Nearly Quadratic Asynchronous Distributed Key Generation

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

[1]Intel Labs
[2]CISPA Helmholtz Center for Information Security
[3]Tel Aviv University

Abstract

We prove that for any $1 \le k \le \log n$, given a VRF setup and assuming secure erasures, there exists a protocol for Asynchronous Distributed Key Generation (ADKG) that is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties. With all but negligible probability, all nonfaulty parties terminate in an expected $O(k)$ rounds and send a total expected $\tilde{O}(n^{2+1/k})$ messages.

1 Introduction

Many efficient distributed applications require the use of a distributed key generation (DKG) protocol. However, the best known *asynchronous* distributed key generation (ADKG) protocol, that is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties, requires cubic communication [AJM+23]. Recent ADKG implementations, proven resilient to static adversaries only, also require cubic communication [ZDL+23, DXKKR23], significantly limiting their scalability.

In a recent major advance, [BLL+24, FLT24] show subcubic *synchronous* distributed key generation protocols. A key paradigm used in these works and [FMT24] is a recursive structure (in some cases just one level of recursion) that can be seen as following the classic recursive paradigm of [CW92, BGP92].

We study asynchronous distributed key generation protocols with nearly quadratic expected communication. Conceptually, our work adopts the recursive paradigm to the asynchronous model.

Our main result is that for any $1 \le k \le \log n$, security parameters λ, c, given a VRF setup and assuming secure erasures, random oracles, and one-more discrete logarithm, there exists a protocol for (high-threshold) Asynchronous Distributed Key Generation that is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties. Except for a $O(2^{-\lambda \log^c n})$ error probability, all nonfaulty parties terminate in an expected $O(k)$ rounds and send a total expected $\tilde{O}(n^{2+1/k})$ messages.

For $k = \log n$, this is $O(\log n)$ expected time and $\tilde{O}(n^2)$ communication. For any fixed $0 < \epsilon < 1$, this is $O(1/\epsilon)$ expected time and $\tilde{O}(n^{2+\epsilon})$ communication.

1.1 High-level overview

Recursive partition. We partition the parties recursively, splitting the set into $\ell = n^{1/k}$ smaller sets of roughly the same size at each step. This hierarchical partition forms a tree T of depth k and degree ℓ. For a set S that is a vertex of T, let $p(S)$ be its parent set.

The protocol for S runs a validated asynchronous Byzantine agreement (VABA) protocol on each child. The output of this agreement is an aggregated PVSS for S and the randomness used for this agreement will come recursively from its children.

Input generation via PVSS aggregation. Since the output for the agreement on a child C is an aggregated PVSS for its parent $S = p(C)$, the protocol for S first constructs validated inputs for the agreement. That is, parties start by collecting aggregated PVSS transcripts for S that contain (with high probability) an unpredictable secret. We do this by having each party send a PVSS to a random set of polylog other parties (using a VRF to make sure this set is indeed near uniform). Parties erase their PVSS secret before sending it out.

If a party receives sufficiently many PVSS transcripts, it aggregates them to a validated input for the agreement protocol. Parties can prove that their inputs are valid, so they forward these inputs to each other to make sure that all parties have inputs for the next stages of the protocol. Moreover, we show that with high probability, this aggregated PVSS contains a transcript sent by a nonfaulty party that was generated and erased without that party being corrupted. Hence, the aggregated PVSS is unpredictable to the adversary even if it corrupts that party following that point.

Base case. To begin the recursion, we start with the small base sets of size $O(\ell)$ (leaf vertices of T). On each such set S, we run any constant expected round and $O(|S|^3) = O(n^{3/k})$ expected communication VABA protocol. So, a total of $n^{1+2/k}$ for the base layer. The input for this VABA protocol on a base set S is a validated aggregated PVSS for $p(S)$. Hence the output is some validated aggregated PVSS for $p(S)$.

Recursion. In the recursive step, for a non-leaf set $S \in V(T)$, we assume that each child set $C \in V(T)$ runs a VABA that outputs a validated aggregated PVSS for $p(C) = S$.

There are several challenges:

1. Even if the fraction of nonfaulty parties in S is $2/3$, we are only guaranteed that this holds for one child node. This means that members of S can only wait for at most *one* child VABA to output a valid value. Because the other children may not have enough nonfaulty parties to guarantee liveness.

2. Unfortunately, a child node that has less than a $2/3$ fraction of nonfaulty parties might complete its VABA early (this child may have no nonfaulty parties at all). There are three notable concerns for this '*malicious child*' case:

   (a) The first is that the VABA from this malicious child may output an *invalid* aggregated PVSS for S, i.e. with no nonfaulty contributions. We solve this by defining a *valid* output of a child VABA to include a proof that the PVSS is valid and indeed contains $2/3-\epsilon$ fraction of randomly chosen parties from S where these random choices can also be verified. This is also why parties aggregate their transcripts in S and not in any of the child nodes because we don't know which of the child nodes might have a dishonest majority.

   (b) The second is that a malicious child node may generate several outputs from its child VABA. We solve this by running a *non-equivocation protocol* by the set S to guarantee there is at most one such valid output for S (assuming that S has less than a $1/3$ fraction of faulty parties).

   (c) The third is that a malicious child may cause its nonfaulty parties to send too many messages. We overcome this by limiting the number of views in each child VABA at the cost of a negligible probability of non-termination.

Assuming that S has a $2/3$ fraction of nonfaulty parties, each nonfaulty member will eventually see some aggregated PVSS for S from the (at most one) output of some child VABA that passeed the non-equivocation protocol. Each such aggregated PVSS is valid and even though S may have a linear number of parties, there are at most $n^{1/k}$ possible aggregated PVSSs to choose from.

NWH variant and proposal election protocol. The VABA protocol that we use is a (new) variation of No-Waitin-Hotstuff [AJM+21] that we call Succinct-Knowledge-NWH which uses SNARKS to reduce the communication complexity to $\tilde{O}(|S|^2)$ per view. SK-NWH runs a proposal election protocol in each view and this proposal election needs a weak coin to obtain liveness.

Weak coin protocol The proposal election protocol allows parties to commit to proposals using a *provable AVID* protocol and then uses a weak coin protocol to determine which of the proposals to output. Unlike previous proposal election protocols, it also deals with the case that the weak common coin may choose parties that did not commit any input. The crux of our proposal election protocol is a new weak coin protocol.

The weak coin protocol is built from a *setup* protocol we call *weak distributed coin generation* (WDCG) that can be followed by any number of invocations of a weak coin flip protocol. Each instance of the weak coin flip protocol has a communication complexity of $\tilde{O}(|S|^2)$.

The core of this paper is a recursive protocol for WDCG. For a set S, the WDCG protocol first generates inputs for each child and recursively sets up the consensus for each child by calling a WDCG protocol with fewer parties. Parties then call a VABA protocol (SK-NWH) that uses the output of this recursive WDCG protocol for its randomness, finally outputting an aggregated transcript. The output of each child VABA is used as input to a non-equivocation protocol and then each party can choose one aggregated transcript, from a set of at most $n^{1/k}$ options (at most one option per child).

Weak distributed coin generation protocol and weak coin flip protocol. The goal of a WDCG is to enable a *weak coin flip* protocol, that with constant probability, outputs a uniformly random validated index in $1, \dots, |S|$ (and with the remaining probability, parties can disagree).

Given that each party chooses one transcript from a set of at most $n^{1/k}$ options, the goal is to build a *weak distributed coin generation (WDCG)* for S. The challenge is to build such a WDCG for a set S at a cost of just $\tilde{O}(|S|^2 n^{1/k})$ communication without being able to reach agreement in S.

Weighted binding cover gather protocol. The crux of the weak distributed coin generation protocol is a new binding cover gather protocol that we call *weighted gather*. In order to pay just $\tilde{O}(|S|^2 n^{1/k})$ communication for the weighted binding cover gather protocol, we use the fact that there are only $n^{1/k}$ possible aggregated PVSS transcripts to choose from and *compress* the choice of potentially n parties, to a weighted vector of length $n^{1/k}$ where each weight is at most $\log n$ bits (recording the number of parties that chose it, instead of their identities).

Each member of the weighted gather protocol implicitly commits to *two* unique coins that are induced from the PVSS it committed to, one coin is used to flip a *random rank*, and the other one is used to flip a *random coin value*. The output of the weak distributed coin generation protocol is a weighted vector of aggregated PVSS transcripts. The guarantee is that each such weighted vector is a subset of a large *core* weighted vector.

After outputting a vector, parties can use each PVSS transcript to flip several coins to generate ranks. The weights determine how many times each coin is flipped, generating more ranks for coins with larger values in the vector. Finally, the PVSS with the highest rank is chosen and is used once more to flip a single coin, which is the output from the protocol. With constant probability, the largest rank is associated with a PVSS that will be flipped by all parties because they all flip any coin determined by the core. If that happens, they all choose the same transcript to generate the final coin and agree on the output.

Total costs. The weak distributed coin generation protocol runs in $\tilde{O}(|S|^2 n^{1/k})$ complexity. The NWH and the weak coin flip run in $O(n^2)$ complexity per view, and the weak coin flip has constant success probability per view when the set S contains less than a $1/3$ fraction of faulty parties. After a constant expected number of rounds and $O(|S|^2 n^{1/k})$ expected messages, for such a set S, the VABA protocol on S will complete with a valid aggregated PVSS for $p(S)$. Moreover, even if it has more than a $1/3$ fraction of faulty parties, we limit the number of views so the total cost is $\tilde{O}(|S|^2 n^{1/k})$ communication complexity. Summing up, each level requires $O(1)$ expected time and $\tilde{O}(n^{2+1/k})$ communication. Since there are $O(k)$ levels, from linearity, the total expected time is $O(k)$ and the total expected communication is $\tilde{O}(n^{2+1/k})$.

1.2 Future work and open questions

We strongly believe that similar techniques can be adopted for optimally resilient *synchronous* DKG, getting $O(k)$ expected time and $\tilde{O}(n^{2+1/k})$ expected communication. Another interesting question is removing the assumption of a VRF setup and the ability of secure erasures. Finally, for $k = \log n$, can DKG protocols obtain $\tilde{O}(n^2)$ communication in $o(\log n)$ time, or is there an inherent lower bound?

