A Survey on Distributed Randomness

Kevin Choi

New York University

1 Introduction

1.1 Perfect Synchrony

Suppose n participants attempt to source a common random number in a distributed yet agreeable manner. In an ideal world where the communication model is perfectly synchronous such that all participants are able to successfully send and receive their random shares s_i (which will make up the common random number) to and from all other participants at the exact same time, each participant can simply add all shares involved and call $s = \sum s_i$ to be the randomness produced in that round. Reminiscent of rock-paper-scissors (which can facilitate quick decision-making in real life), such protocol would be not only simple, but also agreeable by all participants, as there is no room to cheat (e.g. predict the randomness in advance).

While the problem of distributed randomness is trivial in this setting, situations can change dramatically in practical settings where participants can go offline (perhaps temporarily) due to network failure, messages can be delayed either non-significantly or significantly, and Byzantine attackers can try to predict or bias the randomness to their benefit. Namely, consider the following simple scenario without perfect synchrony: three participants (P_1, P_2, P_3) coordinate to produce a common random number $s = s_1 + s_2 + s_3$ where each P_i sends s_i, and suppose P_3 already obtains the knowledge of s_1 and s_2 before sending s_3 to P_1 and P_2 as the last step of the protocol. Then P_3 is able to set s to be whichever value he desires, as he can choose his s_3 depending on s_1 and s_2. Effectively, this protocol becomes centralized by P_3 as a result.

1.2 Commit-Reveal

A strawman solution to the above issue, the classical commit-reveal provides an intuitive way to source a random number from a group of nodes by first allowing each node to commit (in the cryptographic sense) to a secret share and then waiting until all shares are committed before revealing them and summing them to compute the final randomness of a round. Due to the additional commit step, it becomes impossible for one participant to centralize the process and set the final randomness to be whichever value he desires. Nonetheless, note that the problem of biasing still exists, as the last person to reveal his share can in fact check the round's output faster than others and hence can decide not to reveal his number if he doesn't like the round's output. This is called the *last revealer attack*.

1.3 Desired Properties of Distributed Randomness

Clearly, we should prevent anyone from biasing the distributed randomness in any way. Assuming a distributed randomness beacon (DRB) setting in which a distributed randomness protocol is conducted in successive rounds in a continuous manner, we can define overall security of a DRB by requiring the following properties.

Definition 1.1 (d-unpredictability)**.** A DRB protocol is said to be *unpredictable* with absolute bound d for $d \geq 1$ if the probability of an adversary $\mathcal{A}$ predicting the honest output for round $r+d$ (where r denotes the current round) is a negligible function $\mathsf{negl}(\lambda)$.

Definition 1.2 (Secure distributed randomness beacon)**.** A DRB protocol is said to be *d-secure* if it satisfies the following conditions:

1. Bias Resistance. Let O be the output of the beacon for some round r. No computationally bounded adversary $\mathcal{A}$ can bias the output of the beacon, i.e. fix c bits of O for any round $r > 1$ with probability greater than $\frac{1}{2^c} + \mathsf{negl}(\lambda)$.
2. Unpredictability. The protocol satisfies d-unpredictability.
3. Guaranteed Output Delivery. For every round $r \geq 1$, the protocol outputs a value.

4. Public Verifiability. Any third party should be able to verify the beacon values even if not partaking in the generation of those values.

2 Building Blocks

2.1 Verifiable Secret Sharing

The issue with the classical Shamir's secret sharing scheme is that either the dealer or other participants could in fact be acting maliciously, e.g. Alice (a participant) needs to trust that she has received her share correctly from the dealer while she also needs to trust that other participants' revealed shares are correct.

This issue can be fixed by the notion of verifiable secret sharing (VSS). The idea is that we add an additional verification process to the usual Shamir's secret sharing scheme. Here, we take note of two commonly used VSS schemes: Feldman-VSS and Pedersen-VSS.

The mathematical setup is as follows. We choose primes p and q such that $q \mid p-1$ and let g be a generator of G_q, a cyclic subgroup of $\mathbb{Z}_p^*$. This setup has the effect of achieving the following: $a \equiv b \pmod{q} \iff g^a \equiv g^b \pmod{p}$. As a result, it should be understood that modular arithmetics are done in modulo q whenever the numbers involved concern the exponents while in modulo p otherwise. For convenience, we may omit mod p such that one can aptly assume arithmetics are done in modulo p given an equation unless stated otherwise.

2.1.1 Feldman-VSS

The following summarizes a simple VSS scheme (where t among n participants can reconstruct the group secret) proposed by Paul Feldman.

- $f(x) = \sum_{i=0}^{t-1} a_i x^i$ is randomly selected by the dealer, where $a_i \in \mathbb{Z}_q$ and $f(0) = a_0$ is the secret
- The shares are $f(1), f(2), ..., f(n)$ in mod q and are distributed to n participants, respectively
- Also distributed from the dealer are commitments to coefficients of f, i.e. $c_j = g^{a_j}$ for $j = 0, ..., t-1$
- Given her share $f(k)$ and the polynomial coefficient commitments, Alice (a participant) can verify her share by checking:

$$g^{f(k)} = g^{\sum_{i=0}^{t-1} a_i k^i} = \prod_{j=0}^{t-1} c_j^{k^j} = c_0 c_1^k c_2^{k^2} \cdots c_{t-1}^{k^{t-1}}$$

- Any t number of participants (say) $i = 1, 2, ..., t$ can recover the secret a_0 by performing Lagrange interpolation involving Lagrange coefficients $\lambda_i = \prod_{j \neq i} \frac{j}{j-i}$ in mod q:

$$a_0 = f(0) = \sum_{i=1}^{t} f(i)\lambda_i$$

What is new here (compared to Shamir's secret sharing scheme) is the inclusion of commitments to polynomial coefficients in the scheme. These commitments enable participants to verify the validity of their corresponding shares.

