\mathsf{Gen}_{\mathsf{VRF}}(\lambda) \to (\mathsf{pk}, \mathsf{sk})$: *The randomized* key generation algorithm *takes as input the security parameter* λ*. It outputs a public key* pk *and a corresponding secret key* sk*.*
- $\mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m) \to (u, \pi)$: *The deterministic* function evaluation algorithm *takes as input a secret key* sk *and a message* m*. It outputs a function value* $u \in S$ *along with a proof* π*. Here,* S *is a finite set denoting the codomain of the function.*
- $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u, \pi) \to 0/1$: *The deterministic* verification algorithm *takes as input the public key* pk*, a message* m*, an output* u*, and a proof* π*. It outputs 1 (accept) or 0 (reject).*

Additionally, VRF *must satisfy the following security properties.*

- ***Correctness.*** *For all* $\lambda \in \mathbb{N}$*, for all* $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}_{\mathsf{VRF}}(\lambda)$*, for all* m*, and for all* (u, π) *in the image of* $\mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m)$*, it holds* $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u, \pi) = 1$*.*
- ***Unique Provability.*** *For all possible* pk *(not necessarily in the image of* $\mathsf{Gen}_{\mathsf{VRF}}$*), for all* m*, for all* $u_1, u_2 \in S$*, and for all possible proofs* π_1, π_2*, the following implication holds:*

$$(\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u_1, \pi_1) = \mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u_2, \pi_2) = 1) \implies (u_1 = u_2)\,.$$

Informally, this means that for every message m*, there exists a valid proof* π *for at most one function value* u*.*

- ***Pseudorandomness.*** *For all PPT adversaries $\mathcal{A}$, its advantage in the pseudorandomness experiment defined hereafter is negligible:*
  - Offline Phase. *Initialize set $\mathcal{M} := \emptyset$. Run the key generation algorithm on input λ to obtain keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}_{\mathsf{VRF}}(\lambda)$. Run $\mathcal{A}$ on input pk.*
  - Evaluation Queries. *At any point of the experiment, $\mathcal{A}$ may submit a message m. In this case, return $(u, \pi) \leftarrow \mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m)$ and update $\mathcal{M} := \mathcal{M} \cup \{m\}$.*
  - Online Phase. *When $\mathcal{A}$ outputs a message m^*, run the evaluation algorithm on it to obtain $(u_0, \pi) \leftarrow \mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m^*)$. Sample a bit $b \leftarrow\!\!\$\ \{0,1\}$ and a value $u_1 \leftarrow\!\!\$\ S$ uniformly at random.*
  - Winning Condition. *On input u_b, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$. Return 1 if $b' = b$ and $m^* \notin \mathcal{M}$. Otherwise, return 0.*

Definition 15 (Non-Interactive Proof System [BFM88])**.** *Let $\mathcal{R}$ be an* **NP** *relation and H be a random oracle. A non-interactive proof system for $\mathcal{R}$ with respect to H is a tuple of PPT algorithms $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVerify})$ with oracle access to H with the following properties:*

- $\mathsf{PProve}^{\mathsf{H}}(x, w) \to \pi$: *The randomized* proof generation algorithm *takes as input a statement x and a witness w. It outputs a proof π.*
- $\mathsf{PVerify}^{\mathsf{H}}(x, \pi) \to 0/1$: *The deterministic* verification algorithm *takes as input a statement x and a proof π. It outputs 1 (accept) or 0 (reject).*

We require completeness *to hold: For any $(x, w) \in \mathcal{R}$, we have*

$$\Pr\left[\mathsf{PVerify}^{\mathsf{H}}(x, \pi) = 1 \;\middle|\; \pi \leftarrow \mathsf{PProve}^{\mathsf{H}}(x, w)\right] = 1.$$

Further, we also require soundness *to hold. Concretely, we say that PS is sound if for every PPT algorithm $\mathcal{A}$, the following advantage is negligible:*

$$\Pr\left[b = 1 \;\wedge\; \forall w : (x, w) \notin \mathcal{R} \;\middle|\; (x, \pi) \leftarrow \mathcal{A}^{\mathsf{H}},\; b := \mathsf{PVerify}^{\mathsf{H}}(x, \pi)\right].$$

We also require the proof system to be zero-knowledge*. Concretely, we say that PS is zero-knowledge, if there is a possibly stateful PPT algorithm PSim such that for every (potentially unbounded) adversary $\mathcal{A}$ with polynomial-time access to H, the following advantage is negligible:*

$$\left|\Pr\left[\mathcal{A}^{\mathsf{H},\mathsf{Oracle}_0} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{H}^{\mathsf{PSim}},\mathsf{Oracle}_1} = 1\right]\right|,$$

where $\mathsf{H}^{\mathsf{PSim}}$ denotes a random oracle simulated by PSim, and Oracle_b for $b \in \{0,1\}$ takes as input pairs $(x, w) \in \mathcal{R}$ and outputs $\pi \leftarrow \mathsf{PProve}^{\mathsf{H}}(x, w)$ if $b = 0$ and $\pi \leftarrow \mathsf{PSim}(x)$ if $b = 1$.

Definition 16 (Verifiable Consensus)**.** *In a verifiable consensus protocol $(\mathsf{Consensus}, \mathsf{ConsensusVerify})$, every party has an input x_i and outputs a value y_i and a proof π_i. Parties have access to an external validity function valid. Assuming all honest parties participate in the protocol and that $\mathsf{valid}(x_i) = 1$ for every honest i, the protocol has the following properties:*

- ***Correctness.*** *Every honest party that completes the protocol outputs the same y_i.*
- ***External Validity.*** *If some honest party outputs y_i, then $\mathsf{valid}(y_i) = 1$.*
- ***Verifiability.*** *If some honest party outputs y_i, π_i, then $\mathsf{ConsensusVerify}(y_i, \pi_i) = 1$. Moreover, the adversary cannot generate any y, π such that $y \neq y_i$ and $\mathsf{ConsensusVerify}(y, \pi) = 1$.*
- ***Termination.*** *All honest parties almost-surely terminate.*

Note that most consensus constructions have additional properties disallowing trivial solutions such as α-Quality (there is at least an α probability of outputting an honest party's input), validity (if all honest parties have the same input, they output that value), or weak validity (if all parties are honest and have the same input, they output that value). While the protocols we use have these properties, this is not necessary in our construction.

Definition 17 (OMDL Assumption [BNPS03])**.** *We denote by* DL_g *an oracle that on input* $\xi := g^z \in \mathbb{G}$ *returns the discrete logarithm* $z \in \mathbb{Z}_p$ *of* ξ *(to the base* g*). For an algorithm* $\mathcal{A}$ *and* $k \in \mathbb{N}$*, we consider the following security experiment:*

- Offline Phase. *Sample* $(z_1, \ldots, z_k) \leftarrow \mathbb{Z}_p^k$ *and set* $\xi_i := g^{z_i} \in \mathbb{G}$ *for all* $i \in [k]$*.*
- Online Phase. *Run* $\mathcal{A}$ *on input* $(\mathbb{G}, p, g, h)$ *and* $(\xi_1, \ldots, \xi_k)$*. In this phase,* $\mathcal{A}$ *gets access to the discrete logarithm oracle* DL_g*.*
- Winning Condition. *Let* $(z'_1, \ldots, z'_k)$ *denote the output of* $\mathcal{A}$*. Return 1 if (i)* $z'_i = z_i$ *for all* $i \in [k]$*, and (ii)* DL_g *was queried at most* $k-1$ *times during the online phase. Otherwise, return 0.*

We say that the OMDL assumption of degree k *holds relative to* $(\mathbb{G}, p, g, h)$ *if for any efficient algorithm* $\mathcal{A}$*, the probability that the above experiment returns 1 is negligible in* λ*.*

C Proofs for Section 4

In this section, we present the proofs for our helper primitives in Section 4.

C.1 Non-Equivocation Rounds (Section 4.1)

Theorem 8. *The pair* (NonEquiv, NEVerify) *as described in Algorithms 1 and 2 is a non-equivocation protocol resilient to f Byzantine parties if $n > 3f$.*

Proof. Each property is proven separately.

Validity. We have to show that if an honest party i outputs a tuple (out_i, π_i) from the non-equivocation protocol, then $\mathsf{NEVerify}(i, \mathsf{out}_i, \pi_i, tag) = 1$. So assume an honest party i outputs (out_i, π_i) from the protocol, then by construction π_i is a threshold signature of weight $n-f$ on out_i. In particular, the underlying threshold signature verification algorithm $\mathsf{PVerify}_{\mathsf{TS}}$ verifies and so does NEVerify. This shows the validity property of the protocol.

Non-Equivocation. We have to show that an adversary cannot generate two tuples (j, m, π) and (j, m', π') for the same party j such that $m \neq m'$ and $\mathsf{NEVerify}(j, m, \pi, tag) = \mathsf{NEVerify}(j, m', \pi', tag) = 1$. By way of contradiction, assume that there are two messages $m \neq m'$ with respective proofs π, π' that verify. By construction, we know that π and π' are verifying threshold signatures of weight $n-f$ each. In particular, by quorum intersection, there is at least one honest party i that has signed both m and m'. But this clearly contradicts the requirement that each (honest) party only signs the first received message $\langle \text{“help”}, m_i, tag \rangle$ from any other party, including party j. This shows the non-equivocation property of the protocol.

Liveness. We have to show that every honest party ventually outputs a proof and terminates. Since there are at least $n-f$ honest parties, each party will eventually receive $n-f$ signatures on its input message. As such, it will also be able to generate a proof π and terminate. This shows the liveness property of the protocol. □

Theorem 9. *The pair* (AbandonableNonEquiv, ANEVerify) *as described in Algorithms 3 and 4 is an abandonable non-equivocation protocol resilient to f Byzantine parties if $n > 3f$.*

Proof. Each property is proven separately.

Validity and Non-Equivocation. The same arguments as in Theorem 8 applies.

Abandonability. We have to show that at the time $f+1$ honest parties abandon the protocol, then for every $i \in [n]$ there is a value m_i such that the adversary cannot generate a tuple (m'_i, π'_i) such that $m'_i \neq m_i$ and $\mathsf{ANEVerify}(i, m'_i, \pi'_i, tag) = 1$. By way of contradiction, assume that there is an index $i \in [n]$ for which the adversary can generate a verifying output tuple (m'_i, π'_i) such that $m'_i \neq m_i$. By construction, we know that π' is a verifying threshold signature of weight $n - f$. In particular, there are at least $n - 2f$ honest parties that have signed m'_i. On the other hand, by assumption, $f + 1$ honest parties have abandoned the protocol and are already locked on m_i. But this is clearly a contradiction by quorum intersection. This shows the abandonability property of the protocol.

Liveness. We have to show that if no honest party abandons the protocol, then every honest party eventually outputs a proof and terminates. Since there are at least $n - f$ honest parties, each party will eventually receive $n-f$ signatures on its input message with tag “sig”. Then, each party will also eventually receive $n - f$ signatures on its input message with tag “lockSig” and thus be able to generate a proof π and terminate. This shows the liveness property of the protocol. □

Theorem 10. *The* NonEquiv *and* AbandonableNonEquiv *have a communication complexity of $O((\lambda+m)n^2)$, where m is the size of the input, and a round complexity of $O(1)$.*

Proof. Both protocols consist of a constant number of all-to-all rounds, sending messages of size $O(\lambda + m)$. In total, this results in a constant number of rounds, and $O((\lambda + m)n^2)$ communication. □

C.2 Provable AVID (Section 4.2)

Theorem 11. *The pair* (Disperse, DisperseVerify, Retrieve) *is a provable AVID protocol resilient to* f *Byzantine parties if* $n > 3f$.

Proof. Each property is proven separately.

Binding. We have to show that at the time the first honest party completes the Disperse protocol, there exists a unique committed value m'. By way of contradiction, assume that there exists a second committed value $m \neq m'$ after the dispersal phase (that a different honest party will reconstruct in the retrieval phase). By construction, we know that an honest party completes the dispersal phase only after seeing a threshold signature of weight $n - f$ on acc. In particular, by quorum intersection, there is at most one such accumulation value acc that comes with a verifying threshold signature π_{acc}. From this, it follows by collision-resistance of the underlying accumulator scheme Acc that there is at most one unique M' that has acc as an accumulation value (i.e., such that $\mathsf{acc} = \mathsf{Eval}_{\mathsf{Acc}}(ak, M')$). With that, it is clear that the decoding function Decode of the underlying erasure code yields uniqueness of committed value m'.