2 Preliminaries and Model

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

General Notation. Let λ denote the computational security parameter. Throughout the paper, we assume that global parameters $par := (\mathbb{G}, \mathbb{G}_T, p, g, h, e)$, implicitly parameterized by λ, are fixed and known to all parties. Here, $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a symmetric pairing of prime order p cyclic groups with two independent generators $g, h \in \mathbb{G}$. 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 $[\![b]\!]$. For a finite set S, we write $x \leftarrow_\$ S$ to denote that x is sampled from S uniformly at random. We use $\mathbb{N}$ to denote the set of positive integers, and $\mathbb{N}_0$ the set of non-negative integers. We use the acronym PPT for probabilistic polynomial-time. We measure the communication complexity of our distributed protocols in bits.

Vector Notation. Let $v = (v_1, \ldots, v_k)$ be a vector in $\mathbb{N}_0^k$. We define $v[i] := v_i$ for every $i \in [k]$. In addition, 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 define $|v| := \sum_{i=1}^{k} v[i]$.

Adversarial and Network Model. We consider a complete network of n parties $\mathcal{P} := \{1, \ldots, n\}$ connected by pairwise private and authenticated channels. We consider an *asynchronous* network, where any message can be delayed arbitrarily under the constraint that messages sent between nonfaulty parties must eventually be delivered. We assume a Byzantine adversary who can corrupt up to $f < n/3$ parties maliciously and may cause them to deviate from the protocol arbitrarily. Further, the adversary is *strongly adaptive*: i.e., 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 nonfaulty. We may also refer to the nonfaulty parties as *honest* and to the faulty parties as *corrupt* or *malicious*.

Public Key Infrastructure. As common in this line of work on distributed cryptographic protocols, we assume that parties have established a bulletin board *public key infrastructure* (PKI) before the protocol execution. Concretely, we assume that every party i has a public-secret key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for a digital signature scheme $\mathsf{DS} = (\mathsf{KGen}_{\mathsf{DS}}, \mathsf{Sign}_{\mathsf{DS}}, \mathsf{Verify}_{\mathsf{DS}})$ and an encryption-decryption key pair $(\mathsf{ek}_i, \mathsf{dk}_i)$ for a public key encryption scheme, where pk_i and ek_i are known to all parties but sk_i and dk_i are known only to i. For this, we assume that 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. Further, we assume a trusted setup for generating key pairs $(\tilde{\mathsf{pk}}, \tilde{\mathsf{sk}}_i)$ of parties for a verifiable random function (VRF). In particular, this thwarts common grinding attacks.

Idealized Models. We assume the *random oracle model* (ROM) [BR93]. In this model, a hash function H is treated as an idealized random function to which the adversary gets oracle access. Further, we assume the *algebraic group model* (AGM) [FKL18]. Here, all algorithms are treated as algebraic (over the group $\mathbb{G}$): whenever an algorithm $\mathcal{A}$ outputs a group element $\zeta \in \mathbb{G}$, it additionally outputs a vector $\mathbf{z} = (z_1, \ldots, z_k)$ of integers such that $\zeta = \prod_{i \in [k]} g_i^{z_i}$, where $(g_1, \ldots, g_k) \in \mathbb{G}^k$ is the list of group elements $\mathcal{A}$ has received so far (either as input or as oracle responses).

Computational Assumptions. We rely on the *one-more discrete logarithm (OMDL) assumption* [BNPS03] for our security proofs. Throughout the paper, 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 base g.

For an algorithm $\mathcal{A}$ and $k \in \mathbb{N}$, we consider the following 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 $(\xi_1, \ldots, \xi_k)$. Here, $\mathcal{A}$ gets access to the 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 one-more discrete logarithm assumption of degree k holds relative to $(\mathbb{G}, p, g, h)$ if for any PPT algorithm $\mathcal{A}$, the probability that the above experiment outputs 1 is negligible in λ.

2.1 Cryptographic and Consensus Primitives

In this section, we introduce the cryptographic and consensus primitives used in the paper. For formal definitions of syntax and security, we refer to Appendix A.

Aggregatable Publicly Verifiable Secret Sharing. In a verifiable secret sharing (VSS) scheme, 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 [GJM+21].

Distributed Coin Flip from APVSS. The work of [GJM+21] demonstrates how to use an aggregated PVSS transcript to derive a non-interactive distributed coin flipping protocol. At a high level, the APVSS transcript can be used as a threshold key setup for a unique threshold signature similar to BLS, which then gives a coin via a random oracle on the threshold signature. In more detail, each party i derives a partial signature on some message identifier using its secret decryption key dk_i, done via the $\mathsf{PartialCoin}$ algorithm and verified via the $\mathsf{VerifyPartial}$ algorithm on input ek_i. Then, any $n-f$ valid partial signatures can then be combined into a full signature via the $\mathsf{AggregateCoin}$ algorithm. Finally, using a random oracle the common coin can be derived from that message identifier.

Error Correcting 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). In our protocols, we use Reed-Solomon codes with varying parameters (q, b). Specifically, q corresponds to the number of parties in a designated subset $\mathcal{Q} \subset \mathcal{P} := \{1, \ldots, n\}$, and $b := \lceil q/3 \rceil$. In the special case $\mathcal{Q} = \mathcal{P}$, the parameters are $(q, b) = (n, f+1)$, where n is the total number of parties and f the corruption threshold. This choice ensures robust error correction 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. A common example of a cryptographic accumulator is a Merkle tree. Here, the tree's root serves as the accumulation value, while authentication paths act as membership proofs for individual leaves.

Verifiable Random Function. A verifiable random function (VRF) 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 $\tilde{\mathsf{sk}}$, a verifiable random function VRF produces a pseudorandom string $u = \mathsf{VRF}(\tilde{\mathsf{sk}}, m)$ along with a proof π. Using the proof π and the public key $\tilde{\mathsf{pk}}$ (corresponding to $\tilde{\mathsf{sk}}$), anyone can verify that u was computed correctly without revealing any information about $\tilde{\mathsf{sk}}$. Looking ahead, we will make use of VRFs for randomized local committee election, and we note that this use does not require a global source of randomness. However, we assume a trusted setup for generating the VRF key pairs of parties to thwart known grinding attacks.

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. And 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 readability, we will omit the random oracle associated to PS in our descriptions from now on.

3 Building Blocks

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

3.1 Non-Equivocation Rounds

We construct two protocols for non-equivocation, $\mathsf{NonEquiv}$ and $\mathsf{AbandonableNonEquiv}$. 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 $\mathsf{NEVerify}$ in the case of the $\mathsf{NonEquiv}$ protocol and $\mathsf{ANEVerify}$ in the case of the $\mathsf{AbandonableNonEquiv}$ protocol. The $\mathsf{AbandonableNonEquiv}$ protocol has the additional property that parties can "abandon" the round and cause it to halt. By that we mean that after $f+1$ nonfaulty 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 $\mathsf{NonEquiv}$ protocol is similar to the provable broadcast primitive of [AMS18], and the $\mathsf{AbandonableNonEquiv}$ is similar to the locked broadcast primitive described in [AS22]. Looking ahead, we use the abandonability property in order to get the cover property for our weighted gather protocol.

3.1.1 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 nonfaulty parties call the protocol with the same tag:*

- ***Validity.*** *If a nonfaulty 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 nonfaulty 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 parties call the protocol with the same tag:*

- ***Validity.*** *If a nonfaulty 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 $f+1$ nonfaulty 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 nonfaulty party eventually outputs a proof, or some nonfaulty party abandons the protocol.*

3.1.2 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 nonfaulty 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

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

where $\vec{\mathsf{pk}} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ is the list of all public keys from the PKI setup. Essentially, this proof system is a threshold signature, proving knowledge of $n-f$ signatures on a message from different parties.

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

3.1.3 Security Analysis

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

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 $\mathsf{AbandonableNonEquiv}$ protocol with tag tag

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

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

Proof. Each property is proven separately.

Validity. We have to show that if a nonfaulty 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 a nonfaulty 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 $\mathsf{NEVerify}$. 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 nonfaulty party i that has signed both m and m'. But this clearly contradicts the requirement that each (nonfaulty) 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 nonfaulty party ventually outputs a proof and terminates. Since there are at least $n - f$ nonfaulty 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 2. *The pair* $(\mathsf{AbandonableNonEquiv}, \mathsf{ANEVerify})$ *as described in Algorithms 3 and 4 is an abandonable non-equivocation protocol resilient to f Byzantine faults if $n > 3f$.*

Proof. Each property is proven separately.

Validity and Non-Equivocation. The same arguments as in Theorem 1 applies.

Abandonability. We have to show that at the time $f + 1$ nonfaulty 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 $n - 2f$ nonfaulty parties that have signed m'_i. On the other hand, by assumption, $f + 1$ nonfaulty 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 nonfaulty party abandons the protocol, then every nonfaulty party eventually outputs a proof and terminates. Since there are at least $n - f$ nonfaulty 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. □

3.1.4 Efficiency

Theorem 3. *The* $\mathsf{NonEquiv}$ *and* $\mathsf{AbandonableNonEquiv}$ *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. □

3.2 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 nonfaulty contribution with high probability.

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

3.2.1 Definition

Parties call a PVSS exchange protocol with no input, and may eventually output an aggregated transcript and a proof. Parties can also run the $\mathsf{VerifyExchange}$ algorithm in order to verify that an output includes at least one honestly generated transcript with high probability.

Definition 3. *A PVSS exchange protocol* $(\mathsf{PVSSExchange}, \mathsf{VerifyExchange})$ *has the following properties if all nonfaulty parties call the protocol:*

- ***Verifiability.*** *If some nonfaulty party outputs* $(\mathsf{trans}_i, \pi_i)$*, then* $\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_i) = 1$*. Furthermore, if the adversary provides a pair* (trans, π) *such that* $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$*, then* trans *includes at least one contribution generated by a nonfaulty party that erased its contents without being corrupted.*

- ***Termination.*** *At least one forever-nonfaulty party completes the protocol and outputs a value.*

3.2.2 Construction

In this construction, we use a VRF that receives a key and an index and outputs a boolean flag and a proof. The VRF outputs *true* with probability $\frac{\lambda \log^c n}{n}$ for some chosen constant $c > 1$ 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 between 0 and $\frac{1}{3}$, for example $\frac{1}{6}$. After hearing from $(\frac{2}{3} - \epsilon)\lambda \log^c n$ 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 a nonfaulty party with high probability. In order prove that at least one forever-nonfaulty party terminates we show that for any given set of corrupted parties, there is an extremely low probability that no nonfaulty 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-nonfaulty party terminating.

For our construction, we assume an additional *forward secure* digital signature scheme [BM99], with the syntax $\mathsf{KDS} = (\mathsf{KGen}_{\mathsf{KDS}}, \mathsf{Update}_{\mathsf{KDS}}, \mathsf{Sign}_{\mathsf{KDS}}, \mathsf{Verify}_{\mathsf{KDS}})$, where each party i has a base public-secret key pair $(\bar{\mathsf{pk}}_i, \bar{\mathsf{sk}}_{i,0})$, the public key is fixed but the secret key $\bar{\mathsf{sk}}_{i,0}$ is dynamically updated at regular time intervals (in our case, for each VRF evaluation). Concretely, there exists an update function Update that on input a secret key $\bar{\mathsf{sk}}_{i,r}$ for some time number $r \in \mathbb{N}_0$ outputs the secret key $\bar{\mathsf{sk}}_{i,r+1}$ for the next time number $r+1$. We formally define the syntax in Appendix A. We further define the following non-interactive proof system $\mathsf{PS}_{\mathsf{exch}} = (\mathsf{PProve}_{\mathsf{exch}}, \mathsf{PVerify}_{\mathsf{exch}})$ for the relation

$$\mathcal{R}_{\mathsf{exch}} := \left\{ \left(\vec{\tilde{\mathsf{pk}}}, \vec{\bar{\mathsf{pk}}}, \vec{\mathsf{ek}}, \mathsf{trans}, j; S, \{(\sigma_i, \mathsf{trans}_i, \pi_i)\}_{i \in S}\right) \;\middle|\; \begin{array}{l} |S| = (\frac{2}{3} - \epsilon)\lambda \log^c n,\ \forall i \in S: \\ \mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}_i, \mathsf{ek}_1, \ldots, \mathsf{ek}_n) = 1, \\ \mathsf{Verify}_{\mathsf{VRF}}(\tilde{\mathsf{pk}}_i, true, j, \pi_i) = 1, \\ \mathsf{Verify}_{\mathsf{KDS}}(\bar{\mathsf{pk}}_i, (j, \mathsf{trans}_i), \sigma_i) = 1 \end{array} \right\},$$

where $\vec{\tilde{\mathsf{pk}}} = (\tilde{\mathsf{pk}}_1, \ldots, \tilde{\mathsf{pk}}_n)$ (public keys for the VRF), $\vec{\bar{\mathsf{pk}}} = (\bar{\mathsf{pk}}_1, \ldots, \bar{\mathsf{pk}}_n)$ (public keys for the forward-secure DS), and $\vec{\mathsf{ek}} := (\mathsf{ek}_1, \ldots, \mathsf{ek}_n)$ (encryption keys for the PVSS). Essentially, the proof system tells that a party received $(\frac{2}{3} - \epsilon)\lambda \log^c n$ valid PVSS transcripts from other parties with corresponding valid signatures along with verifying VRF proofs.

3.2.3 Security Analysis

Lemma 1. *If some 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 nonfaulty at the time of erasure.*

Algorithm 5 $\mathsf{PVSSExchange}()$

$\mathsf{messages} \leftarrow \emptyset,\ \mathsf{transcripts} \leftarrow \emptyset$
// Check which parties to send messages to, generate transcripts, and store them
for $j \in [n]$ **do**
  $(\mathsf{sendFlag}_j, \pi_j) \leftarrow \mathsf{VRF}(\tilde{\mathsf{sk}}_i, j)$
  if $\mathsf{sendFlag}_j = true$ **then**
    $\mathsf{trans}_j \leftarrow\!\!\$\ \mathsf{Dist}_{\mathsf{PVSS}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_n)$
    $\mathsf{messages} \leftarrow \mathsf{messages} \cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \mathsf{Sign}_{\mathsf{KDS}}(\bar{\mathsf{sk}}_i, (j, \mathsf{trans}_j)))\}$
// Erase keys and then send messages to all parties
update $\bar{\mathsf{sk}}_i$ via Update and **erase** the old key and the randomness used for $\mathsf{Dist}_{\mathsf{PVSS}}()$
for all $(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j) \in \mathsf{messages}$ **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 $\mathsf{Verify}_{\mathsf{VRF}}(\tilde{\mathsf{pk}}_j, \mathsf{sendFlag}, j, \pi_j) = \mathsf{Verify}_{\mathsf{KDS}}(\bar{\mathsf{pk}}_j, (i, \mathsf{trans}_j), \sigma_j) = 1$ and $\mathsf{sendFlag} = true$ **then**
    $\mathsf{transcripts} \leftarrow \mathsf{transcripts} \cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j)\}$
    if $|\mathsf{transcripts}| \geq (\frac{2}{3} - \epsilon)\lambda \log^c n$ **then**
      $\mathsf{trans} \leftarrow \mathsf{Aggregate}_{\mathsf{PVSS}}(\{\mathsf{trans}_j \mid \exists (j, \mathsf{sendFlag}_j, \pi_j, \mathsf{trans}_j, \sigma_j) \in \mathsf{transcripts}\})$
      $\pi \leftarrow \mathsf{PProve}_{\mathsf{exch}}((\tilde{\mathsf{pk}}_1, \ldots, \tilde{\mathsf{pk}}_n), (\bar{\mathsf{pk}}_1, \ldots, \bar{\mathsf{pk}}_n), \vec{\mathsf{ek}}, \mathsf{trans}; \mathsf{transcripts})$
      output (trans, π) and **terminate**

Algorithm 6 $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi)$

return $\mathsf{PVerify}_{\mathsf{exch}}((\tilde{\mathsf{pk}}_1, \ldots, \tilde{\mathsf{pk}}_n), (\bar{\mathsf{pk}}_1, \ldots, \bar{\mathsf{pk}}_n), \vec{\mathsf{ek}}, \mathsf{trans}; \pi)$

Proof. If trans, π verify, then trans includes transcripts from at least $(\frac{2}{3} - \epsilon)\lambda \log^c n$ 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. Based on the previous discussion, $X_i \leq \sum_{j \text{ } is \text{ } faulty} X_{i,j}$. Therefore: $\mathbb{E}[X_i] \leq \mathbb{E}[\sum_{j \text{ } is \text{ } faulty} X_{i,j}] \leq f \cdot \frac{\lambda \log^c n}{n} < \frac{1}{3}\lambda \log^c n$. Next, we would like to bound the probability that for any $i \in [n]$, at least $(\frac{2}{3}\lambda \log^c n)$ faulty parties are allowed to to send a PVSS transcript generated by them. Applying the Chernoff bound, assuming $\epsilon < \frac{1}{3}$:

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

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

$$n^{-\lambda \log^{c-1} n \cdot \frac{(1-3\epsilon)^2}{9-9\epsilon} + 1}$$

Therefore, choosing any $\epsilon \ll \frac{1}{3}$, such as $\epsilon = \frac{1}{6}$, results in a probability of $n^{-O(\lambda \log^{c-1} n)}$. $\square$

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

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 nonfaulty party receives messages from $(\frac{2}{3} - \epsilon)\lambda \log^c n$ parties. Following that, we will apply the union bound 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-nonfaulty party will eventually receive messages from $(\frac{2}{3} - \epsilon)\lambda \log^c n$ nonfaulty 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 $\frac{\lambda \log^c n}{n}$ and thus:

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

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

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

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:

$$\Pr[\forall i \in I : \; X_i < (\frac{2}{3} - \epsilon)\lambda \log^c n] \leq n^{-n\lambda \log^{c-1} n \cdot \frac{3}{4}\epsilon^2}$$

The adversary can choose any set of f parties to corrupt. In total, the number of options the adversary has is $\binom{n}{f} \leq \binom{n}{\frac{n}{3}} \leq (en \cdot \frac{3}{n})^{\frac{1}{3}n} = (3e)^{\frac{1}{3}n}$. 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 nonfaulty party receives enough messages.

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

In other words, for every $\epsilon > 0$, the probability that no nonfaulty party receives enough messages is $e^{-O(n\lambda \log^c n)}$. $\square$

Theorem 4. *The pair* $(\mathsf{PVSSExchange}, \mathsf{ExchangeVerify})$ *described in Algorithms 5 and 6 is a PVSS Exchange protocol resilient to f Byzantine faults if $n > 3f$.*

Proof. Each property is proven separately.

Verifiability. If some nonfaulty 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 a nonfaulty 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-nonfaulty party receives at least $(\frac{2}{3} - \epsilon)\lambda \log^c n$ messages from other nonfaulty parties. After receiving these messages, it sees that the messages and proofs are correct, performs local computations, and terminates. □

3.2.4 Efficiency

Theorem 5. *The* $\mathsf{PVSSExchange}$ *protocol has an expected communication complexity of* $O(\lambda^2 n^2 \log^c n)$ *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 $\frac{\lambda \log^c n}{n}$. In expectation, every party sends $O(\lambda \log^c n)$ such messages, containing transcripts of size $O(\lambda n)$. In total, this means that the communication complexity is $O(\lambda^2 n^2 \log^c n)$, and the protocol takes a single round. □

3.3 Weighted Gather

The *Weighted Gather* protocol is a modification of the Verifiable Gather protocol [AJM+21]. 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 forward, the verifiability of the weighted gather will be implicitly checked via a quorum of parties in the Proposal Election protocol.