Termination. We have to show that if the sender is honest, then all honest parties will complete the Disperse protocol. Since there are at least $n-f$ honest parties, we know that the sender will be able to collect $n-f$ signatures on the accumulation value acc of its input message m and generate a proof π_{acc}. With that, it will then also be able to collect $n - f$ signatures for tag "share" from different parties. In particular, all honest parties will receive a verifying proof π_{acc} along with its share with corresponding witness (m_i, w_i) of the input message m. Then, it is also clear the the sender will be able to collect $n - f$ signatures on its index and generate a proof of dispersal π_{tot}. In particular, it will complete the Disperse protocol and output a proof π.

Provability. We have to show that if the sender is honest and it outputs a proof π, then DisperseVerify $(s, \pi) = 1$. But this is clear from the above termination considerations, since the proof is a threshold signature of weight $n - f$. As such, the underlying threshold signature verification algorithm $\mathsf{PVerify}_{\mathsf{TS}}$ verifies for π and so does $\mathsf{DisperseVerify}(s, \pi)$.

External Validity. We have to show that if the sender is honest, then the committed value is its input $m' = m$. Further, any honest party that outputs a value from the Retrieve protocol, outputs an externally valid value. The first part is clear by the binding property as shown above. For the second part, we know that the sender provides a proof π along with its input message m that shows that $\mathsf{valid}(m) = 1$, i.e., a proof of external validity. This proof is verified along with the accumulation value acc corresponding to m. As such, parties only accept acc if the proof π is valid. This ensures that if an honest party completes the dispersal phase, then the committed value m is externally valid.

Agreement. We have to show that any two honest parties that output a value from the Retrieve protocol, output the same value m' (defined in the binding property above). But this is clear from the underlying decoding function Decode, which tells us that any $f + 1$ shares yield the same reconstructed message m.

Totality. We have to show that if an honest party receives a proof π such that $\mathsf{DisperseVerify}(s, \pi) = 1$ and all honest parties call Retrieve, then they all complete Retrieve. So assuming an honest party receives a verifying proof π, we know that at least $n-f$ parties received their shares for a common accumulation value acc. In particular, there are $n-2f$ honest parties with valid shares corresponding to acc. Thus, if all honest parties call the Retrieve protocol, then any honest party will receive at least $n - 2f \geq f + 1$ valid shares from other honest parties. From that, it is clear that any honest party will be able to reconstruct m via the decoding function Decode with its set of at least $f + 1$ valid shares. This shows that all honest parties will complete the Retrieve protocol.

□

Theorem 12. *The* Disperse *protocol has a communication complexity of* $O(\lambda n + m)$*, with* m *being the size of the message, and a round complexity of* $O(1)$*. The* Retrieve *protocol has a communication complexity of* $O(\lambda n^2 + mn)$ *and a round complexity of* $O(1)$.

Proof. The Disperse protocol consists of a constant number of rounds, with the disperser sending messages of size $O(\lambda + \frac{m}{n})$ and of parties responding with messages of size $O(\lambda)$. In total, the protocol requires $O(\lambda n + m)$

communication and $O(1)$ rounds. The $\mathsf{Retrieve}$ protocol consists of a constant number of all-to-all rounds, with parties sending messages of size $O(\lambda + \frac{m}{n})$. In total, the protocol requires $O(\lambda n^2 + mn)$ communication and $O(1)$ rounds. □

D Proofs for Section 5

In this section, we present the proofs for our main building blocks in Section 5.

D.1 PVSS Exchange (Section 5.1)

Lemma 1. *If the adversary produces a pair* (trans, π) *such that* $\mathsf{Verify}(\mathsf{trans}, \pi) = 1$ *then, with high probability,* trans *includes at least one transcript that was generated by a party that was honest at the time of erasure.*

Proof. If trans, π verify, then trans includes transcripts from at least $(\frac{2}{3} - \epsilon)\lambda c$ parties. For every $i, j \in [n]$, let $X_{i,j}$ be a Bernoulli random variable which equals 1 if j is allowed to send i a message according to the VRF (i.e., $\mathsf{VRF}(\tilde{\mathsf{sk}}_j, i) = true, \pi$) and 0 otherwise. Before corrupting a party, or that party sending a message, the adversary does not know which parties are allowed to send messages to each other. Notably, parties erase their keys before sending messages, so the adversary cannot generate correct proofs with verifying signatures for any other message if it corrupts a party after it starts sending messages. This means that for every party j that the adversary corrupts, and for every party i, the probability that the adversary will be able to generate a PVSS transcript and send it to i is distributed according to $X_{i,j}$ and is independent of the rest of the X variables. Now, for every $i \in [n]$, let X_i be the number of PVSS transcripts that the adversary manages to generate and send to i with proof. We denote the set of corrupt parties by $\mathcal{C}$. Based on the previous discussion, $X_i \leq \sum_{j \in \mathcal{C}} X_{i,j}$. Therefore: $\mathbb{E}[X_i] \leq \mathbb{E}[\sum_{j \in \mathcal{C}} X_{i,j}] \leq f \cdot \frac{\lambda c}{n} < \frac{1}{3}\lambda c$. Next, we would like to bound the probability that for any $i \in [n]$, at least $(\frac{2}{3}\lambda c)$ corrupt parties are allowed to to send a PVSS transcript generated by them. Applying the Chernoff bound, assuming $\epsilon < \frac{1}{50}$:

$$\begin{aligned}
\Pr[X_i \geq (\frac{2}{3} - \epsilon)\lambda c] &= \Pr[X_i \geq (2 - 3\epsilon)(\frac{1}{3}\lambda c)] \\
&\leq \Pr[X_i \geq (1 + (1 - 3\epsilon))\mathbb{E}[X_i]] \\
&\leq e^{-\frac{(1-3\epsilon)^2}{3-3\epsilon} \cdot \frac{1}{3}\lambda c} \\
&= e^{-\lambda c \cdot \frac{(1-3\epsilon)^2}{9-9\epsilon}} \\
&\leq e^{-\lambda c \cdot \frac{1}{10}} \\
&\leq e^{-\lambda}.
\end{aligned}$$

From the union bound, the probability that this holds for any i is no greater than:

$$e^{-\lambda} \cdot n = e^{-\lambda + \log n} \leq 2^{-\lambda},$$

since $\lambda \gg \log n$ (otherwise, the adversary would have exponential computational power, as $\lambda \in \Theta(\log n)$ implies $n \in \Theta(2^\lambda)$). Thus, we obtain negligible failure probability. □

Lemma 2. *For any adversary strategy, with high probability, at least one forever-honest party eventually receives at least* $(\frac{2}{3} - \epsilon)\lambda c$ *"trans" messages from honest parties.*[18]

Proof. We will start by showing that for any choice of $f < \frac{1}{3}n$ parties to corrupt, there is a low probability that no honest party receives messages from $(\frac{2}{3} - \epsilon)\lambda c$ parties. Following that, we will apply the union bound

[18] In fact, our proof readily extends to show that the adversary can block at most $\Theta(n/\lambda)$ forever-honest parties. Consequently, a constant fraction of the forever-honest parties is guaranteed to receive sufficiently many messages.

to show that there is a low probability there is any possible set of f parties for which this event occurs. Clearly, if this event does not occur for any possible set of f parties, then regardless of which parties the adversary chooses to corrupt, at least one forever-honest party will eventually receive messages from $(\frac{2}{3}-\epsilon)\lambda c$ honest parties with high probability.

First, let I be some set of $n-f > \frac{2}{3}n$ parties. For every $i, j \in I$, let $X_{i,j}$ be a Bernoulli random variable which equals 1 if j is allowed to send i a message according to the VRF (i.e., $\mathsf{VRF}(\tilde{\mathsf{sk}}_j, i) = true, \pi$) and 0 otherwise. In addition, let $X_i = \sum_{j \in I} X_{i,j}$ be the number of parties in I that are allowed to send messages to i. Every party is allowed to send a message to i with probability $\frac{\lambda c}{n}$ and thus:

$$\mathbb{E}[X_i] = \mathbb{E}[\sum_{j \in I} X_{i,j}] > \frac{2}{3} n \frac{\lambda c}{n} = \frac{2}{3}\lambda c.$$

Bounding the probability that X_i is smaller than $(\frac{2}{3}-\epsilon)$, we get:

$$\begin{aligned}
\Pr[X_i < (\frac{2}{3}-\epsilon)\lambda c] &\leq \Pr[X_i \leq (1-\frac{3}{2}\epsilon)\mathbb{E}[X_i]] \\
&\leq e^{-(\frac{3}{2}\epsilon)^2 \cdot \frac{1}{2} \cdot \frac{2}{3}\lambda c} \\
&= e^{-\lambda c \cdot \frac{3}{4}\epsilon^2}
\end{aligned}$$

The probability that any i can send a message to any j is independent of any other such pair, and thus the probability the probability that X_i is small is independent of the probability that any other X_j is small. Therefore:

$$\Pr[\forall i \in I : \ X_i < (\frac{2}{3}-\epsilon)\lambda c] \leq e^{-n\lambda c \cdot \frac{3}{4}\epsilon^2}$$

The adversary can choose any set of f parties to corrupt. In total, the number of options the adversary has is $\binom{n}{f} \leq 2^n \leq e^n$ (the number of all possible subsets of $[n]$ is already 2^n). Applying the union bound over all options, we can bound the probability that there is some possible choice of parties to corrupt such that no honest party receives enough messages.

$$\begin{aligned}
&\Pr[\exists I \subseteq [n] \ s.t. \ (|I| = n-f) \ \wedge \ (\forall i \in I : \ X_i < (\frac{2}{3}-\epsilon)\lambda c)] \\
&\leq e^n \cdot e^{-n\lambda c \cdot \frac{3}{4}\epsilon^2} \\
&= e^{-n\lambda c \cdot \frac{3}{4}\epsilon^2 + n} \\
&= e^{-n(\lambda c \cdot \frac{3}{4}\epsilon^2 - 1)}.
\end{aligned}$$

Since c and ϵ are chosen constants, this is clearly negligible in λ. In other words, the probability that no honest party receives enough messages is $e^{-O(n\lambda)}$. □

Theorem 13. *The pair* $(\mathsf{PVSSExchange}, \mathsf{ExchangeVerify})$ *described in Algorithms 8 and 9 is a PVSS Exchange protocol resilient to f Byzantine parties if $n > 3f$.*

Proof. Each property is proven separately.

Verifiability. If some honest party outputs a pair (trans, π), then it correctly computed the proof π from trans and $\mathsf{transcripts}$ after receiving $(\frac{2}{3}-\epsilon)$ verifying "trans" messages. This means that $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$. In addition, from Lemma 1, if the adversary produces a pair (trans, π), then trans includes a transcript generated by an honest party before erasure. Even if the adversary corrupts that party after it sends its transcript, that party erases its contents before sending the message.

Termination. From Lemma 2, for any adversary strategy, at least one forever-honest party receives at least $(\frac{2}{3}-\epsilon)\lambda c$ messages from other honest parties. After receiving these messages, it sees that the messages and proofs are correct, performs local computations, and terminates. □

Theorem 14. *The* PVSSExchange *protocol has an expected communication complexity of* $O(\lambda^2 n^2)$ *and a round complexity of* $O(1)$.

Proof. The protocol consists of a single round, with each party sending a transcript to every other party with probability $\frac{\lambda c}{n}$. In expectation, every party sends $O(\lambda c)$ such messages, containing transcripts of size $O(\lambda n)$. In total, this means that the communication complexity is $O(\lambda^2 n^2)$, and the protocol takes a single round. □

D.2 Weighted Gather (Section 5.2)

Lemma 3. *By the time the first honest party completes the* WeightedGather *protocol, there exists a vector* $v \in \mathbb{N}_0^\ell$ *such that* $|v| \geq n-f$ *and the honest parties received* $\langle$ *"agg"*, $\mathsf{agg}_j, \pi_{\mathsf{agg},j}, \pi_{\mathsf{aggNE},j}\rangle$ *messages with verifying proofs from at least* $f+1$ *parties* j *such that* $\mathsf{agg}_j \geq v$.