3.3.1 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 nonfaulty 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 4. *A Weighted Gather protocol has the following properties, assuming all nonfaulty parties participate with valid inputs:*

- ***Binding Core.*** *At the time the first nonfaulty 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 nonfaulty* i.
- ***Binding Cover.*** *At the time the first nonfaulty 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 nonfaulty* 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 nonfaulty party completes the protocol.*
- ***Termination.*** *All parties output a value and complete the protocol.*

3.3.2 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 nonfaulty 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 introduces into the protocol, and thus the values that have already been input will form a cover for the outputs of the protocol.

In order to reduce communication complexity, we define the following non-interactive proof systems. First, $\mathsf{PS}_{\mathsf{cnt}} = (\mathsf{PProve}_{\mathsf{cnt}}, \mathsf{PVerify}_{\mathsf{cnt}})$ for the relation

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

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.

We additionally require another 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

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

3.3.3 Security Analysis

Lemma 3. *By the time the first nonfaulty party completes the* WeightedGather *protocol, there exists a vector $v \in \mathbb{N}_0^\ell$ such that $|v| \geq n-f$ and the nonfaulty 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 $\mathsf{NonEquiv}$ 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 nonfaulty 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.

Algorithm 7 $\mathsf{WeightedGather}(x_i, \pi_i)$

$\mathsf{vec}_i \leftarrow \bot,\ \pi_{\mathsf{vec},i} \leftarrow \bot,\ \mathsf{agg}_i \leftarrow \bot,\ \pi_{\mathsf{agg},i} \leftarrow \bot,\ \mathsf{out}_i \leftarrow \bot,\ \pi_{\mathsf{out},i} \leftarrow \bot$

// Non-equivocation of inputs

$\mathsf{vals}_i \leftarrow \emptyset,\ \mathsf{vecs}_i \leftarrow \emptyset,\ \mathsf{aggs}_i \leftarrow \emptyset$

call $\mathsf{AbandonableNonEquiv}((x_i, \pi_i), \text{"val"})$

upon $\mathsf{AbandonableNonEquiv}((x_i, \pi_i), \text{"val"})$ outputting $\pi_{\mathsf{valANE},i}$, **do**

  send $\langle \text{"val"}, x_i, \pi_i, \pi_{\mathsf{valANE},i} \rangle$ to all parties

// Collecting inputs in a vector, non-equivocation of vector

upon receiving the first $\langle \text{"val"}, x_j, \pi_j, \pi_{\mathsf{valANE},j} \rangle$ message from j, **do**

  if $\mathsf{ANEVerify}(j, (x_j, \pi_j), \pi_{\mathsf{valANE},j}, \text{"val"}) = 1$, $\mathsf{valid}(x_j, \pi_j) = 1$ 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 $v_i = |\{j \in [n] \mid \exists \pi, \pi_{\mathsf{valANE}} \text{ s.t. } (j, i, \pi, \pi_{\mathsf{valANE}}) \in \mathsf{vals}_i\}|$

      $\mathsf{vec}_i \leftarrow (v_1, \ldots, v_\ell),\ \pi_{\mathsf{vec},i} \leftarrow \mathsf{PProve}_{\mathsf{cnt}}(\vec{\mathsf{pk}}, \mathsf{vec}_i, (i)_{i \in [\ell]}; \mathsf{vals}_i)$

      call $\mathsf{NonEquiv}(\mathsf{vec}_i, \text{"vec"})$

upon $\mathsf{NonEquiv}(\mathsf{vec}_i, \text{"vec"})$ outputting $\pi_{\mathsf{vecNE},i}$, **do**

  send $\langle \text{"vec"}, \mathsf{vec}_i, \pi_{\mathsf{vec},i}, \pi_{\mathsf{vecNE},i} \rangle$ to all parties

// Collecting vectors in an aggregated vector, non-equivocation of aggregated vector

upon receiving the first $\langle \text{"vec"}, \mathsf{vec}_j, \pi_{\mathsf{vec},j}, \pi_{\mathsf{vecNE},j} \rangle$ message from j, **do**

  if $\mathsf{NEVerify}(j, \mathsf{vec}_j, \pi_{\mathsf{vecNE},j}, \text{"vec"}) = 1$ 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**

      $\mathsf{agg}_i \leftarrow \max\{\mathsf{vec}_j \mid \exists j, \pi_{\mathsf{vec}}, \pi_{\mathsf{vecNE}} \text{ s.t. } (j, \mathsf{vec}_j, \pi_{\mathsf{vec}}, \pi_{\mathsf{vecNE}}) \in \mathsf{vecs}_i\}$

      $\pi_{\mathsf{agg},i} \leftarrow \mathsf{PProve}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{agg}_i, (i)_{i \in [\ell]}; \mathsf{vecs}_i)$

      call $\mathsf{NonEquiv}(\mathsf{agg}_i, \text{"agg"})$

upon $\mathsf{NonEquiv}(\mathsf{agg}_i, \text{"agg"})$ outputting $\pi_{\mathsf{aggNE},i}$, **do**

  send $\langle \text{"agg"}, \mathsf{agg}_i, \pi_{\mathsf{agg},i}, \pi_{\mathsf{aggNE},i} \rangle$ to all parties

// Collecting aggregated vectors for output, send output in order to guarantee availability

upon receiving the first $\langle \text{"agg"}, \mathsf{agg}_j, \pi_{\mathsf{agg},j}, \pi_{\mathsf{aggNE},j} \rangle$ message from j, **do**

  if $\mathsf{NEVerify}(j, \mathsf{agg}_j, \pi_{\mathsf{aggNE},j}, \text{"agg"}) = 1$ 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**

      $\mathsf{out}_i \leftarrow \max\{\mathsf{agg}_j \mid \exists j, \pi_{\mathsf{agg}}, \pi_{\mathsf{aggNE}} \text{ s.t. } (j, \mathsf{agg}_j, \pi_{\mathsf{agg}}, \pi_{\mathsf{aggNE}}) \in \mathsf{aggs}_i\}$

      $\pi_{\mathsf{out},i} \leftarrow \mathsf{PProve}_{\mathsf{max}}(\vec{\mathsf{pk}}, \mathsf{out}_i, (i)_{i \in [\ell]}; \mathsf{aggs}_i)$

upon receiving the first $\langle \text{"out"}, \mathsf{out}_j, \pi_{\mathsf{out},j} \rangle$ 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 $\mathsf{AbandonableNonEquiv}$ *to block new entries*

upon $\mathsf{out}_i \neq \bot$ and $\pi_{\mathsf{out},i} \neq \bot$, **do**

  send $\langle \text{"out"}, \mathsf{out}_i, \pi_{\mathsf{out},i} \rangle$ to all parties

  abandon the $\mathsf{AbandonableNonEquiv}$ session with tag "val"

upon receiving "out" messages from $n - f$ parties, **do**

  output out_i and **terminate**

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:

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

Note that $n > 3f$ and thus:

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

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. $\square$

Theorem 6. $\mathsf{WeightedGather}$ *is a Weighted Gather protocol resilient to f Byzantine faults 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 nonfaulty party completes the protocol, it had already received "out" messages from $n-f$ parties, with at least $f+1$ of those coming from nonfaulty parties. Let I be the set of at least $f+1$ nonfaulty parties described above. When parties send the "out" messages, they abandon the $\mathsf{AbandonableNonEquiv}$ instances with the tag "val". From the Abandonability property of the $\mathsf{AbandonableNonEquiv}$ 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 $x_j = \bot$ for any $j \in [n]$, then all nonfaulty 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 nonfaulty party i updates $\mathsf{out}_i, \pi_{\mathsf{out},i}$ to be non-$\bot$, all nonfaulty 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 nonfaulty j will also update $\mathsf{out}_j, \pi_{\mathsf{out},i}$ to non-$\bot$ 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 nonfaulty party i ever has $\mathsf{out}_i \neq \bot$ and $\pi_{\mathsf{out},i} \neq \bot$. All parties start by calling the $\mathsf{AbandonableNonEquiv}$ protocol. Note that if some nonfaulty party i abandons the $\mathsf{AbandonableNonEquiv}$ protocol, it has $\mathsf{out}_i \neq \bot$ and $\pi_{\mathsf{out},i} \neq \bot$. As shown above, in that case all parties complete the protocol, reaching a contradiction. Therefore, no nonfaulty party abandons the protocol. From the Liveness property of $\mathsf{AbandonableNonEquiv}$, every nonfaulty 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 nonfaulty party calls $\mathsf{NonEquiv}$ with tag "vec" and outputs a verifying proof. Parties then send "vec" messages, accept messages sent by nonfaulty parties and similarly compute agg values and call $\mathsf{NonEquiv}$ with tag "agg". Finally, nonfaulty parties send "agg" messages, accept each other's messages, and compute $\mathsf{out}_i, \pi_{\mathsf{out},i}$, reaching a contradiction. Therefore, eventually some nonfaulty party has $\mathsf{out}_i \neq \bot, \pi_{\mathsf{out},i} \neq \bot$. As shown above, this means that eventually all parties complete the protocol. $\square$

3.3.4 Efficiency

Theorem 7. *The* $\mathsf{WeightedGather}$ *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 $\mathsf{NonEquiv}$ and $\mathsf{AbandonableNonEquiv}$ 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. $\square$

3.4 Provable AVID Protocol

A *Provable Asynchronous Verifiable Information Dispersal (AVID)* protocol consists of two sub-protocols $\mathsf{Disperse}$ and $\mathsf{Retrieve}$ along with a $\mathsf{DisperseVerify}$ function. Parties also have access to an external validity function valid. In the $\mathsf{Disperse}$ 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 $\mathsf{DisperseVerify}$ algorithm), which indicates that the $\mathsf{Retrieve}$ protocol will succeed. In the $\mathsf{Retrieve}$ 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.

3.4.1 Definition

Formally, we define a provable AVID protocol as follows.

Definition 5. *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 nonfaulty 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 nonfaulty, then the committed value is its input* $m' = m$*.*
- ***Agreement.*** *Any two nonfaulty parties that output a value from the* $\mathsf{Retrieve}$ *protocol, output the same value* m' *(defined in the binding property above).*
- ***Termination.*** *If the sender is nonfaulty, then it completes the* $\mathsf{Disperse}$ *protocol and outputs a proof* π*.*
- ***Provability.*** *If* s *is nonfaulty and it outputs a proof* π*, then* $\mathsf{DisperseVerify}(s, \pi) = 1$*.*
- ***Totality.*** *If a nonfaulty party receives a proof* π *such that* $\mathsf{DisperseVerify}(s, \pi) = 1$ *and all nonfaulty parties call* $\mathsf{Retrieve}$*, then they all complete* $\mathsf{Retrieve}$*.*

3.4.2 Construction

For our protocol, we assume an external validity function valid that parties have access to. Further, we define 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.

In the following, we give an informal description of how our provable AVID protocol works, which is based on linear erasure codes and cryptographic accumulators. We first describe the dispersal phase, where a designated sender $s \in [n]$ holds an input message m that it wants to disperse among the network of parties. Then, we describe 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 generate 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.

3.4.3 Security Analysis

Theorem 8. *The pair* ($\mathsf{Disperse}, \mathsf{DisperseVerify}, \mathsf{Retrieve}$) *is a provable AVID protocol resilient to f Byzantine faults if $n > 3f$.*

Proof. Each property is proven separately.

Binding. We have to show that at the time the first nonfaulty party completes the $\mathsf{Disperse}$ 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 nonfaulty party will reconstruct in the retrieval phase). By construction, we know that a nonfaulty 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 nonfaulty, then all nonfaulty parties will complete the $\mathsf{Disperse}$ protocol. Since there are at least $n - f$ nonfaulty 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 nonfaulty 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 $\mathsf{Disperse}$ protocol and output a proof π.

Provability. We have to show that if the sender is nonfaulty and it outputs a proof π, then $\mathsf{DisperseVerify}(s, \pi) = 1$. But this is clear from the above termination considerations, since the proof is a threshold signature of

Algorithm 8 $\mathsf{Disperse}(m)$

$\mathsf{sigs} \leftarrow \emptyset,\ \mathsf{sigs}' \leftarrow \emptyset,\ \pi_{acc} \leftarrow \perp,\ \pi_{tot} \leftarrow \perp,\ M' \leftarrow \perp$
// Encode the message in an error-correcting code and compute an accumulator of the encoding
$M' \leftarrow (m'_1, \ldots, m'_n) \leftarrow \mathsf{Encode}(m; n, f+1)$
$\mathsf{acc} \leftarrow \mathsf{Eval}_{\mathsf{Acc}}(ak, M')$
for $i \in [n]$ **do**
  $w_i \leftarrow \mathsf{Witness}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, M')$
// Non-equivocation of accumulator
send $\langle \text{"help"}, \mathsf{acc} \rangle$ to all parties
upon receiving the first $\langle \text{"help"}, \mathsf{acc}, \pi \rangle$ message from j, **do**
  if $\mathsf{valid}(\mathsf{acc}) = 1$ **then**
    send $\langle \text{"sig"}, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (j, \mathsf{acc})) \rangle$ to j
upon receiving the first $\langle \text{"sig"}, \sigma_i \rangle$ message from i, **do**
  if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (i, \mathsf{acc})) = 1$ **then**
    $\mathsf{sigs} \leftarrow \mathsf{sigs} \cup \{(i, \sigma_i)\}$
    if $|\mathsf{sigs}| = n - f$ **then**
      $\pi_{\mathsf{TS}} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, \mathsf{acc}); \mathsf{sigs})$
      $\pi_{acc} \leftarrow \mathsf{PProve}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); m, \pi_{\mathsf{TS}})$
// Send shares of the encoding to each party, parties reply once they receive their shares
upon $\pi_{acc} \neq \perp$, **do**
  for $i \in [n]$ **do**
    send $\langle \text{"share"}, \mathsf{acc}, \pi_{acc}, m'_i, w_i \rangle$ to i
upon receiving the first $\langle \text{"share"}, \mathsf{acc}, \pi_{acc}, m'_i, w_i \rangle$ message from j, **do**
  if $\mathsf{PVerify}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); \pi_{acc}) = 1$ and $\mathsf{Verify}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, w_i) = 1$ **then**
    send $\langle \text{"total"}, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, j) \rangle$ to j
// Wait until $n - f$ parties received their shares, and output a proof
upon receiving the first $\langle \text{"total"}, \sigma_i \rangle$ message from i, **do**
  if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, i) = 1$ **then**
    $\mathsf{sigs}' \leftarrow \mathsf{sigs}' \cup \{(i, \sigma_i)\}$
    if $|\mathsf{sigs}'| = n - f$ **then**
      $\pi_{tot} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, i; \mathsf{sigs}')$
      output π_{tot} and **terminate**

Algorithm 9 $\mathsf{DisperseVerify}(i, \pi)$

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

Algorithm 10 $\mathsf{Retrieve}()$

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

weight $n-f$. As such, the underlying threshold signature verification algorithm $\mathsf{PVerify}_T S$ verifies for π and so does $\mathsf{DisperseVerify}(s, \pi)$.

External Validity. We have to show that if the sender is nonfaulty, then the committed value is its input $m' = m$. Further, any nonfaulty 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 a nonfaulty party completes the dispersal phase, then the committed value m is externally valid.

Agreement. We have to show that any two nonfaulty 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 a nonfaulty party receives a proof π such that $\mathsf{DisperseVerify}(s, \pi) = 1$ and all nonfaulty parties call Retrieve, then they all complete Retrieve. So assuming a nonfaulty 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 $n-2f$ nonfaulty parties with valid shares corresponding to acc. Thus, if all nonfaulty parties call the Retrieve protocol, then any nonfaulty party will receive at least $n-2f \geq f+1$ valid shares from other nonfaulty parties. From that, it is clear that any nonfaulty 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 nonfaulty parties will complete the Retrieve protocol.

□

3.4.4 Efficiency

Theorem 9. *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 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. □

3.5 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 $i^* \in [n]$.

3.5.1 Definition

Definition 6. *A Weak Coin protocol* (WDCG, Flip) *has the following properties, assuming nonfaulty parties call* $\mathsf{Flip}(v_i, tag)$ *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 nonfaulty parties output the same uniformly sampled* i^* *from* Flip*. Furthermore, if* E *takes place, the adversary's view is independent of* i^* *before some nonfaulty party calls* Flip *with tag.*
- ***Termination.*** *If all nonfaulty parties participate in the* WDCG *protocol, they complete it. If all nonfaulty parties participate in the* Flip *protocol with the same tag, they all complete it.*

3.5.2 Construction

Weak Distribution 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.

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.

To reduce message complexity, we also define the following non-interactive proof system, which models a threshold signature with generalized weight r. Concretely, we define $\mathsf{PS}_{r\mathsf{TS}} = (\mathsf{PProve}_{r\mathsf{TS}}, \mathsf{PVerify}_{r\mathsf{TS}})$ with $r \in \mathbb{N}$ for the relation

$$\mathcal{R}_{r\mathsf{TS}} := \Big\{ \Big(\vec{\mathsf{pk}}, m; S, \{\sigma_i\}_{i \in S} \Big) \ \Big| \ |S| = r,\ \forall i \in S : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, m, \sigma_i) = 1 \Big\}.$$

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 nonfaulty 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, $\mathsf{BaseConsensus}$, 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, $\mathsf{RecConsensus}$ is called in every other later 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 3.6. 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 $\mathsf{PVerify}_{\mathsf{TS}}$. This reduces the costs of every part of the protocol other than the PE to $O(\lambda n^2)$ in expectation per view.

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

$$(1-\alpha)^r = \left(\frac{7}{9}\right)^{\lambda \log^c n \cdot \log^{-1}(\frac{9}{7})} = 2^{-\lambda \log^c},$$

in which case even the calls to the consensus protocols with a nonfaulty 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 $\mathsf{BaseConsensus}$ and $\mathsf{RecConsensus}$ protocols with 3 arguments, $x, i_{\min}, i_{\max}$. x is the input to the protocol, and $i_{\min}, i_{\max}$ signify that parties $i_{\min}, \dots, i_{\max}$ 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 $i_{\min}, \dots, i_{\max}$.

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 nonfaulty 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.

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

if $i < i_{\min}$ or $i > i_{\max}$ **then**
    terminate without participating in the protocol
$\mathsf{child} \leftarrow \mathsf{partition}(i_{\min}, i_{\max}, \ell)$, $\mathsf{trans}_i \leftarrow \perp, \pi_{\mathsf{trans},i} \leftarrow \perp, \mathsf{agreed}_i \leftarrow \perp, \pi_{\mathsf{agreed},i} \leftarrow \perp, \pi_{\mathsf{child},i} \leftarrow \perp, \mathsf{sigs}_i \leftarrow \emptyset$
$\forall j \in [\ell]:$ $\mathsf{helped}[j] \leftarrow false$, $\mathsf{outputs}[j] \leftarrow \perp$, $\mathsf{proofs}[j] \leftarrow \perp$, $\mathsf{proofSigs}[j] \leftarrow \emptyset$, $\mathsf{retrieved}[j] \leftarrow \perp$
// Get aggregated PVSS, disperse and forward
call $\mathsf{PVSSExchange}()$
upon $\mathsf{PVSSExchange}()$ terminating with output trans, π, **do**
    call $\mathsf{Disperse}((\mathsf{trans}, \pi))$ with external validity $\mathsf{validDisperse}$
upon $\mathsf{Disperse}()$ 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 $\mathsf{BaseConsensus}((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), i_{\min}, i_{\max})$ with external validity function $\mathsf{DisperseVerify}$
    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 $\mathsf{WDCG}(i'_{\min}, i'_{\max})$
        upon $\mathsf{WDCG}(i'_{\min}, i'_{\max})$ terminating with output v_i, **do**
            call $\mathsf{RecConsensus}((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), i'_{\min}, i'_{\max})$ with external validity function $\mathsf{DisperseVerify}$, using v_i and the state of $\mathsf{WDCG}(i'_{\min}, i'_{\max})$ for the PE protocol
upon $\mathsf{BaseConsensus}$ outputting $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$, **do**
    output $v_i = (1, 0, \ldots, 0)$, update $\mathsf{outputs}[1] \leftarrow \mathsf{agreed}$, and terminate
upon $\mathsf{RecConsensus}$ outputting $(\mathsf{agreed}, \pi_{\mathsf{agreed}})$, **do**
    $\mathsf{agreed}_i \leftarrow \mathsf{agreed}$, $\pi_{\mathsf{agreed},i} \leftarrow \pi_{\mathsf{agreed}}$
// $\mathsf{ChildNonEquiv}$ runs a non-equivocation procedure per child on agreed
// When seeing non-equivocation proofs, parties send signatures, which are stored in $\mathsf{proofSigs}$
run $\mathsf{ChildNonEquiv}()$
// After seeing that the output is unique and sent to all parties, continue with $\mathsf{WeightedGather}$
upon $|\mathsf{proofSigs}[k]| = f + 1$ for any $k \in [\ell]$, **do**
    if $\mathsf{WeightedGather}$ has not been called yet **then**
        call $\mathsf{WeightedGather}(k, \mathsf{Prove}(k, \mathsf{proofSigs}[k]))$ with external validity function $\mathsf{validWG}$
upon $\mathsf{WeightedGather}$ 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 $\mathsf{outputs}[k] \neq \perp$ for any $k \in [\ell]$ and $\mathsf{WeightedGather}$ terminating, **do**
    call $\mathsf{Retrieve}()$ for retrieving the value dispersed by disp, where $\mathsf{outputs}[k] = (\mathsf{disp}, \pi)$
upon $\mathsf{Retrieve}()$ outputting $(\mathsf{trans}_j, \pi_j)$ for the value dispersed by j, **do**
    $\mathsf{retrieved}[j] \leftarrow \mathsf{trans}_j$