Proof. Since $\mathsf{agg}_j, \pi_{\mathsf{agg},j}$ verify, there are $n-f$ vectors vec_k with verifying proofs such that $\mathsf{agg}_j \geq \mathsf{vec}_k$. Note that from the non-equivocation property of the NonEquiv protocol for every possible party k, there exists a single vector vec_k that can be included in agg_j vectors. At the time the first honest party completes the protocol, it received at least $n-f$ agg_j vectors. In total, the $n-f$ agg_j vectors each include $n-f$ vec_k vectors. Therefore, we can count a total of at least $(n-f)^2$ times vec_k vectors are included in agg_j vectors. Assume by way of contradiction, that all vec_k vectors are included in at most f agg_j vectors. This would mean that in total, there are at most nf times that vec_k vectors are included in agg_j vectors. Therefore:

$$\begin{aligned} & nf \geq (n-f)^2 = n^2 - 2nf + f^2 \\ \iff \quad & 0 \geq n^2 - 3nf + f^2 \end{aligned}$$

Note that $n > 3f$ and thus:

$$0 \geq n^2 - n(3f) + f^2 \geq n^2 - n^2 + f^2 = f^2 > 0.$$

Reaching a contradiction. This means that for some k, $v = \mathsf{vec}_k$ satisfies the condition that $\mathsf{agg}_j \geq v$ for $f+1$ different parties j. In addition, computing that vec_k can easily be done by counting the number of agg_j vectors such that $\mathsf{agg}_j \geq \mathsf{vec}_k$ for each $k \in [\ell]$ and choosing one for which the condition holds. □

Theorem 15. WeightedGather *is a Weighted Gather protocol resilient to* f *Byzantine parties if* $n > 3f$.

Proof. Each property is proven separately.

Binding Core. As shown in Lemma 3, there exists some v such that $|v| \geq n-f$ and for at least $f+1$ possible parties j, $\mathsf{agg}_j \geq v$. Since $\mathsf{out}_k, \pi_{\mathsf{out},k}$ verifies, there must be at least $n-f$ parties j with vectors agg_j such that $\mathsf{out}_k \geq \mathsf{agg}_j$. Out of those, at least one is one of the $f+1$ vectors for which $\mathsf{agg}_j \geq v$ and thus $\mathsf{out}_k \geq \mathsf{agg}_j \geq v$, as required.

Binding Cover. At the time the first honest party completes the protocol, it had already received "out" messages from $n-f$ parties, with at least $f+1$ of those coming from honest parties. Let I be the set of at least $f+1$ honest parties described above. When parties send the "out" messages, they abandon the AbandonableNonEquiv instances with the tag "val". From the Abandonability property of the AbandonableNonEquiv protocol, at that time, there exists a unique (x_j, π_j) for each $j \in [n]$ for which a proof $\pi_{\mathsf{valNE},j}$ might be produced. For every $i \in [\ell]$, let $u_i = |\{j | x_j = i \wedge \mathsf{valid}(x_j, \pi_j) = 1\}|$ and $u = (u_1, \ldots, u_\ell)$. If $x_j = \perp$ for any $j \in [n]$, then all honest parties in I didn't output any value for party j. Since proofs can only be produced for (x_j, π_j), every correct vector can only have one contribution in the index x_j per such pair such that $\mathsf{valid}(x_j, \pi_j) = 1$. In other words, $u \geq \mathsf{vec}$ for every correct vector vec. Following this, only

these values can be included in "vec" messages and subsequently in "agg" and "out" messages. In total, this means that $u \geq \mathsf{out}$ for any verifying out, π.

External Validity. Let u be the vector defined in the binding cover property, and let (x_j, π_j) be the values defined in the proof of the property. As defined above, $u[i] > 0$ for any $i \in [\ell]$ if there exists some $j \in [n]$ such that $x_j = i$ and $\mathsf{valid}(x_j, \pi_j) = 1$. In that case, for any $i \in [\ell]$ such that $u[i] > 0$, choose an arbitrary such proof π as the extracted proof, e.g., choose π to be π_j for the minimal index j such that $x_j = i$ and $\mathsf{valid}(x_j, \pi_j) = 1$.

Termination. We will start by showing that if some honest party i updates $\mathsf{out}_i, \pi_{\mathsf{out},i}$ to be non-$\perp$, all honest parties eventually complete the protocol. Party i updates these values after either computing them directly with a correct proof, or after checking that $\mathsf{Verify}(out_i, \pi_{\mathsf{out},i}) = 1$. It then sends these values to all parties in an "out" message, so every honest j will also update $\mathsf{out}_j, \pi_{\mathsf{out},i}$ to non-$\perp$ values after receiving the message if it hasn't already. After receiving such messages from $n - f$ parties, every party terminates.

Assume by way of contradiction that no honest party i ever has $\mathsf{out}_i \neq \perp$ and $\pi_{\mathsf{out},i} \neq \perp$. All parties start by calling the AbandonableNonEquiv protocol. Note that if some honest party i abandons the AbandonableNonEquiv protocol, it has $\mathsf{out}_i \neq \perp$ and $\pi_{\mathsf{out},i} \neq \perp$. As shown above, in that case all parties complete the protocol, reaching a contradiction. Therefore, no honest party abandons the protocol. From the Liveness property of AbandonableNonEquiv, every honest i eventually outputs a proof $\pi_{\mathsf{valNE},i}$ from the protocol. Following that, parties send "val" messages. Parties receive these messages, see that the proof verifies and that x_j, π_j are valid, and update their vals sets. After updating the vals sets $n - f$ times, every honest party calls NonEquiv with tag "vec" and outputs a verifying proof. Parties then send "vec" messages, accept messages sent by honest parties and similarly compute agg values and call NonEquiv with tag "agg". Finally, honest parties send "agg" messages, accept each other's messages, and compute $\mathsf{out}_i, \pi_{\mathsf{out},i}$, reaching a contradiction. Therefore, eventually some honest party has $\mathsf{out}_i \neq \perp, \pi_{\mathsf{out},i} \neq \perp$. As shown above, this means that eventually all parties complete the protocol. □

Theorem 16. *The* WeightedGather *protocol has a communication complexity of* $O((\lambda + \ell \log n)n^2)$ *and a round complexity of* $O(1)$.

Proof. The protocol has a constant number of calls to AbandonableNonEquiv and NonEquiv with inputs of size $\lambda + \log n$. In addition, parties send a constant number of messages of the same size, totalling in $O((\lambda + \ell \log n)n^2)$ communication and a constant number of rounds. □

D.3 Weak Coin (Section 5.3)

In the security proof, we consider $n = i_{\max} - i_{\min} + 1$, i.e., the exact number of parties participating in this call to the protocol. Further, we consider f to be the number of corrupt parties in the range $i_{\min}$ to $i_{\max}$.

Lemma 4. *If all honest parties complete output a vector from* WDCG *and call* Flip *with the same tag, then they all complete the call to* Flip.

Proof. If $\lfloor (i_{\max} - i_{\min})/\ell \rfloor \geq 4$, then parties complete the WDCG protocol after completing a call to WeightedGather. From the binding core property of the WeightedGather protocol, when the first honest party completes the call to the WeightedGather protocol there exists a vector $v \in \mathbb{N}_0^\ell$ such that $|v| \geq n - f$ and $v_i \geq v$ for every honest i. Similarly, from the binding cover property of the protocol, there exists a vector $u \in \mathbb{N}_0^\ell$ such that $|u| \leq n$ and $u \geq v_i$ for every honest i. From the external validity property of the protocol, for every $j \in [\ell]$ such that $u[j] > 0$, it is possible to extract a proof π_j such that $\mathsf{validWG}(j, \pi_j) = 1$. Since validWG returns 1, we know that $f+1$ parties sent "childProof" messages with signatures on $(k, \text{"childProof"})$. Honest parties send such a message after storing some values agreed and π_{child} in $\mathsf{outputs}[j]$ and $\mathsf{proofs}[j]$ and sending these values to all parties. Every party receives these messages and stores these values as well if it hasn't previously updated these variables. Note that every such π_{child} proves that $n - f$ parties signed $(j, \mathsf{agreed}, \text{"childSig"})$, and every honest party signs only one such message per $j \in [\ell]$. This means that only one such value agreed might receive such a proof because any two sets of $n - f$ parties have an intersection of $f + 1$ parties, with one of those being an honest party that sends only one such message. In other words,

all honest parties eventually update $\mathsf{outputs}[j]$ to be the same value agreed. In addition, an honest party only sends a "childSig" message on the value $\mathsf{agreed} = (\mathsf{disp}, \pi)$ after seeing that $\mathsf{Verify}(\mathsf{disp}, \pi) = 1$. This means that the party disp successfully dispersed a value. Following that, when honest parties call Flip, they eventually have $\mathsf{outputs}[k] = (\mathsf{disp}, \pi)$ for every k such that $u[k] > 0$. They then retrieve a transcript from the dispersal, store it in $\mathsf{retrieved}[k]$, and send "partial" messages to each other. After receiving verifying "partial" messages from $n - f$ parties, they finally aggregate the coin and store the value in $\mathsf{flips}[k]$. Note that $u \geq v_i$ for every honest party i, and thus every honest party eventually sees that $\mathsf{flips}[j] \neq \perp$ for every j such that $v_i[j] > 0$. At that point, they perform local computations and complete the protocol.

On the other hand, if $\lfloor (i_{\max} - i_{\min})/\ell \rfloor < 4$, then parties complete the protocol after completing the call to $\mathsf{BaseConsensus}$. In that case, they all output the same value agreed from the consensus, update $\mathsf{outputs}[1] \leftarrow \mathsf{agreed}$, and output $v_i = (1, 0, \ldots, 0)$. From the external validity of the $\mathsf{BaseConsensus}$ protocol, $\mathsf{agreed} = (\mathsf{disp}, \pi)$ such that $\mathsf{DisperseVerify}(\mathsf{disp}, \pi) = 1$. This means that disp successfully dispersed a correct transcript trans. Following the exact same reasoning as above, all parties eventually complete the Flip protocol. □

Theorem 17. $(\mathsf{WDCG}, \mathsf{Flip})$ *are a Weak Coin protocol resilient to f Byzantine parties if $n > 3f$, with $\alpha = \frac{2}{3}$.*

Proof. Each property is proven separately.

$\frac{2}{3}$-Uniformity. Assume all honest parties complete the WDCG protocol and call Flip with a given tag. If $\lfloor (i_{\max} - i_{\min})/\ell \rfloor \geq 4$, before completing the protocol, every party completes the call to $\mathsf{WeightedGather}$ and outputs a vector v_i. From Lemma 4, all honest parties complete their call to Flip. From the binding core property of the $\mathsf{WeightedGather}$ protocol, when the first honest party completes the $\mathsf{WeightedGather}$ protocol there exists a vector $v \in \mathbb{N}_0^\ell$ such that $|v| \geq n - f$ and $v_i \geq v$ for every honest i. Similarly, from the binding cover property of the protocol, there exists a vector $u \in \mathbb{N}_0^\ell$ such that $|u| \leq n$ and $u \geq v_i$ for every honest i. As shown in the proof of Lemma 4, all parties store the same correct transcripts in their $\mathsf{retrieved}$ arrays, and thus compute the same values in their flips array.