Algorithm 12 $\mathsf{ChildNonEquiv}()$

upon $\mathsf{agreed}_i \neq \bot$, **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 $\mathsf{helped}[\mathsf{child}[j]] = \mathit{false}$ **then**
      send $\langle$"childSig"$, \mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (\mathsf{child}[j], \mathsf{agreed}_j,$ "childSig"$))\rangle$ to all k s.t. $\mathsf{child}[k] = \mathsf{child}[j]$
      $\mathsf{helped}[\mathsf{child}[j]] \leftarrow \mathit{true}$
// Collect signatures, send message with proof after receiving enough
upon receiving the first $\langle$"childSig"$, \sigma_j\rangle$ message from j and having $\mathsf{agreed}_i \neq \bot$, **do**
  if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (\mathsf{child}[i], \mathsf{agreed}_i,$ "childSig"$), \sigma_j) = 1$ **then**
    $\mathsf{sigs}_i \leftarrow \mathsf{sigs}_i \cup \{(j, \sigma_j)\}$
    if $|\mathsf{sigs}_i| = n - f$ and $\mathsf{outputs}[\mathsf{child}[i]] = \bot$ **then**
      $\mathsf{outputs}[\mathsf{child}[i]] \leftarrow \mathsf{agreed}_i,\ \mathsf{proofs}[\mathsf{child}[i]] \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (\mathsf{child}[i], \mathsf{agreed}_i,$ "childSig"$); \mathsf{sigs}_i)$
upon $\mathsf{outputs}[j] \neq \bot$ for any $j \in [\ell]$, **do**
  send $\langle$"childProof"$, j, \mathsf{outputs}[j], \mathsf{proofs}[j], \mathsf{Sign}(\mathsf{sk}_i, (j,$ "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 $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (k, \mathsf{agreed},$ "childSig"$); \pi_{\mathsf{child}}) = \mathsf{Verify}(\mathsf{pk}_j, (k,$ "childProof"$), \sigma) = 1$ **then**
    if $\mathsf{outputs}[k] = \bot$ **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 $\mathsf{validDisperse}((\mathsf{trans}, \pi))$

if $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, (\mathsf{ek}_1, \ldots, \mathsf{ek}_n)) = 1$ and $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$ **then**
  return 1
else
  return 0

Algorithm 14 $\mathsf{validWG}(i, \pi_i)$

return $\mathsf{PVerify}_{f+1\mathsf{TS}}(\vec{\mathsf{pk}}, (i,$ "childProof"$); \pi_i)$

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

$\forall j \in [n]:\ \mathsf{flips}[j] \leftarrow \bot,\ \mathsf{partials}[j] \leftarrow \emptyset$
// Participate in flipping all coins
upon $\mathsf{retrieved}[j] \neq \bot$ for any j, **do**
  send $\langle$"partial"$, \mathsf{PartialCoin}(\mathsf{dk}_i, j, \mathsf{retrieved}[j]), j, tag\rangle$ to all parties
upon receiving the first $\langle$"partial"$, \mathsf{pc}_j, k\rangle$ message from j for any k, **do**
  if $\mathsf{VerifyPartial}(\mathsf{ek}_j, j, \mathsf{pc}_j, \mathsf{retrieved}[k]) = 1$ **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 $\mathsf{flips}[j] \neq \bot$ 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 $\mathsf{H}(r_j, 0, tag)$

3.5.3 Security Analysis

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

Lemma 4. *If all nonfaulty 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 nonfaulty party completes 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 nonfaulty 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 nonfaulty 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 $(k,$ "childProof"$)$. Nonfaulty 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 $(j, \mathsf{agreed},$ "childSig"$)$, and every nonfaulty 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 a nonfaulty party that sends only one such message. In other words, all nonfaulty parties eventually update $\mathsf{outputs}[j]$ to be the same value agreed. In addition, a nonfaulty 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 nonfaulty 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 nonfaulty party i, and thus every nonfaulty party eventually sees that $\mathsf{flips}[j] \neq \bot$ 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 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 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. $\square$

Theorem 10. (WDCG, Flip) *are a Weak Coin protocol resilient to f Byzantine faults if $n > 3f$, with $\alpha = \frac{2}{3}$.*

Proof. Each property is proven separately.

$\frac{2}{3}$-Uniformity. Assume all nonfaulty 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 WeightedGather and outputs a vector v_i. From Lemma 4, all nonfaulty parties complete their call to Flip. From the binding core property of the WeightedGather protocol, when the first nonfaulty party completes 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 nonfaulty 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 nonfaulty i. As shown in the proof of Lemma 4, all parties store the same correct transcripts in their 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 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[1] 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 nonfaulty 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 nonfaulty 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 nonfaulty party calls Flip with tag, no nonfaulty party computes 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 a nonfaulty 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 outputs$[1]$ and output a uniformly sampled value with probability 1.

Termination. Lemma 4 shows that if all nonfaulty 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 nonfaulty parties start the protocol by calling PVSSExchange. From the Termination and Correctness properties of PVSSExchange, at least one forever-nonfaulty party i completes the protocol with a verifying $\mathsf{trans}, \pi_{\mathsf{Exchange}}$. Party i then calls the 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 nonfaulty party receives these messages and updates its corresponding $\mathsf{disp}, \pi_{\mathsf{disp}}$ variables if it hasn't already done so. Following that, every nonfaulty party calls a consensus protocol.

In the base case, all parties call the 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 RecConsensus has less than a $\frac{1}{3}$ fraction of faulty parties. Let that call be the call of child i, i.e. the one for which every party j participating in the call has child$[j] = i$. Therefore, the i'th call has all properties of a WDCG protocol and thus all nonfaulty parties complete it. Following that, all nonfaulty parties use a correct WDCG setup in the consensus protocol and thus all nonfaulty 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 nonfaulty party j such that child$[j] = i$ will send a "childNE" message upon completing the protocol. Every nonfaulty 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 nonfaulty parties will respond to the first such message with the value agreed. After receiving $n - f$ such signatures, every party j such that child$[j] = i$ will update its outputs$[i]$ and proofs$[i]$ variables and send a "childProof" message, if it

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

hasn't done so earlier. If some nonfaulty 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 nonfaulty party eventually receives that message, updates its $\mathsf{outputs}[k], \mathsf{proofs}[k]$ values, adds a signature to $\mathsf{proofSigs}$, and sends a "childProof" message as well. Eventually, every nonfaulty party will gather enough signatures for in $\mathsf{proofSigs}[i]$ (and possibly other entries in $\mathsf{proofSigs}$), generate a proof that it saw at least $f+1$ such signatures, and call $\mathsf{WeightedGather}$ if it hasn't done so earlier. Since the proof is honestly generated, every nonfaulty party calls the $\mathsf{WeightedGather}$ protocol with an externally valid input. From the Termination property of the $\mathsf{WeightedGather}$ protocol, all nonfaulty 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 $\mathsf{BaseConsensus}$ and $\mathsf{RecConsensus}$. As discussed above, parties only run $\lambda\log^c n\cdot\log^{-1}(\frac{9}{7})$ views of each call. The probability that the $\mathsf{RecConsensus}$ call for the i'th child does not terminate in this number of rounds is at most $\frac{7}{9}^{\lambda\log^c n\cdot\log^{-1}(\frac{9}{7})} = 2^{-\lambda\log^c n}$, and thus Termination holds with all but a negligible probability. $\square$

3.5.4 Efficiency

Theorem 11. *The* WDCG *protocol has a communication complexity of* $O(\ell\cdot\lambda^2 n^2\log^c n)$ *and a round complexity of* $O(\log_\ell n)$*. The* Flip *protocol has a communication complexity of* $O(\lambda n^2)$ *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 $\mathsf{PVSSExchange}$ protocol, costing $O(\lambda^2 n^2\log^c n)$;
- $O(n)$ calls to $\mathsf{Disperse}$ with inputs of size $O(\lambda n)$, for a total of $O(\lambda n^2)$;
- either one call to $\mathsf{BaseConsensus}$ with a cost of $O(\lambda n^3\cdot\lambda\log^c n)$ or ℓ calls to $\mathsf{RecConsensus}$, for a total cost of $O(\ell\lambda n^2\cdot\lambda\log^c n)$;
- ℓ calls to $\mathsf{Retrieve}$, for a total cost of $O(\ell\lambda n^2)$;
- a single call to $\mathsf{WeightedGather}$ 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 $\ell\lambda^2 n^2\log^c n$ and each call in the base case costs

$$\ell\lambda^2 n^2\log^c n + \lambda^2 n^3\log^c n = O(\lambda^2 n^3\log^c n).$$

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:

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

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:

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

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 nonfaulty 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 nonfaulty 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 $O(\lambda+\log n)$, for a total of $O((\lambda+\log n)n^2)$ communication and $O(1)$ rounds. $\square$

3.6 Proposal Election from Weak Coin

3.6.1 Definition

In a proposal election protocol, every nonfaulty 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 nonfaulty parties start the protocol with valid inputs prop_i. Every nonfaulty party eventually outputs a value out_i and a proof π_i. Parties can verify the correctness of outputs and proofs with the $\mathsf{PEVerify}$ algorithm by computing $\mathsf{PEVerify}(\mathsf{out}, \pi, tag)$.