For every $j \in [\ell], k \in [u[j]]$, let r_j be the coin produced from the transcript in $\mathsf{retrieved}[j]$ with tag and let $r_{j,k} = \mathsf{H}(r_j, k, tag)$. Let the event E be the event that for $j^*, k^* = \mathsf{argmax}\{r_{j,k} | j \in [\ell], k \in [u[j]]\}$, $v[j^*] \geq k^*$. Note that for every $j \in [\ell], k \in [u[j]]$, the probability that $r_{j,k}$ is the maximal value[19] is $\frac{1}{|u|} \geq \frac{1}{n}$. Since $|v| \geq n - f$, the probability that E holds is $\frac{|v|}{|u|} \geq \frac{n-f}{n} > \frac{2}{3}$. If the event E occurs, then every honest party i will compute $r_{j,k}$ for every $j \in [\ell], k \in [v_i[j]]$, which includes j^*, k^* since $v_i \geq v$. All honest parties then see that r_{j^*,k^*} is maximal and output $\mathsf{H}(r_{j^*}, 0, tag)$, which is uniform and independent of the other values. In addition, before some honest party calls Flip with tag, no honest party computes $\mathsf{PartialCoin}$ with tag. In addition, from the external validity of the AVID protocol, any $(\mathsf{trans}, \pi_{\mathsf{Exchange}}), \pi_{\mathsf{Disperse}}$ such that $\mathsf{DisperseVerify}((\mathsf{trans}, \pi_{\mathsf{Exchange}}), \pi_{\mathsf{Disperse}}) = 1$ also have $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi_{\mathsf{Exchange}}) = 1$, meaning that trans includes at least one contribution from an honest party that is unknown to the adversary. Therefore, the adversary's view is independent of r_j and of the $r_{j,k}$ values, and thus of $\mathsf{H}(r_{j^*}, 0, tag)$.

On the other hand, if $\lfloor (i_{\max} - i_{\min})/\ell \rfloor < 4$, all parties complete the protocol with the same vector $v_i = (1, 0, \ldots, 0)$. In that case, they all flip the one coin stored in $\mathsf{outputs}[1]$ and output a uniformly sampled value with probability 1.

Termination. Lemma 4 shows that if all honest parties complete the call to WDCG and call Flip with the same tag, then they all complete the call as well. This means that showing that parties complete the WDCG protocol shows that the property holds. All honest parties start the protocol by calling $\mathsf{PVSSExchange}$. From the Termination and Correctness properties of $\mathsf{PVSSExchange}$, at least one forever-honest party i completes the protocol with a verifying $\mathsf{trans}, \pi_{\mathsf{Exchange}}$. Party i then calls the $\mathsf{Disperse}$ protocol and completes it with a valid proof π. At that point, i updates disp_i to i and $\pi_{\mathsf{disp},i}$ to π if it hasn't updated these variables previously, after verifying their correctness. Once it updates its $\mathsf{disp}_i, \pi_{\mathsf{disp},i}$ variables it sends these values to all parties in a "disp" message. Every honest party receives these messages and updates its corresponding $\mathsf{disp}, \pi_{\mathsf{disp}}$ variables if it hasn't already done so. Following that, every honest party calls a consensus protocol.

In the base case, all parties call the $\mathsf{BaseConsensus}$ protocol, eventually complete it, and complete the WDCG protocol. Otherwise, since $f < \frac{n}{3}$, at least one of the ℓ recursive calls to WDCG and to $\mathsf{RecConsensus}$

[19] We ignore the negligible probability that the maximal value is not unique.

has less than a $\frac{1}{3}$-fraction of corrupt parties. Let that call be the call of child i, i.e., the one for which every party j participating in the call has $\mathsf{child}[j] = i$. Therefore, the i'th call has all properties of a WDCG protocol and thus all honest parties complete it. Following that, all honest parties use a correct WDCG setup in the consensus protocol and thus all honest parties complete the consensus protocol with the same externally valid value $\mathsf{agreed} = (\mathsf{disp}, \pi)$ such that $\mathsf{DisperseVerify}(\mathsf{disp}, \pi) = 1$. Furthermore, the adversary cannot generate a verifying proof π_{child} for any other value $\mathsf{agreed}' \neq \mathsf{agreed}$ such that $\mathsf{Verify}(\mathsf{agreed}', \pi_{\mathsf{child}}, i) = 1$. Every honest party j such that $\mathsf{child}[j] = i$ will send a "childNE" message upon completing the protocol. Every honest party then receives that message, and responds with a "childSig". Note that no other verifying value can be produced for parties in the i'th recursive call, and thus honest parties will respond to the first such message with the value agreed. After receiving $n - f$ such signatures, every party j such that $\mathsf{child}[j] = i$ will update its $\mathsf{outputs}[i]$ and $\mathsf{proofs}[i]$ variables and send a "childProof" message, if it hasn't done so earlier. If some honest party sends a $\langle$"childProof", $k, \mathsf{agreed}, \pi, \sigma\rangle$ message, it does so with a verifying proof and signature. This means that every honest party eventually receives that message, updates its $\mathsf{outputs}[k], \mathsf{proofs}[k]$ values, adds a signature to proofSigs, and sends a "childProof" message as well. Eventually, every honest party will gather enough signatures for in $\mathsf{proofSigs}[i]$ (and possibly other entries in proofSigs), generate a proof that it saw at least $f + 1$ such signatures, and call WeightedGather if it hasn't done so earlier. Since the proof is honestly generated, every honest party calls the WeightedGather protocol with an externally valid input. From the Termination property of the WeightedGather protocol, all honest parties complete these calls and then complete the WDCG protocol, as required.

Note that in the above discussion, we assumed that parties run until completing the calls to BaseConsensus and RecConsensus. As discussed above, parties only run $\lambda c \cdot \log^{-1}(\frac{9}{7})$ views of each call. The probability that the RecConsensus call for the i-th child does not terminate in this number of rounds is at most $(\frac{7}{9})^{\lambda c \cdot \log^{-1}(\frac{9}{7})} = 2^{-\lambda c}$, and thus Termination holds with all but a negligible probability. □

Theorem 18. *The* WDCG *protocol has a communication complexity of* $O(\ell\lambda^2 n^2)$ *and a round complexity of* $O(\log_\ell n)$. *The* Flip *protocol has a communication complexity of* $O(\ell\lambda n^2)$ *and a round complexity of* $O(1)$.

Proof. Other than the recursive calls, the WDCG protocol consists of the following:

- a single call to the PVSSExchange protocol, costing $O(\lambda^2 n^2)$;
- $O(n)$ calls to Disperse with inputs of size $O(\lambda n)$, for a total of $O(\lambda n^2)$;
- either one call to BaseConsensus with a cost of $O(\lambda^2 n^3)$ (with n being the number of parties in the base call) or ℓ calls to RecConsensus, for a total cost of $O(\ell\lambda^2 n^2)$;
- ℓ calls to Retrieve, for a total cost of $O(\ell\lambda n^2)$;
- a single call to WeightedGather with a total cost of $O((\lambda + \ell \log n)n^2)$;
- and a constant number of all-to-all rounds with messages of size $O(\lambda+\ell)$, for a total cost of $O((\lambda+\ell)n^2)$.

In total, each call not in the base case costs $\ell\lambda^2 n^2$. Each call to the recursion generates $O(\ell)$ children, each with $O(\frac{n}{\ell})$ parties. In total, this means that the costs for all non-base calls are:

$$\sum_{i=0}^{\log_\ell n-1} \ell^i \cdot \ell\lambda^2(\frac{n}{\ell^i})^2 \leq \ell\lambda^2 n^2 \sum_{i=0}^{\log_\ell n-1} \frac{1}{\ell^i} = O(\ell\lambda^2 n^2)$$

In addition, in the base case there are $O(\frac{n}{\ell})$ leaves, each with $O(\ell)$ parties. In total, this means that the costs of the base case are:

$$\frac{n}{\ell} \cdot \lambda^2 \ell^3 = \lambda^2 n \ell^2 = O(\ell\lambda^2 n^2)$$

Note that all sub-protocols require a constant number of rounds, except the recursive calls and the calls to consensus. The consensus call for the honest child always terminates in a constant number of rounds in expectation. As shown in the proof of Termination, all parties complete the protocol a constant number of rounds after the honest child does, and thus the entire protocol other than the recursive calls requires a constant number of rounds in expectation. The depth of the recursion tree is $O(\log_\ell n)$, with a constant expected number of rounds in each layer, for a total of $O(\log_\ell n)$ expected rounds.

The Flip protocol consists of a constant number of all-to-all rounds, with messages of size $O(\ell(\lambda + \log n))$, for a total of $O((\lambda + \log n)n^2)$ communication and $O(1)$ rounds. □

D.4 Proposal Election from Weak Coin (Section 5.4)

Theorem 19. *The* PE *protocol is a Proposal Election protocol resilient to* f *Byzantine parties if* $n > 3f$*, with parameter* $\alpha = \frac{2}{9}$.

Proof. Each property is proven separately.

$\frac{2}{9}$-Binding. Assume some honest party completes the PE protocol. At that time, it had $n-f$ non-$\perp$ entries in its $\mathsf{elected}$ variable, which it updated after receiving "elected" messages with verifying contents from $n-f$ different parties. At least $n-f$ of those were sent by honest parties after completing the $\mathsf{Flip}(v_i, tag)$ protocol. Parties only complete the Flip protocol after receiving "ready" messages from $n-f$ parties. Let I be the set of all honest parties that send such messages at that time, and note that $|I| \geq n-f$. Honest parties send such a message after receiving "stored" messages from $n-f$ parties, with $f+1$ of those being sent by honest parties. Finally, honest parties only send such a "stored" message to j after seeing that receiving a $\langle$"proposal", $\mathsf{prop}_j, \pi_{\mathsf{propNE},j}, (\text{"proposal"}, tag)\rangle$ from j such that $\mathsf{Verify}(j, \mathsf{prop}_j, \pi_{\mathsf{propNE},j}, (\text{"proposal"}, tag)) = 1$ and storing $(\mathsf{prop}_j, \pi_{\mathsf{propNE},j})$ in $\mathsf{proposals}[j]$.

As shown in Theorem 17, the Flip protocol has the $\frac{2}{3}$-Uniformity property. This means that with probability $\frac{2}{3}$ or greater, all honest parties output the same uniformly sampled index $r \in [n]$ that is independent of the adversary's view before the first honest party calls Flip. In that case, every honest party sends the message $\langle$"elected", $r, \mathsf{proposals}[r], \sigma, tag\rangle$. If r happens to be some index in I, then at least honest $f+1$ parties stored the value $(\mathsf{prop}_r, \pi_{\mathsf{propNE},r})$ sent by r in $\mathsf{proposals}[r]$ and sent it in its "proposal" message. Let J be the set of honest parties that stored $(\mathsf{prop}_r, \pi_{\mathsf{propNE},r})$. This means that any set of $n-f$ "elected" messages must include one message sent by a party in J. Note that parties in J check that $\mathsf{Verify}(r, \mathsf{prop}_r, \pi_{\mathsf{propNE},r}, (\text{"proposal"}, tag)) = 1$ before storing the value, so the adversary will not be able to generate any other verifying $\mathsf{prop}'_r, \pi'_{\mathsf{propNE},r}$ such that $\mathsf{prop}_r \neq \mathsf{prop}'_r$.

Assume the adversary provides a pair $(\mathsf{out}, (r', \pi_{r'}, \pi_{\mathsf{out}}))$ such that $\mathsf{PEVerify}(\mathsf{out}, (r', \pi_{r'}, \pi_{\mathsf{out}}), tag) = 1$. First, $r' = r$ because any set of $n-f$ "proposal" messages contains at least $f+1$ messages sent by honest parties with the value r, and the honest parties sign no other values. This means that $\mathsf{Verify}(\perp, \pi_{r'}, tag)$ must equal 0, and so would $\mathsf{Verify}(r', \pi_{r'}, tag)$ if $r' \neq r$. Similarly, as any set of "proposal" messages sent by $n-f$ parties must include a message from one party in J, and no other verifying $\mathsf{prop}'_r, \pi'_{\mathsf{propNE},r}$ can be provided with $\mathsf{prop}'_r \neq \mathsf{prop}_r$, $\mathsf{Verify}(\mathsf{out}, (r', \pi_{r'}, \pi_{\mathsf{out}}), tag)$ would equal 0 for $\mathsf{out} = \perp$ or for any $\mathsf{out} \neq \mathsf{prop}_r$. By assumption, $\mathsf{Verify}(\mathsf{out}, (r', \pi_{r'}, \pi_{\mathsf{out}}), tag) = 1$, so $\mathsf{out} = \mathsf{prop}_r$. Finally, note that given the event E, the probability that all honest parties output the same index $r \in I$ is $\frac{|I|}{n} \geq \frac{f+1}{n} \geq \frac{1}{3}$. In total, if both events take place, then there exists some party i^* that was honest when sending its "ready" message (and thus also when starting the protocol) such that the adversary cannot provide a verifying out, π unless $\mathsf{out} = \mathsf{prop}_{i^*}$. This means that the property holds with $\alpha = \frac{2}{3} \cdot \frac{1}{3} = \frac{2}{9}$.

Completeness. When an honest party i outputs $(\mathsf{out}_i, (r^*, \pi_{r^*,i}, \pi_{\mathsf{out},i}))$, if $r^* = \perp$, it generates a correct proof $\pi_{r^*,i}$ and sets $\mathsf{out}_i = \perp$. In that case, $r^* = \mathsf{out}_i = \perp$ and $\mathsf{Verify}(\perp, \pi_{r^*,i}, tag) = 1$ and thus $\mathsf{PEVerify}$ outputs 1. If $r^* \neq \perp$, then i generates a correct proof $\pi_{r^*,i}$ and thus $\mathsf{Verify}(r^*, \pi_{r^*,i}, tag) = 1$. In addition, i generates a correct proof $\pi_{\mathsf{out},i}$ both when $\mathsf{out}_i = \perp$ and when $\mathsf{out}_i \neq \perp$, and thus $\mathsf{PEVerify}$ outputs 1.

External Validity. If $\mathsf{out} = \perp$, the claim trivially holds. Otherwise, $\mathsf{PEVerify}(\mathsf{out}, (r, \pi_r, \pi_{\mathsf{out}}), tag)$ only if $\mathsf{Verify}(r, \pi_r, tag) = \mathsf{Verify}(r, \mathsf{out}, \pi_{\mathsf{out}}, tag) = 1$. The fact that $\mathsf{Verify}(r, \mathsf{out}, \pi_{\mathsf{out}}, tag) = 1$ directly implies that $\mathsf{valid}(\mathsf{out}) = 1$.

Termination. If every honest i calls $\mathsf{PE}(\mathsf{prop}_i, tag)$, they all start by calling $\mathsf{WDCG}()$ and $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$. Every honest i completes both calls, updates v_i, and sends a "proposal" message. Honest parties then receive each other's "proposal" messages, see that the prop_j values are valid and that the $\pi_{\mathsf{propNE},j}$ proofs verify, update their $\mathsf{proposals}[j]$ value and respond with a "stored" message. After receiving $n-f$ such "stored" messages, every party sends a "ready" message to all parties. Finally, after receiving "ready" messages from $n-f$ parties, each honest party calls $\mathsf{Flip}(v_i, tag)$. Parties then send "elected" messages to each other. Finally, parties receive each other's "elected" messages and update their $\mathsf{elected}$ variables. After doing so for $n-f$ parties, parties perform local computations and complete the protocol. □

Note that as stated in the protocol, the WDCG protocol can be called once to set up many instances of PE, as each call to Flip can utilize the same WDCG setup. In fact, in the WDCG protocol, we recursively call WDCG once per child. Because of this, in the following theorem, we analyze the costs of the protocol without counting the costs of the WDCG. For the total costs, calling PE many times would cost an expected $O(\ell \cdot \lambda^2 n^2)$ for the single call to WDCG plus $O((\lambda\ell + m)n^2)$ per call to PE.

Theorem 20. *The* PE *protocol requires a single call to* WDCG *plus* $O((\lambda\ell+m)n^2)$ *communication and* $O(1)$ *rounds, where* m *is the size of the input.*

Proof. Other than the WDCG protocol, parties call the Flip protocol for a cost of $O(\ell\lambda n^2)$ and a constant number of rounds of sending messages of size $\lambda + m$. In total, this costs $O((\lambda\ell + m)n^2)$ communication. In addition, the Flip protocol terminates after a constant number of rounds, and other parts of the protocol complete after a constant number of rounds. □

E Proofs for Section 6

In this section, we present the proofs for our ADKG protocol in Section 6.

E.1 Security

Theorem 21. *The* ADKG *protocol described in Algorithm 18 is an Asynchronous Distributed Key Generation protocol resilient to* f *Byzantine parties if* $n > 3f$*, assuming hardness of the OMDL problem (cf. Definition 17).*

Proof. Each property is proven separately.

Correctness. If some honest party completes the protocol and outputs trans, it output (trans, π) from some call to Retrieve. From the external validity of the AVID protocol, $\mathsf{validDisperse}(\mathsf{trans}, \pi) = 1$, and thus $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$. From the Verifiability property of the PVSSExchange protocol, trans includes at least one contribution that erased its contents without being corrupted. This means that trans is uniform and sampled independently of the adversary's view. In addition, since $\mathsf{validDisperse}(\mathsf{trans}, \pi) = 1$, $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)) = 1$. That is, the output is a correct PVSS transcript, as required.

Termination. At least one honest party i completes the PVSSExchange protocol and calls the Disperse protocol. Following that, that party completes the disperse protocol and updates $\mathsf{disp}_i, \pi_{\mathsf{disp},i}$ if it hasn't done so before. Party i updates its $\mathsf{disp}_i, \pi_{\mathsf{disp},i}$ variables after generating the verifying proof $\pi_{\mathsf{disp},i}$ or receiving these values and verifying the proof. It then sends these values to all parties. All honest parties receive these values and update their own $\mathsf{disp}, \pi_{\mathsf{disp}}$ variables if they haven't done so before. After updating these variables, every honest party calls the WDCG protocol. From the Termination of WDCG, parties complete the protocol and call RecConsensus with an externally valid input and a WDCG setup. Parties eventually complete the call and output the same externally valid output $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$. Since the output is externally valid, $\mathsf{DisperseVerify}(\mathsf{agreed}, \pi_{\mathsf{agreed}}) = 1$, and thus $\mathsf{agreed} = (i, \pi)$ for some party i that dispersed a pair (trans, π) such that $\mathsf{validDisperse}(\mathsf{trans}, \pi) = 1$. Every honest party calls the Retrieve protocol, retrieves (trans, π) and outputs trans. In particular, all parties output the same public key pk and the same vector of public key shares $(\mathsf{tpk}_1, \ldots, \mathsf{tpk}_n)$.

Secrecy. We show secrecy of our DKG protocol, assuming aggregated unpredictability of the underlying aggregatable PVSS scheme APVSS. Concretely, we build a reduction against the aggregated unpredictability of APVSS. For this, we split our proof into two parts. First, we provide a simulation of the aggregated unpredictability experiment to an adversary $\mathcal{A}$ via a sequence of games. In particular, we interpolate between some games using reductions against the security of the consensus protocol $\mathsf{RecConsensus}$ and the security of the asynchronous verifiable dispersal protocol AVID. Second, we bound $\mathcal{A}$'s winning probability in the final game by providing an efficient reduction against the aggregated unpredictability of APVSS. We consider the following sequence of games with $\mathcal{A}$ as adversary. Throughout, we denote by $\mathcal{C} \subset [n]$ the set of corrupt parties and by $\mathcal{H} := [n] \setminus \mathcal{C}$ the set of honest parties.

Game $\mathbf{G}_0$: This is the real secrecy game. In particular, the game samples system parameters *par* and initializes a corruption set $\mathcal{C} := \emptyset$ and updates $\mathcal{H} := [n] \setminus \mathcal{C}$ throughout the game. Then, the game runs $\mathcal{A}$ on input *par* with access to a corruption oracle. Whenever $\mathcal{A}$ decides to corrupt a party $i \in \mathcal{H}$, the game honestly returns the internal state of that party i to $\mathcal{A}$ and updates $\mathcal{C} := \mathcal{C} \cup \{i\}$. From this point on, $\mathcal{A}$ gets complete control over party i. Further, all honest parties follow the protocol instructions for the DKG protocol. In particular, each party i honestly generates its PVSS transcripts for the protocol execution $\mathsf{PVSSExchange}((1, n))$. At the end of the protocol ADKG, each party i outputs a transcript trans and derives the public key pk, the vector of public key shares $(\mathsf{tpk}_1, \ldots, \mathsf{tpk}_n)$, and its secret key share tsk_i from trans. At the end of the game, $\mathcal{A}$ outputs a secret s^* and wins the game if $|\mathcal{C}| \leq f$ and $s^* = \mathsf{sk}$ (which is the secret key for pk). Clearly, $\mathcal{A}$'s advantage in winning the game is given by

$$\Pr[\mathbf{G}_0 \Rightarrow 1] = \varepsilon.$$

Game $\mathbf{G}_1$: This game is identical to the previous game, except that we add an abort condition. The idea of this hybrid is to rule out failure of the protocol AVID. Namely, whenever an instance of AVID fails to output the correct message trans or a verifying invalid proof π, the game aborts. Since there is a polynomial number of instances of AVID overall, we can bound the winning probability of this game by

$$|\Pr[\mathbf{G}_0 \Rightarrow 1] - \Pr[\mathbf{G}_1 \Rightarrow 1]| \leq \mathsf{poly}(n) \cdot \varepsilon_{\mathsf{AVID}}.$$

Here, we note that the probability of failure in AVID is directly given by the collision-resistance of the underlying cryptographic accumulator scheme and by the soundness of the underlying non-interactive proof system $\mathsf{PS}_{\mathsf{TS}} = (\mathsf{PProve}_{\mathsf{TS}}, \mathsf{PVerify}_{\mathsf{TS}})$ for relation $\mathcal{R}_{\mathsf{TS}}$.

Game $\mathbf{G}_2$: This game is identical to the previous game, except that we add another abort condition. The idea of this hybrid is to rule out failure of the consensus protocol $\mathsf{RecConsensus}$. Namely, whenever an instance of the consensus protocol $\mathsf{RecConsensus}$ fails to establish consensus, the game aborts. Assuming the security of the consensus protocol being given by $\varepsilon_{\mathsf{RecConsensus}}$, we can bound the winning probability of this game by

$$|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_2 \Rightarrow 1]| \leq \varepsilon_{\mathsf{RecConsensus}}.$$

Game $\mathbf{G}_3$: This game is identical to the previous game, except that we add another abort condition. So far we have rule out failure of the distributed protocols $\mathsf{RecConsensus}$ and AVID. From our termination and correctness considerations in the first part of our proof and the security of the $\mathsf{PVSSExchange}$ protocol, it then follows that the protocol establishes the same aggregated transcript trans for all parties that has contribution from at least one honest party. As such, the idea of this hybrid is to guess this special party $i^* \in [n]$ that contributes to the aggregated transcript trans and that remains honest until the end of the game. Concretely, at the beginning of the game, the game makes a random guess by sampling $\tilde{i} \leftarrow\!\!\$\ [n]$ and executes the game as in $\mathbf{G}_2$. At the end of the game, the game aborts if $\tilde{i} \neq i^*$ or $i^* \notin \mathcal{H}$ (i.e., we gues the wrong special party or the special party gets corrupted). Since the choice of $\tilde{i}$ remains information-theoretically hidden from $\mathcal{A}$'s view, we can bound the winning probability of this game by by

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \geq \Pr[\mathbf{G}_2 \Rightarrow 1]/(3n).$$

We note that the factor n comes from the condition $\tilde{i} \neq i^*$, while the factor 3 comes from the condition $i^* \notin \mathcal{H}$. It remains to bound the probability that the final game $\mathbf{G}_3$ outputs 1. For that, we build an efficient reduction $\mathcal{R}$ against the aggregated unpredictability of APVSS. The design is straightforward.

Building a reduction. We build an efficient reduction $\mathcal{R}$ to the aggregated unpredictability game of APVSS. For this, we first recall the unpredictability game. At a high level, it captures malleability attacks and prohibits any adversary (corrupting at most f parties) from learning the secret of an aggregated transcript that has contribution from at least one honest party. In more detail, there are two kinds of queries we can make: (i) corruption queries, and (ii) PVSS transcript queries. For the second point (ii), we specify a party index $i \in \mathcal{H}$ and obtain a PVSS transcript on behalf of that party i from the game. At the end of the game, we have to output a PVSS transcript trans^* along with a secret s^* and win the game if we have made at most f corruption queries, trans^* is a valid PVSS transcript with contribution from at least one honest party $i^* \in \mathcal{H}$ and s^* is the secret encoded in the transcript trans^*. Having said that, building a reduction should be immediate.