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

- ***α-Binding.*** *With probability at least α, at the time the first nonfaulty party completes the* PE *protocol, there exists some party i^* that was nonfaulty 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 a nonfaulty 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* $\mathsf{out} = \bot$*.*

- ***Termination.*** *If all nonfaulty parties participate in the* PE *protocol, they complete it.*

Note that combining α-Binding and completeness, we immediately find that with probability α or greater, all nonfaulty parties output the same value out. In addition, since $\mathsf{PEVerify}$ is an algorithm (not a protocol with a state) in this construction, all nonfaulty 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 $\mathsf{PEVerify}$ protocol outputting 1 to it terminating and 0 to it not terminating (for example, by constructing a protocol that wraps $\mathsf{PEVerify}$ and does this exactly) results in the exact same properties as those of [AJM+21].

3.6.2 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 4 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 $\bot$.

2. Second, parties might elect different indices with different proposals. Any party that hears two different proposals can also output $\bot$.

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 nonfaulty parties output the same index r of a nonfaulty 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 $\bot$ and prove that there was either disagreement on the elected party r, or that many parties didn't store its proposal.

To reduce communication complexity, we define the following non-interactive proof system, which essentially shows that the same value r was received $f+1$ 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

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

3.6.3 Security Analysis

Theorem 12. *The* PE *protocol is a Proposal Election protocol resilient to* f *Byzantine faults if* $n > 3f$*, with* $\alpha = \frac{2}{9}$.

Proof. Each property is proven separately.

$\frac{2}{9}$-Binding. Assume some nonfaulty party completes the PE protocol. At that time, it had $n-f$ non-$\bot$ 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 nonfaulty 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 nonfaulty parties that send such messages at that time, and note that $|I| \geq n-f$. Nonfaulty parties send such a message after receiving "stored" messages from $n-f$ parties, with $f+1$ of those being sent by nonfaulty parties. Finally, nonfaulty parties only send such a "stored" message to j after seeing that receiving a $\langle$"proposal"$, \mathsf{prop}_j, \pi_{\mathsf{propNE},j}, ($"proposal"$, tag)\rangle$ from j such that $\mathsf{Verify}(j, \mathsf{prop}_j, \pi_{\mathsf{propNE},j}, ($"proposal"$, tag)) = 1$ and storing $(\mathsf{prop}_j, \pi_{\mathsf{propNE},j})$ in $\mathsf{proposals}[j]$.

As shown in Theorem 10, the Flip protocol has the $\frac{2}{3}$-Uniformity property. This means that with probability $\frac{2}{3}$ or greater, all nonfaulty parties output the same uniformly sampled index $r \in [n]$ that is independent of the adversary's view before the first nonfaulty party calls Flip. In that case, every nonfaulty 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 nonfaulty $f+1$ 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 nonfaulty 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 $\mathsf{Verify}(r, \mathsf{prop}_r, \pi_{\mathsf{propNE},r}, ($"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 nonfaulty parties with the value r, and the nonfaulty parties sign no other values. This means that $\mathsf{Verify}(\bot, \pi_{r'}, tag)$ 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 $\mathsf{out} = \bot$ 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 nonfaulty 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 nonfaulty 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 $\alpha = \frac{2}{3} \cdot \frac{1}{3} = \frac{2}{9}$.

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

1: $\forall j \in [n]:\ \mathsf{proposals}[j] \leftarrow \bot,\ \mathsf{elected}[j] \leftarrow \bot$
2: $v_i \leftarrow \bot,\ r^* \leftarrow \bot,\ \pi_{r^*,i} \leftarrow \bot,\ \mathsf{out}_i \leftarrow \bot,\ \pi_{\mathsf{out},i} \leftarrow \bot$
// Lines 3-5 can be called once to set up many instances of PE
3: call $\mathsf{WDCG}()$
4: **upon** WDCG outputting v, **do**
5: $\quad v_i \leftarrow v$
// Non-equivocation of inputs, store proposals for future use
6: call $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$
7: **upon** $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$ terminating with output π_{propNE}, **do**
8: $\quad$ send $\langle$“proposal”$, \mathsf{prop}_i, \pi_{\mathsf{propNE}}, tag\rangle$ to all parties
9: **upon** receiving the first $\langle$“proposal”$, \mathsf{prop}_j, \pi_{\mathsf{propNE},j},($“proposal”$, tag)\rangle$ message from j, **do**
10: $\quad$ **if** $\mathsf{valid}(\mathsf{prop}_j) = 1$ and $\mathsf{NEVerify}(j, \mathsf{prop}_j, \pi_{\mathsf{propNE},j},($“proposal”$, tag)) = 1$ **then**
11: $\quad\quad \mathsf{proposals}[j] \leftarrow (\mathsf{prop}_j, \pi_{\mathsf{propNE},j})$
12: $\quad\quad$ send $\langle$“stored”$, tag\rangle$ to j
// Wait until many parties stored proposal, then flip coin
13: **upon** receiving $\langle$“stored”$, tag\rangle$ messages from $n-f$ parties, **do**
14: $\quad$ send $\langle$“ready”$, tag\rangle$ messages to all parties
15: **upon** receiving $\langle$“ready”$, tag\rangle$ messages from $n-f$ parties and $v_i \neq \bot$, **do**
16: $\quad$ call $\mathsf{Flip}(v_i, tag)$
// Inform each other of coin value, and of the stored proposal for the winning party
17: **upon** $\mathsf{Flip}(v_i, tag)$ terminating with output r_i, **do**
18: $\quad$ 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
19: **upon** receiving the first $\langle$“elected”$, r_j, \mathsf{val}_j, \sigma_j, tag\rangle$ message from j, **do**
20: $\quad$ **if** $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j,(r_j, \mathsf{val}_j, tag), \sigma_j) = 1$ **then**
21: $\quad\quad \mathsf{elected}[j] \leftarrow (r_j, \mathsf{val}_j, \sigma_j)$
22: $\quad\quad$ **if** $\mathsf{elected}[j] \neq \bot$ for exactly $n-f$ indices $j \in [n]$ **then**
23: $\quad\quad\quad$ **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**
24: $\quad\quad\quad\quad r^* \leftarrow r,\ \pi_{r^*,i} \leftarrow \mathsf{PProve}_{\mathsf{elect}}(\vec{\mathsf{pk}},(r, tag); \mathsf{elected})$ ▷ choose an r arbitrarily if more than one
25: $\quad\quad\quad\quad$ **if** $\exists j \in [n]$ such that $\mathsf{elected}[j] = (r, \mathsf{val}_j, \sigma_j)$, $\mathsf{val}_j = (\mathsf{prop}, \pi) \neq \bot$, $\mathsf{valid}(\mathsf{prop}) = 1$,
26: $\quad\quad\quad\quad$ and $\mathsf{NEVerify}(r, \mathsf{prop}, \pi,($“proposal”$, tag)) = 1$ **then**
27: $\quad\quad\quad\quad\quad \mathsf{out}_i \leftarrow \mathsf{prop},\ \pi_{\mathsf{out},i} \leftarrow \mathsf{Prove}(r^*, \mathsf{out}_i, \mathsf{elected}, tag)$
28: $\quad\quad\quad\quad$ **else**
29: $\quad\quad\quad\quad\quad \mathsf{out}_i \leftarrow \bot,\ \pi_{\mathsf{out},i} \leftarrow \mathsf{Prove}(r^*, \bot, \mathsf{elected}, tag)$
30: $\quad\quad\quad$ **else**
31: $\quad\quad\quad\quad r^* \leftarrow \bot,\ \pi_{r^*,i} \leftarrow \mathsf{Prove}(\bot, \mathsf{elected}, tag)$
32: $\quad\quad\quad\quad \mathsf{out}_i \leftarrow \bot,\ \pi_{\mathsf{out},i} \leftarrow \bot$
33: $\quad\quad\quad$ **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 $r = \mathsf{out} = \bot$ and $\mathsf{Verify}(\bot, \pi_r, tag) = 1$ **then**
$\quad$ **return** 1
else if $\mathsf{Verify}(r, \pi_r, tag) = 1$ **then**
$\quad$ **if** $\mathsf{out} \neq \bot$ and $\mathsf{Verify}(r, \mathsf{out}, \pi_{\mathsf{out}}, tag) = 1$ **then**
$\quad\quad$ **return** 1
$\quad$ **else if** $\mathsf{out} = \bot$ and $\mathsf{Verify}(r, \bot, \pi_{\mathsf{out}}, tag) = 1$ **then**
$\quad\quad$ **return** 1
return 0

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

External Validity. If $\mathsf{out} = \bot$, 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 nonfaulty i calls $\mathsf{PE}(\mathsf{prop}_i, tag)$, they all start by calling $\mathsf{WDCG}()$ and $\mathsf{NonEquiv}(\mathsf{prop}_i, tag)$. Every nonfaulty i completes both calls, updates v_i, and sends a "proposal" message. Nonfaulty 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 nonfaulty 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. □

3.6.4 Efficiency

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 $O(\ell \cdot \lambda^2 n^2 \log^c n)$ for the single call to WDCG plus $O((\lambda + \log n + m)n^2)$ per call to PE.

Theorem 13. *The* PE *protocol requires a single call to* WDCG *plus* $O((\lambda + \log n + m)n^2)$ *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 $O((\lambda + \log n)n^2)$ and a constant number of rounds of sending messages of size $\lambda + m$. In total, this costs $O((\lambda + \log n + m)n^2)$ 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. □

4 Asynchronous Distributed Key Generation

In this section, we describe a simple DKG 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 of Section 3.6. 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.