At the beginning of the aggregated unpredictability experiment, $\mathcal{R}$ submits a PVSS transcript request for party index $i^* \in \mathcal{H}$ and obtains a PVSS transcript trans_{i^*}. Then it simulates the game $\mathbf{G}_3$ to $\mathcal{A}$ by honestly generating PVSS transcripts in the $\mathsf{PVSSExchange}$ protocol execution for all parties $i \in \mathcal{H} \setminus \{i^*\}$. For the special party i^*, however, it uses trans_{i^*} that will be included in trans (other PVSS transcripts of i^* will also be generated honestly). Whenever $\mathcal{A}$ decides to corrupt a party $i \in \mathcal{H}$, the reduction $\mathcal{R}$ simply forwards this corruption query to its own challenger in its aggregated unpredictability experiment and returns the output to the adversary $\mathcal{A}$. In this way, $\mathcal{R}$ can correctly answer all corruption queries of $\mathcal{A}$. At the end of the simulation to $\mathcal{A}$, the adversary $\mathcal{A}$ outputs a secret s^* to $\mathcal{R}$. We assume that the adversary outputs a correct forgery s^*, so that s^* is the secret of the final aggregated transcript trans which has contribution from i^*. Now, the reduction $\mathcal{R}$ outputs the tuple (trans, s^*) to the challenger of its underlying aggregated unpredictability experiment. In particular, the winning conditions are satisfied: (i) $|\mathcal{C}| \leq f$ is clear, since the same holds true for the adversary $\mathcal{A}$. (ii) $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n) = 1$ is clear, since the DKG protocol execution succeeded by assumption. (iii) The existence of an index $i \in \mathcal{H}$ such that $i \in \mathsf{Contr}(\mathsf{trans}) \subseteq [n]$ is clear, since i^* is this index. Then, it follows that we can bound the winning probability of this final

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \leq \varepsilon_{\mathsf{APVSS}}.$$

Finally, the work [BL23] shows that aggregated unpredictability of APVSS is reduced to the one-more discrete logarithm (OMDL) assumption of degree n. With that, we conclude our proof.

□

F Pseudocode Figures

F.1 Non-Equivocation Rounds (Section 4.1)

Algorithm 1 $\mathsf{NonEquiv}(m_i, tag)$

$\mathsf{sigs}_i \leftarrow \emptyset$
send $\langle$"help"$, m_i, tag\rangle$ to all parties
// Sign received messages
upon receiving the first $\langle$"help"$, m_i, tag\rangle$ message from j, **do**
    send $\langle$"sig"$, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (j, m_j, tag))\rangle$ to j
// Collect signatures, output a proof once $n - f$ have been received
upon receiving the first $\langle$"sig"$, \sigma_j\rangle$ message from j, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (i, m_i, tag), \sigma_j) = 1$ **then**
        $\mathsf{sigs}_i \leftarrow \mathsf{sigs}_i \cup \{(j, \sigma_j)\}$
        if $|\mathsf{sigs}_i| = n - f$ **then**
            output $\mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m_i, tag); \mathsf{sigs}_i)$, but continue sending messages

Algorithm 2 $\mathsf{NEVerify}(i, m, \pi, tag)$

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m, (\text{"lockSig"}, tag)); \pi)$

Algorithm 3 $\mathsf{AbandonableNonEquiv}(m_i, tag)$

$\mathsf{sigs}_i \leftarrow \emptyset,\ \mathsf{locksigs}_i \leftarrow \emptyset$
send $\langle$"help"$, m_i, tag\rangle$ to all parties
// Non-equivocation of inputs
upon receiving the first $\langle$"help"$, m_i, tag\rangle$ message from j, **do**
    send $\langle$"sig"$, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (j, m_j, (\text{"sig"}, tag)))\rangle$ to j
upon receiving the first $\langle$"sig"$, \sigma_j\rangle$ message from j, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (i, m_i, (\text{"sig"}, tag))) = 1$ **then**
        $\mathsf{sigs}_i \leftarrow \mathsf{sigs}_i \cup \{(j, \sigma_j)\}$
        if $|\mathsf{sigs}_i| = n - f$ **then**
            send $\langle$"lock"$, m_i, \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m_i, (\text{"sig"}, tag)); \mathsf{sigs}_i)\rangle$ to all parties
// Provide proof that non-equivocation proof has been heard before abandoning
upon receiving the first $\langle$"lock"$, m_j, \pi_j\rangle$ message from j, **do**
    if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (j, m_j, (\text{"sig"}, tag)); \pi_j) = 1$ **then**
        send $\langle$"lockSig"$, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (j, m_j, (\text{"lockSig"}, tag)))\rangle$ to j
upon receiving the first $\langle$"lockSig"$, \sigma_j\rangle$ message from j, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (i, m_i, (\text{"lockSig"}, tag))) = 1$ **then**
        $\mathsf{locksigs}_i \leftarrow \mathsf{locksigs}_i \cup \{(j, \sigma_j)\}$
        if $|\mathsf{locksigs}_i| = n - f$ **then**
            output $\mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m_i, (\text{"lockSig"}, tag)); \mathsf{locksigs}_i)$, but continue sending messages
upon receiving an abandon signal, **do**
    terminate and stop sending messages in the AbandonableNonEquiv protocol with tag tag

Algorithm 4 $\mathsf{ANEVerify}(i, m, \pi, tag)$

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m, (\text{"lockSig"}, tag)); \pi)$

F.2 Provable AVID (Section 4.2)

Algorithm 5 $\mathsf{Disperse}(m)$

1: $\mathsf{sigs} \leftarrow \emptyset,\ \mathsf{sigs}' \leftarrow \emptyset,\ \pi_{tot} \leftarrow \bot,\ M' \leftarrow \bot$
2: $m_i \leftarrow \bot,\ w_i \leftarrow \bot,\ \mathsf{acc}_i \leftarrow \bot,\ \pi_{acc,i} \leftarrow \bot$
// Lines 3-7 are only run by the sender s
// Encode the message in an error-correcting code and compute an accumulator of the encoding
3: $M' \leftarrow (m'_1, \ldots, m'_n) \leftarrow \mathsf{Encode}(m; n, f+1)$
4: $\mathsf{acc} \leftarrow \mathsf{Eval}_{\mathsf{Acc}}(ak, M')$
5: **for** $i \in [n]$ **do**
6: $\quad w_i \leftarrow \mathsf{Witness}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, M')$
// Non-equivocation of accumulator
7: send $\langle$"help", $\mathsf{acc}, \pi\rangle$ to all parties
8: **upon** receiving the first $\langle$"help", $\mathsf{acc}\rangle$ message from the sender s, **do**
9: $\quad$ send $\langle$"sig", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (s, \mathsf{acc}))\rangle$ to s
10: **upon** the sender s receiving the first $\langle$"sig", $\sigma_i\rangle$ message from i, **do**
11: $\quad$ **if** $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (s, \mathsf{acc})) = 1$ **then**
12: $\quad\quad \mathsf{sigs} \leftarrow \mathsf{sigs} \cup \{(i, \sigma_i)\}$
13: $\quad\quad$ **if** $|\mathsf{sigs}| = n - f$ **then**
14: $\quad\quad\quad \pi_{\mathsf{TS}} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}); \mathsf{sigs})$
15: $\quad\quad\quad \pi_{acc,s} \leftarrow \mathsf{PProve}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}); m, \pi_{\mathsf{TS}})$
// Send shares of the encoding to each party, parties reply once they receive their shares
16: **upon** the sender s having $\pi_{acc,s} \neq \bot$, **do**
17: $\quad$ **for** $i \in [n]$ **do**
18: $\quad\quad$ send $\langle$"share", $\mathsf{acc}_s, \pi_{acc,s}, m', w\rangle$ to i
19: **upon** receiving the first $\langle$"share", $\mathsf{acc}, \pi_{acc}, m'_i, w_i\rangle$ message from j, **do**
20: $\quad$ **if** $\mathsf{PVerify}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); \pi_{acc}) = 1$ and $\mathsf{Verify}_{\mathsf{Acc}}(ak, \mathsf{acc}, m', w) = 1$ **then**
21: $\quad\quad m'_i \leftarrow m',\ w'_i \leftarrow w,\ \mathsf{acc}_i \leftarrow \mathsf{acc},\ \pi_{\mathsf{acc},i} \leftarrow \pi_{\mathsf{acc}}$
22: $\quad\quad$ send $\langle$"total", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, j)\rangle$ to j
// Wait until $n - f$ parties received their shares, and output a proof
23: **upon** the sender s receiving the first $\langle$"total", $\sigma_i\rangle$ message from i, **do**
24: $\quad$ **if** $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, i) = 1$ **then**
25: $\quad\quad \mathsf{sigs}' \leftarrow \mathsf{sigs}' \cup \{(i, \sigma_i)\}$
26: $\quad\quad$ **if** $|\mathsf{sigs}'| = n - f$ **then**
27: $\quad\quad\quad \pi_{tot} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, i; \mathsf{sigs}')$
28: $\quad\quad\quad$ **output** π_{tot} and **terminate**

Algorithm 6 $\mathsf{Retrieve}()$

$\mathsf{shares} \leftarrow \emptyset,\ m \leftarrow \perp$
send $\langle$"retrieve", $\mathsf{acc}_i, \pi_{\mathsf{acc},i}, m'_i, w_i\rangle$ to all parties
upon receiving the first $\langle$"retrieve", $\mathsf{acc}, \pi_{\mathsf{acc}}, m'_i, w_i\rangle$ message, **do**
    if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}_i); \pi_{\mathsf{acc},i}) = 1$ and $\mathsf{Verify}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, w_i) = 1$ **then**
        $\mathsf{shares} \leftarrow \mathsf{shares} \cup \{(i, m'_i)\}$
        if $|\mathsf{shares}| = f + 1$ **then**
            $m \leftarrow \mathsf{Decode}(\mathsf{shares}; n, f + 1)$
            output m

Algorithm 7 $\mathsf{DisperseVerify}(i, \pi)$

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, \text{"total"}); \pi)$

F.3 PVSS Exchange (Section 5.1)

Algorithm 8 PVSSExchange(tag)

messages $\leftarrow \emptyset$, transcripts $\leftarrow \emptyset$
// Check which parties to send messages to, generate transcripts, and store them
for $j \in [n]$ **do**
  $(\mathsf{sendFlag}_j, \pi_j) \leftarrow \mathsf{VRF}(\mathsf{sk}_i, (j, tag))$
  if $\mathsf{sendFlag}_j = true$ **then**
    $\mathsf{trans}_j \leftarrow\!\!\$\ \mathsf{Dist}_{\mathsf{PVSS}}(\vec{\mathsf{pk}})$
    messages $\leftarrow$ messages $\cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \mathsf{Sign}_{\mathsf{KDS}}(\mathsf{sk}_i, (j, \mathsf{trans}_j, tag)))\}$
// Erase keys and then send messages to all parties
update $\bar{\mathsf{sk}}_i$ via Update and **erase** the old key and randomness used for $\mathsf{Dist}_{\mathsf{PVSS}}()$
for all $(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j) \in$ messages **do**
  send $\langle$"trans", $\mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j\rangle$ to j
// Collect transcripts and output an aggregated transcript after receiving enough of them
upon receiving a $\langle$"trans", $\mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j\rangle$ message from j, **do**
  if $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}_j, \mathsf{sendFlag}, (i, tag), \pi_j) = \mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}_j, (i, \mathsf{trans}_j, tag), \sigma_j) = 1$ and $\mathsf{sendFlag} = true$ **then**
    transcripts $\leftarrow$ transcripts $\cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j)\}$
    if $|\mathsf{transcripts}| \geq (\frac{2}{3} - \epsilon)\lambda c$ **then**
      $\mathsf{trans} \leftarrow \mathsf{Aggregate}_{\mathsf{PVSS}}(\{\mathsf{trans}_j \mid \exists (j, \mathsf{sendFlag}_j, \pi_j, \mathsf{trans}_j, \sigma_j) \in \mathsf{transcripts}\})$
      $\pi \leftarrow \mathsf{PProve}_{\mathsf{exch}}(\vec{\mathsf{pk}}, \mathsf{trans}, tag; \mathsf{transcripts})$
      output (trans, π) and **terminate**

Algorithm 9 ExchangeVerify(trans, π, tag)

return $\mathsf{PVerify}_{\mathsf{exch}}(\vec{\mathsf{pk}}, \mathsf{trans}, tag; \pi)$

F.4 Weighted Gather (Section 5.2)

Algorithm 10 WeightedGather(x_i, π_i)

$\mathsf{vec}_i \leftarrow \bot,\ \pi_{\mathsf{vec},i} \leftarrow \bot,\ \mathsf{agg}_i \leftarrow \bot,\ \pi_{\mathsf{agg},i} \leftarrow \bot,\ \mathsf{out}_i \leftarrow \bot,\ \pi_{\mathsf{out},i} \leftarrow \bot$
// Non-equivocation of inputs
$\mathsf{vals}_i \leftarrow \emptyset,\ \mathsf{vecs}_i \leftarrow \emptyset,\ \mathsf{aggs}_i \leftarrow \emptyset$
call AbandonableNonEquiv$((x_i, \pi_i)$, "val")
upon AbandonableNonEquiv$((x_i, \pi_i)$, "val") outputting $\pi_{\mathsf{valANE},i}$, **do**
    send $\langle$"val", $x_i, \pi_i, \pi_{\mathsf{valANE},i}\rangle$ to all parties
// Collecting inputs in a vector, non-equivocation of vector
upon receiving the first $\langle$"val", $x_j, \pi_j, \pi_{\mathsf{valANE},j}\rangle$ message from j, **do**
    if ANEVerify$(j, (x_j, \pi_j), \pi_{\mathsf{valANE},j}$, "val") $= 1$, $\mathsf{valid}(x_j, \pi_j) = 1$ and $x_j \in [\ell]$ **then**
        $\mathsf{vals}_i \leftarrow \mathsf{vals}_i \cup \{(j, x_j, \pi_j, \pi_{\mathsf{valANE},j})\}$
        if $|\mathsf{vals}_i| = n - f$ **then**
            for all $i \in [\ell]$, let $v_i = |\{j \in [n] \mid \exists \pi, \pi_{\mathsf{valANE}}$ s.t. $(j, i, \pi, \pi_{\mathsf{valANE}}) \in \mathsf{vals}_i\}|$
            $\mathsf{vec}_i \leftarrow (v_1, \ldots, v_\ell),\ \pi_{\mathsf{vec},i} \leftarrow \mathsf{PProve}_{\mathsf{cnt}}(\vec{\mathsf{pk}}, \mathsf{vec}_i, (i)_{i\in[\ell]}; \mathsf{vals}_i)$
            call NonEquiv(vec_i, "vec")
upon NonEquiv(vec_i, "vec") outputting $\pi_{\mathsf{vecNE},i}$, **do**
    send $\langle$"vec", $\mathsf{vec}_i, \pi_{\mathsf{vec},i}, \pi_{\mathsf{vecNE},i}\rangle$ to all parties
// Collecting vectors in an aggregated vector, non-equivocation of aggregated vector
upon receiving the first $\langle$"vec", $\mathsf{vec}_j, \pi_{\mathsf{vec},j}, \pi_{\mathsf{vecNE},j}\rangle$ message from j, **do**
    if NEVerify$(j, \mathsf{vec}_j, \pi_{\mathsf{vecNE},j}$, "vec") $= 1$ and $\mathsf{PVerify}_{\mathsf{cnt}}(\vec{\mathsf{pk}}, \mathsf{vec}_j, (i)_{i\in[\ell]}; \pi_{\mathsf{vec},j}) = 1$ **then**
        $\mathsf{vecs}_i \leftarrow \mathsf{vecs}_i \cup \{(j, \mathsf{vec}_j, \pi_{\mathsf{vec},j}, \pi_{\mathsf{vecNE},j})\}$
        if $|\mathsf{vecs}_i| = n - f$ **then**
            $\mathsf{agg}_i \leftarrow \max\{\mathsf{vec}_j \mid \exists j, \pi_{\mathsf{vec}}, \pi_{\mathsf{vecNE}}$ s.t. $(j, \mathsf{vec}_j, \pi_{\mathsf{vec}}, \pi_{\mathsf{vecNE}}) \in \mathsf{vecs}_i\}$
            $\pi_{\mathsf{agg},i} \leftarrow \mathsf{PProve}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{agg}_i, (i)_{i\in[\ell]}; \mathsf{vecs}_i)$
            call NonEquiv(agg_i, "agg")
upon NonEquiv(agg_i, "agg") outputting $\pi_{\mathsf{aggNE},i}$, **do**
    send $\langle$"agg", $\mathsf{agg}_i, \pi_{\mathsf{agg},i}, \pi_{\mathsf{aggNE},i}\rangle$ to all parties
// Collecting aggregated vectors for output, send output in order to guarantee availability
upon receiving the first $\langle$"agg", $\mathsf{agg}_j, \pi_{\mathsf{agg},j}, \pi_{\mathsf{aggNE},j}\rangle$ message from j, **do**
    if NEVerify$(j, \mathsf{agg}_j, \pi_{\mathsf{aggNE},j}$, "agg") $= 1$ and $\mathsf{PVerify}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{agg}_j, (i)_{i\in[\ell]}; \pi_{\mathsf{agg},j}) = 1$ **then**
        $\mathsf{aggs}_i \leftarrow \mathsf{aggs}_i \cup \{(j, \mathsf{agg}_j, \pi_{\mathsf{agg},j}, \pi_{\mathsf{aggNE},j})\}$
        if $|\mathsf{aggs}_i| = n - f$ and $\mathsf{out}_i = \bot$ **then**
            $\mathsf{out}_i \leftarrow \max\{\mathsf{agg}_j \mid \exists j, \pi_{\mathsf{agg}}, \pi_{\mathsf{aggNE}}$ s.t. $(j, \mathsf{agg}_j, \pi_{\mathsf{agg}}, \pi_{\mathsf{aggNE}}) \in \mathsf{aggs}_i\}$
            $\pi_{\mathsf{out},i} \leftarrow \mathsf{PProve}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{out}_i, (i)_{i\in[\ell]}; \mathsf{aggs}_i)$
upon receiving the first $\langle$"out", $\mathsf{out}_j, \pi_{\mathsf{out},j}\rangle$ message from j, **do**
    if $\mathsf{PVerify}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{out}_j, (i)_{i\in[\ell]}; \pi_{\mathsf{out},j}) = 1$ and $\mathsf{out}_i = \bot$ **then**
        $\mathsf{out}_i \leftarrow \mathsf{out}_j,\ \pi_{\mathsf{out},i} \leftarrow \pi_{\mathsf{out},j}$
// Forward output, abandon AbandonableNonEquiv *to block new entries*
upon $\mathsf{out}_i \neq \bot$ and $\pi_{\mathsf{out},i} \neq \bot$, **do**
    send $\langle$"out", $\mathsf{out}_i, \pi_{\mathsf{out},i}\rangle$ to all parties
    abandon the AbandonableNonEquiv session with tag "val"
upon receiving "out" messages from $n - f$ parties, **do**
    output out_i and **terminate**

F.5 Weak Coin (Section 5.3)

Algorithm 11 WDCG$(i_{\min}, i_{\max})$

if $i < i_{\min}$ or $i > i_{\max}$ **then**
  terminate without participating in the protocol
child $\leftarrow$ partition$(i_{\min}, i_{\max}, \ell)$, $\mathsf{trans}_i, \mathsf{agreed}_i, \pi_{\mathsf{trans},i}, \pi_{\mathsf{agreed},i}, \pi_{\mathsf{child},i} \leftarrow \perp, \mathsf{sigs}_i \leftarrow \emptyset$
$\forall j \in [\ell]:$ helped$[j] \leftarrow false$, outputs$[j]$, proofs$[j]$, retrieved$[j] \leftarrow \perp$, proofSigs$[j] \leftarrow \emptyset$
// Get aggregated PVSS, disperse and forward
call PVSSExchange$((i_{i_{\min}}, i_{i_{\max}}))$
upon PVSSExchange$((i_{i_{\min}}, i_{i_{\max}}))$ terminating with output trans, π, **do**
  call Disperse$((\mathsf{trans}, \pi))$ with external validity validDisperse (Algorithm 13)
upon Disperse() terminating with output π, **do**
  if $\mathsf{disp}_i = \perp$ **then**
    $\mathsf{disp}_i \leftarrow i$, $\pi_{\mathsf{disp},i} \leftarrow \pi$
upon receiving the first $\langle$"disp", $\mathsf{disp}_j, \pi_{\mathsf{disp},j}\rangle$ from j, **do**
  if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (\mathsf{disp}_j, tag); \pi_{\mathsf{disp},j}) = 1$ and $\mathsf{disp}_i = \perp$ **then**
    $\mathsf{disp}_i \leftarrow \mathsf{disp}_j$, $\pi_{\mathsf{disp},i} \leftarrow \pi_{\mathsf{disp},j}$
// Recursively generate WDCG for consensus calls, call consensus for each child
upon $\mathsf{disp}_i \neq \perp$ and $\pi_{\mathsf{disp},i} \neq \perp$, **do**
  send $\langle$"disp", $\mathsf{disp}_i, \pi_{\mathsf{disp},i}\rangle$ to all parties
  if $\lfloor (i_{\max} - i_{\min})/\ell \rfloor < 4$ **then**
    call BaseConsensus$((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), i_{\min}, i_{\max})$ with external validity function DisperseVerify
  else
    $i'_{\min} \leftarrow \min\{j \mid \mathsf{child}[j] = \mathsf{child}[i]\}$, $i'_{\max} \leftarrow \max\{j \mid \mathsf{child}[j] = \mathsf{child}[i]\}$
    call WDCG$(i'_{\min}, i'_{\max})$
    upon WDCG$(i'_{\min}, i'_{\max})$ terminating with output v_i, **do**
      call RecConsensus$((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), i'_{\min}, i'_{\max})$ with external validity function DisperseVerify, using v_i and the state of WDCG$(i'_{\min}, i'_{\max})$ for the PE protocol
upon BaseConsensus outputting $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$, **do**
  output $v_i = (1, 0, \ldots, 0)$, update outputs$[1] \leftarrow$ agreed, and terminate
upon RecConsensus outputting $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$, **do**
  $\mathsf{agreed}_i \leftarrow \mathsf{agreed}$, $\pi_{\mathsf{agreed},i} \leftarrow \pi_{\mathsf{agreed}}$
// ChildNonEquiv runs a non-equivocation procedure per child on agreed
// When seeing non-equivocation proofs, parties send signatures, stored in proofSigs
run ChildNonEquiv() (Algorithm 12)
// After seeing that a unique output is sent to all parties, call WeightedGather
upon $|\mathsf{proofSigs}[k]| = f + 1$ for any $k \in [\ell]$, **do**
  if WeightedGather has not been called yet **then**
    call WeightedGather$(k, \mathsf{Prove}(k, \mathsf{proofSigs}[k]))$ with external validity function validWG (Algorithm 14)
upon WeightedGather terminating with output v_i, **do**
  output v_i, but continue updating the state according to the protocol
// Retrieve dispersed outputs and store for future use
upon outputs$[k] \neq \perp$ for any $k \in [\ell]$ and WeightedGather terminating, **do**
  call Retrieve() for retrieving the value dispersed by disp, where outputs$[k] = (\mathsf{disp}, \pi)$
upon Retrieve() outputting $(\mathsf{trans}_j, \pi_j)$ for the value dispersed by j, **do**
  retrieved$[j] \leftarrow \mathsf{trans}_j$