4.1 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 sk_i*, a vector of* public key shares $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*, 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* $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*.*

- ***Correctness.*** *There exists a deterministic algorithm* Reconstruct *that on input any set of $t+1$ secret key shares $\{\mathsf{sk}_i\}_{i \in I}$, 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 $x := \mathsf{sk}$ 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 $s^* = x$.*

4.2 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.

Algorithm 18 ADKG()

$\mathsf{disp}_i \leftarrow \perp,\ \pi_{\mathsf{disp},i} \leftarrow \perp$
call PVSSExchange()
// Generate a candidate transcript, disperse it, and inform all parties about it being dispersed
upon PVSSExchange() terminating with output (trans, π), **do**
  call Disperse$((\mathsf{trans}, \pi))$ with external validity validDisperse
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 $\mathsf{WDCG}(1, n)$
upon the $\mathsf{WDCG}(1, n)$ protocol terminating with output v_i, **do**
  call $\mathsf{RecConsensus}((\mathsf{disp}_i, \pi_{\mathsf{disp},i}), 1, n)$ with external validity function DisperseVerify, using v_i and the state of $\mathsf{WDCG}(i'_{\min}, i'_{\max})$ 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((trans, π))

```
if Verify_PVSS(trans, (ek_1, ..., ek_n)) = 1 and ExchangeVerify(trans, π) = 1 then
    return 1
else
    return 0
```

4.3 Security Analysis

Theorem 14. *The* ADKG *protocol described in Algorithm 18 is an Asynchronous Distributed Key Generation protocol resilient to f faults if $n > 3f$.*

Proof. Each property is proven separately.

Correctness. If some nonfaulty 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 $\mathsf{validDisperse}(\mathsf{trans}, \pi) = 1$, $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, (\mathsf{ek}_1, \ldots, \mathsf{ek}_n)) = 1$. This means that the output is indeed a correct PVSS transcript, as required.

Termination. At least one nonfaulty 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 nonfaulty 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 nonfaulty 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 nonfaulty 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 $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$.

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 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 PVSSExchange(). 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 $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$, and its secret key share sk_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 defined 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 RecConsensus. Namely, whenever an instance of the consensus protocol 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 RecConsensus and AVID. From our termination and correctness considerations in the first part of our proof and the security of the 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 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) $\mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n) = 1$ 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. Finally, it follows that we can bound the winning probability of this final

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

□

4.4 Efficiency

Theorem 15. *The* ADKG *protocol has an expected communication complexity of $O(\ell \cdot \lambda^2 n^2 \log^c n)$ and an expected round complexity of $O(\log_\ell n)$.*

Proof. The protocol consists of a single call to PVSSExchange costing $O(\lambda^2 n^2 \log^c n)$, 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 $O(\ell \cdot \lambda^2 n^2 \log^c n)$ and the consensus costs an expected $O(\lambda n^2)$. In total, the ADKG protocol has an expected complexity of $O(\ell \cdot \lambda^2 n^2 \log^c n)$. In addition, 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 nearly quadratic communication complexity of $O(\lambda^2 n^2 \log^c n)$, but a round complexity of $\log(n)$. 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 $0 < \epsilon \leq 1/2$), 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 $O(\lambda^2 n^{2+\frac{1}{2}} \log^c n)$ (or more generally $O(\lambda^2 n^{2+\epsilon})$.

References

[AJM+21] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. *CoRR*, abs/2102.09041, 2021.

[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.

[AMS18] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. *CoRR*, abs/1811.01332, 2018.

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

[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.

[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.

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

[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.

[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.

[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.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

[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.

[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.

[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.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*, page 33–62, Berlin, Heidelberg, 2018. Springer-Verlag.

[FLT24] 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.

[FMT24] Hanwen Feng, Tiancheng Mai, and Qiang Tang. Scalable and adaptively secure any-trust distributed key generation and all-hands checkpointing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2636–2650, New York, NY, USA, 2024. Association for Computing Machinery.

[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.

[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.

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

[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.

A Formal Definitions

In this section, we give formal definitions for the primitives introduced in Section 2.1. For our definitions, recall that $\mathbb{G}$ is a cyclic group of prime order p with independent generators $g, h \in \mathbb{G}$.

Definition 9 (Aggregatable PVSS Scheme)**.** *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:*

- $\mathsf{KGen}(par) \to (\mathsf{ek}, \mathsf{dk})$: *The randomized* key generation algorithm *takes as input system parameters* par*. It outputs a public encryption key* ek *and a secret decryption key* dk*.*
- $\mathsf{Encr}(\mathsf{ek}, m) \to c$: *The randomized* encryption algorithm *takes as input an encryption key* ek *and a message* m*. It outputs a ciphertext* c*. We may also write* $\mathsf{Encr}_{\mathsf{ek}}(m)$ *instead of* $\mathsf{Encr}(\mathsf{ek}, m)$*.*
- $\mathsf{Decr}(\mathsf{dk}, c) \to m$: *The deterministic* decryption algorithm *takes as input a decryption key* dk *and a ciphertext* c*. It outputs a message* m *(optionally with a proof of correct decryption). We may sometimes also write* $\mathsf{Decr}_{\mathsf{dk}}(c)$ *instead of* $\mathsf{Decr}(\mathsf{dk}, c)$*. Further, for all messages* m *and keys* $(\mathsf{ek}, \mathsf{dk}) \in \mathsf{KGen}(par)$*, we require that* $\Pr[\mathsf{Decr}_{\mathsf{dk}}(\mathsf{Encr}_{ek}(m)) = m] = 1$*.*
- $\mathsf{Dist}(\mathsf{ek}_1, \ldots, \mathsf{ek}_n) \to (\vec{E}, \pi)$: *The randomized* secret sharing algorithm *takes as input a list of* n *encryption keys* $(\mathsf{ek}_1, \ldots, \mathsf{ek}_n)$*. It outputs a vector of encrypted shares* $\vec{E} := (\mathsf{Encr}_{\mathsf{ek}_1}(S_1), \ldots, \mathsf{Encr}_{\mathsf{ek}_n}(S_n))$ *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.*
- $\mathsf{Ver}((\vec{E}, \pi), \vec{w}, (\mathsf{ek}_1, \ldots, \mathsf{ek}_n)) \to 0/1$: *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* $(\mathsf{ek}_1, \ldots, \mathsf{ek}_n)$*. 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* $S \in \mathbb{G}$*.*

Definition 10 (Reed-Solomon Code)**.** *A Reed-Solomon code with parameters* (q, b) *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 11 (Cryptographic Accumulator)**.** *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:*

- $\mathsf{Gen}_{\mathsf{Acc}}(n) \rightarrow ak$: *The randomized* accumulator key generation algorithm *takes as input an accumulation threshold n. It outputs a public accumulator key ak.*

- $\mathsf{Eval}_{\mathsf{Acc}}(ak, D) \rightarrow z$: *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.*

- $\mathsf{Witness}_{\mathsf{Acc}}(ak, z, d, D) \rightarrow w/\perp$: *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.*

- $\mathsf{Verify}_{\mathsf{Acc}}(ak, z, d, w, D) \rightarrow 0/1$: *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 12 (Key-Evolving Digital Signature Scheme)**.** *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:*

- $\mathsf{KGen}_{\mathsf{KDS}}(\lambda, T) \rightarrow (\mathsf{pk}, \mathsf{sk}_0)$: *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.

- $\mathsf{Update}_{\mathsf{KDS}}(\mathsf{sk}_r) \rightarrow \mathsf{sk}_{r+1}$: *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 $r+1$.*

- $\mathsf{Sign}_{\mathsf{KDS}}(\mathsf{sk}_r, m) \rightarrow \sigma$: *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.*

- $\mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}, m, \sigma_r) \rightarrow 0/1$: *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 13 (Verifiable Random Function)**.** *A verifiable random function is a tuple of PPT algorithms* $\mathsf{VRF} = (\mathsf{Gen}_{\mathsf{VRF}}, \mathsf{Eval}_{\mathsf{VRF}}, \mathsf{Verify}_{\mathsf{VRF}})$ *with the following properties:*

- $\mathsf{Gen}_{\mathsf{VRF}}(\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$: *The randomized* key generation algorithm *takes as input the security parameter λ. It outputs a public key* pk *and a corresponding secret key* sk.

- $\mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m) \rightarrow (u, \pi)$: *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.*

- $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u, \pi) \rightarrow 0/1$: *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:* $|\Pr[\mathbf{PseudoRand}^{\mathcal{A}}_{\mathsf{VRF}} = 1] - 1/2| \leq \mathsf{negl}(\lambda)$.

Definition 14 (Pseudorandomness for VRF)**.** *Let* $\mathsf{VRF} = (\mathsf{Gen}_{\mathsf{VRF}}, \mathsf{Eval}_{\mathsf{VRF}}, \mathsf{Verify}_{\mathsf{VRF}})$ *be as defined above. For an algorithm $\mathcal{A}$, define the pseudorandomness experiment* $\mathbf{PseudoRand}^{\mathcal{A}}_{\mathsf{VRF}}$ *as follows:*

- 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)**.** *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:*

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

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

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. *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 nonfaulty parties participate in the protocol and that* $\mathsf{valid}(x_i) = 1$ *for every nonfaulty i, the protocol has the following properties:*

- ***Correctness.*** *Every nonfaulty party that completes the protocol outputs the same y_i.*
- ***External Validity.*** *If some nonfaulty party outputs y_i, then* $\mathsf{valid}(y_i) = 1$.

- ***Verifiability.*** *If some nonfaulty party outputs* y_i, π_i*, then* $\mathsf{ConsensusVerify}(y_i, \pi_i) = 1$*. Furthermore, the adversary cannot generate any* y, π *such that* $y \neq y_i$ *and* $\mathsf{ConsensusVerify}(y, \pi) = 1$*.*

- ***Termination.*** *All nonfaulty 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 a nonfaulty party's input), validity (if all nonfaulty parties have the same input, they output that value), or weak validity (if all parties are nonfaulty and have the same input, they output that value). While the protocols we use have these properties, this is not necessary in our construction.