Algorithm 12 ChildNonEquiv()

upon $\mathsf{agreed}_i \neq \perp$, **do**
  send $\langle$"childNE", $\mathsf{agreed}_i, \pi_{\mathsf{agreed},i}\rangle$ to all parties
// Check proofs of correct outputs, only reply to each child once
upon receiving the first $\langle$"childNE", $\mathsf{agreed}_j, \pi_{\mathsf{agreed},j}\rangle$ message from j, **do**
  if $\mathsf{agreed}_j = (\mathsf{disp}, \pi)$ such that $\mathsf{DisperseVerify}(\mathsf{disp}, \pi) = 1$ **then**
    $i'_{\min} \leftarrow \min\{k \mid \mathsf{child}[k] = \mathsf{child}[j]\}$, $i'_{\max} \leftarrow \max\{k \mid \mathsf{child}[k] = \mathsf{child}[j]\}$
    if $\mathsf{ConsensusVerify}(\mathsf{agreed}_j, \pi_{\mathsf{agreed},j}, i'_{\min}, i'_{\max}) = 1$, and $\mathsf{helped}[\mathsf{child}[j]] = false$ **then**
      send $\langle$"childSig", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (\mathsf{child}[j], \mathsf{agreed}_j, \text{"childSig"}))\rangle$ to all k s.t. $\mathsf{child}[k] = \mathsf{child}[j]$
      $\mathsf{helped}[\mathsf{child}[j]] \leftarrow true$
// Collect signatures, send message with proof after receiving enough
upon receiving the first $\langle$"childSig", $\sigma_j\rangle$ message from j and having $\mathsf{agreed}_i \neq \perp$, **do**
  if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (\mathsf{child}[i], \mathsf{agreed}_i, \text{"childSig"}), \sigma_j) = 1$ **then**
    $\mathsf{sigs}_i \leftarrow \mathsf{sigs}_i \cup \{(j, \sigma_j)\}$
    if $|\mathsf{sigs}_i| = n - f$ and $\mathsf{outputs}[\mathsf{child}[i]] = \perp$ **then**
      $\mathsf{outputs}[\mathsf{child}[i]] \leftarrow \mathsf{agreed}_i$, $\mathsf{proofs}[\mathsf{child}[i]] \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (\mathsf{child}[i], \mathsf{agreed}_i, \text{"childSig"}); \mathsf{sigs}_i)$
upon $\mathsf{outputs}[j] \neq \perp$ for any $j \in [\ell]$, **do**
  send $\langle$"childProof", $j, \mathsf{outputs}[j], \mathsf{proofs}[j], \mathsf{Sign}(\mathsf{sk}_i, (j, \text{"childProof"}))\rangle$ to all parties
// Store any output with a proof, forward output for availability
upon receiving a $\langle$"childProof", $k, \mathsf{agreed}, \pi_{\mathsf{child}}, \sigma\rangle$ message from j, **do**
  if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (k, \mathsf{agreed}, \text{"childSig"}); \pi_{\mathsf{child}}) = \mathsf{Verify}(\mathsf{pk}_j, (k, \text{"childProof"}), \sigma) = 1$ **then**
    if $\mathsf{outputs}[k] = \perp$ **then**
      $\mathsf{outputs}[k] \leftarrow \mathsf{agreed}$, $\mathsf{proofs}[k] \leftarrow \pi_{\mathsf{child}}$
    $\mathsf{proofSigs}[k] \leftarrow \mathsf{proofSigs}[k] \cup \{(j, \sigma)\}$

Algorithm 13 validDisperse((trans, π))

if $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, \vec{\mathsf{pk}}) = 1$ and $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$ **then**
  return 1
else
  return 0

Algorithm 14 validWG(i, π_i)

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, \text{"childProof"}); \pi_i)$

Algorithm 15 $\mathsf{Flip}(v_i, tag)$

$\forall j \in [n]:\ \mathsf{flips}[j] \leftarrow \perp,\ \mathsf{partials}[j] \leftarrow \emptyset$
// Participate in flipping all coins
upon $\mathsf{retrieved}[j] \neq \perp$ for any j, **do**
  send $\langle$"partial", $\mathsf{PartialCoin}(\mathsf{sk}_i, j, \mathsf{retrieved}[j]), j, tag\rangle$ to all parties
upon receiving the first $\langle$"partial", $\mathsf{pc}_j, k\rangle$ message from j for any k, **do**
  if $\mathsf{VerifyPartial}(\mathsf{pk}_j, j, \mathsf{pc}_j, \mathsf{retrieved}[k]) = 1$ **then**
    $\mathsf{partials}[k] \leftarrow \mathsf{partials}[k] \cup \{(j, \mathsf{pc}_j)\}$
    if $|\mathsf{partials}[k]| = n - f$ **then**
      $\mathsf{flips}[k] \leftarrow \mathsf{AggregateCoin}(\mathsf{partials}[k])$
// Derive output from flips
upon $\mathsf{flips}[j] \neq \perp$ for every j such that $v_i[j] > 0$, **do**
  let $r_{j,k} = \mathsf{H}(r_j, k, tag)$ for every $j \in [\ell]$, $k \in [v_i[j]]$
  let $(j^*, k^*) = \mathsf{argmax}\{r_{j,k} \mid j \in [\ell], k \in [v_i[j]]\}$
  output $\mathsf{H}(r_j, 0, tag)$

F.6 Proposal Election from Weak Coin (Section 5.4)

Algorithm 16 $\mathsf{PE}(\mathsf{prop}_i, tag)$

$\forall j \in [n]:\ \mathsf{proposals}[j] \leftarrow \perp,\ \mathsf{elected}[j] \leftarrow \perp$
$v_i \leftarrow \perp,\ r^* \leftarrow \perp,\ \pi_{r^*,i} \leftarrow \perp,\ \mathsf{out}_i \leftarrow \perp,\ \pi_{\mathsf{out},i} \leftarrow \perp$
// Lines 3-5 can be called once to set up many instances of PE
call $\mathsf{WDCG}()$
upon WDCG outputting v, **do**
    $v_i \leftarrow v$
// Non-equivocation of inputs, store proposals for future use
call $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$
upon $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$ terminating with output π_{propNE}, **do**
    send $\langle$"proposal", $\mathsf{prop}_i, \pi_{\mathsf{propNE}}, tag\rangle$ to all parties
upon receiving the first $\langle$"proposal", $\mathsf{prop}_j, \pi_{\mathsf{propNE},j}, (\text{"proposal"}, tag)\rangle$ message from j, **do**
    if $\mathsf{valid}(\mathsf{prop}_j) = 1$ and $\mathsf{NEVerify}(j, \mathsf{prop}_j, \pi_{\mathsf{propNE},j}, (\text{"proposal"}, tag)) = 1$ **then**
        $\mathsf{proposals}[j] \leftarrow (\mathsf{prop}_j, \pi_{\mathsf{propNE},j})$
        send $\langle$"stored", $tag\rangle$ to j
// Wait until many parties stored proposal, then flip coin
upon receiving $\langle$"stored", $tag\rangle$ messages from $n - f$ parties, **do**
    send $\langle$"ready", $tag\rangle$ messages to all parties
upon receiving $\langle$"ready", $tag\rangle$ messages from $n - f$ parties and $v_i \neq \perp$, **do**
    call $\mathsf{Flip}(v_i, tag)$
// Inform each other of coin value, and of the stored proposal for the winning party
upon $\mathsf{Flip}(v_i, tag)$ terminating with output r_i, **do**
    send $\langle$"elected", $r_i, \mathsf{proposals}[r_i], \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (r_i, \mathsf{proposals}[r_i], tag), tag)\rangle$ to all parties
// Check whether one party won, and whether its input is stored somewhere
upon receiving the first $\langle$"elected", $r_j, \mathsf{val}_j, \sigma_j, tag\rangle$ message from j, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (r_j, \mathsf{val}_j, tag), \sigma_j) = 1$ **then**
        $\mathsf{elected}[j] \leftarrow (r_j, \mathsf{val}_j, \sigma_j)$
        if $\mathsf{elected}[j] \neq \perp$ for exactly $n - f$ indices $j \in [n]$ **then**
            if $\exists r \in [n]$ such that $\mathsf{elected}[j] = (r, \mathsf{val}_j, \sigma_j)$ for at least $f + 1$ indices $j \in [n]$ **then**
                $r^* \leftarrow r,\ \pi_{r^*,i} \leftarrow \mathsf{PProve}_{\mathsf{elect}}(\vec{\mathsf{pk}}, (r, tag); \mathsf{elected})$ ▷ choose an r arbitrarily if more than one
                if $\exists j \in [n]$ such that $\mathsf{elected}[j] = (r, \mathsf{val}_j, \sigma_j)$, $\mathsf{val}_j = (\mathsf{prop}, \pi) \neq \perp$, $\mathsf{valid}(\mathsf{prop}) = 1$,
                and $\mathsf{NEVerify}(r, \mathsf{prop}, \pi, (\text{"proposal"}, tag)) = 1$ **then**
                    $\mathsf{out}_i \leftarrow \mathsf{prop},\ \pi_{\mathsf{out},i} \leftarrow \pi$
                else
                    $\mathsf{out}_i \leftarrow \perp,\ \pi_{\mathsf{out},i} \leftarrow \mathsf{PProve}_{\mathsf{bad_win}}(r^*, \perp, \mathsf{elected}, tag)$
            else
                $r^* \leftarrow \perp,\ \pi_{r^*,i} \leftarrow \mathsf{PProve}_{\mathsf{no_win}}(\perp, \mathsf{elected}, tag)$
                $\mathsf{out}_i \leftarrow \perp,\ \pi_{\mathsf{out},i} \leftarrow \perp$
            output $(\mathsf{out}_i, (r^*, \pi_{r^*,i}, \pi_{\mathsf{out},i}))$ and **terminate**

Algorithm 17 $\mathsf{PEVerify}(\mathsf{out}, (r, \pi_r, \pi_{\mathsf{out}}), tag)$

if $r = \mathsf{out} = \perp$ and $\mathsf{PVerify}_{\mathsf{no_win}}(\perp, \pi_r, tag) = 1$ **then**
    return 1
else if $\mathsf{PVerify}_{\mathsf{elect}}(r, \pi_r, tag) = 1$ **then**
    if $\mathsf{out} \neq \perp$, $\mathsf{valid}(\mathsf{out}) = 1$, and $\mathsf{NEVerify}(r, \mathsf{out}, \pi, (\text{“proposal”}, tag)) = 1$ **then**
        return 1
    else if $\mathsf{out} = \perp$ and $\mathsf{PVerify}_{\mathsf{bad_win}}(r, \perp, \pi_{\mathsf{out}}, tag) = 1$ **then**
        return 1
return 0

F.7 Asynchronous DKG (Section 6)

Algorithm 18 ADKG()

$\mathsf{disp}_i \leftarrow \perp,\ \pi_{\mathsf{disp},i} \leftarrow \perp$
call PVSSExchange$((1,n))$
// Generate a candidate transcript, disperse it, and inform all parties about it being dispersed
upon PVSSExchange$((1,n))$ terminating with output (trans, π), **do**
    call Disperse$((\mathsf{trans}, \pi))$ with external validity validDisperse (Algorithm 19)
upon Disperse terminating with output π, **do**
    if $\mathsf{disp}_i = \perp$ **then**
        $\mathsf{disp}_i \leftarrow i,\ \pi_{\mathsf{disp},i} \leftarrow \pi$
upon receiving the first $\langle$"disp", $\mathsf{disp}_j, \pi_{\mathsf{disp},j}\rangle$ message from j, **do**
    if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (\mathsf{disp}_j, tag); \pi_{\mathsf{disp},j}) = 1$ and $\mathsf{disp}_i = \perp$ **then**
        $\mathsf{disp}_i \leftarrow \mathsf{disp}_j,\ \pi_{\mathsf{disp},i} \leftarrow \pi_{\mathsf{disp},j}$
// Agree on a dispersed transcript
upon $\mathsf{disp}_i \neq \perp$ and $\pi_{\mathsf{disp},i} \neq \perp$, **do**
    send $\langle$"disp", $\mathsf{disp}_i, \pi_{\mathsf{disp},i}\rangle$ to all parties
    call WDCG$(1,n)$
upon the WDCG$(1,n)$ protocol terminating with output v_i, **do**
    call RecConsensus$((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), 1, n)$ with external validity function DisperseVerify, using v_i and the state of WDCG$(i'_{\min}, i'_{\max})$ for the PE protocol
// Retrieve the agreed upon dispersed transcript and output it
upon RecConsensus outputting $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$, **do**
    call Retrieve() for retrieving the value dispersed by disp, where $\mathsf{agreed} = (\mathsf{disp}, \pi)$
upon Retrieve() terminating with output (trans, π), **do**
    output trans and **terminate**

Algorithm 19 validDisperse$((\mathsf{trans}, \pi))$

if $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, \vec{\mathsf{pk}}) = 1$ and $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$ **then**
    return 1
else
    return 0