2.1.2 Pedersen-VSS

Reminiscent of Pedersen commitment, Pedersen-VSS is a variation that involves two random polynomials generated by the dealer as opposed to one. Namely, the scheme runs as follows.

- $f(x) = \sum_{i=0}^{t-1} a_i x^i$ and $f'(x) = \sum_{i=0}^{t-1} b_i x^i$ are randomly selected by the dealer, where $a_i, b_i \in \mathbb{Z}_q$ and $f(0) = a_0$ is the secret (as before)
- The shares are $(f(1), f'(1)), ..., (f(n), f'(n))$ in mod q and are distributed to n participants, respectively
- Also distributed from the dealer are commitments to coefficients of f and f', i.e. $c_j = g^{a_j} h^{b_j}$ for $j = 0, ..., t-1$
- Given her share $(f(k), f'(k))$ and the polynomial coefficient commitments, Alice (a participant) can verify her share by checking:

$$g^{f(k)} h^{f'(k)} = \prod_{j=0}^{t-1} c_j^{k^j}$$

- Any t number of participants (say) $i = 1, 2, ..., t$ can recover the secret a_0 by performing Lagrange interpolation involving Lagrange coefficients $\lambda_i = \prod_{j \neq i} \frac{j}{j-i}$ in mod q:

$$a_0 = f(0) = \sum_{i=1}^{t} f(i)\lambda_i$$

Effectively, what Pedersen-VSS is able to achieve is the decoupling of g^{a_0} (as the public key corresponding to the secret key a_0) and $g^{a_0} h^{b_0}$ (as the published commitment for verification purposes). In other words, the verification process in which the participants verify their shares does not (even information-theoretically) leak any information regarding the initial secret a_0, a fact that is not true with Feldman-VSS.

2.2 Distributed Key Generation

The motivation for distributed key generation (DKG), which basically comprises n parallel instances of a VSS (run by each participant), is to achieve and utilize a group secret in

a leaderless manner such that any threshold t number of participants should be able to recover the same group secret. The basic idea is that every participant becomes a leader for a VSS (hence achieving the leaderless property), in which case the protocol deals with n random polynomials $\{f_i\}_{i=1,\ldots,n}$ mod q as opposed to one. Though not computed explicitly, the implicit group polynomial $f = \sum f_i$ then embeds the group secret in the form of $f(0)$ (denoted by x) as well as the corresponding public key $g^{f(0)}$ (denoted by y) via commitments to coefficients of f_i as per each VSS. Notably, this facet is the one that allows a leaderless configuration where there does not exist a leader for f even if each participant P_i remains a leader for f_i with the knowledge of $f_i(0)$. As a result, it is only collectively (i.e. with the collaboration of at least t number of nodes in a group) that the group can make use of x and y in the typical sense of asymmetric cryptography (e.g. signing a message) after performing a DKG.

Before delineating the process of a DKG, we include what it means for a DKG to be uniformly secure. Namely, a uniformly secure DKG should satisfy the following three correctness properties and one secrecy property.

Definition 2.1. In the setting where at most $t-1$ nodes can be controlled by an attacker without compromising the protocol, a DKG is *uniformly secure* if the following properties are satisfied.

Correctness

1. Any subset of shares of size t can be used to recover the same group secret key x.
2. All honest parties have access to the same public key $y = g^x$.
3. x is uniformly distributed in $\mathbb{Z}_q$, and thus y is uniformly distributed in G_q (subgroup of $\mathbb{Z}_p^*$ generated by g).

Secrecy

1. Other than from the fact that $y = g^x$, no information on x is leaked. Equivalently, this can be stated using a simulator *SIM*: for every probabilistic polynomial-time adversary $\mathcal{A}$, there exists a probabilistic polynomial-time simulator *SIM* that, given $y \in G_q$ as input, produces an output distribution which is polynomially indistinguishable from $\mathcal{A}$'s view of a DKG protocol that ends with y as its public key output while $\mathcal{A}$ is allowed to corrupt up to $t-1$ participants.

Without the third correctness property, we call a DKG *secure*. The reason behind this minor distinction is that Joint-Feldman DKG, which is most commonly used in practice, is secure but not uniformly secure while Joint-Pedersen offers uniform security. Simultaneously, however, it has been shown that a DKG protocol that is secure but not uniformly secure in fact suffices in the context of using DKG as a subprotocol for distributed randomness, even if the lack of uniform security does seem insufficient a priori in the context of DKG itself.

2.2.1 Joint-Feldman

1. Each participant P_i runs a Feldman-VSS by choosing a random polynomial $f_i(z) = \sum_{j=0}^{t-1} a_{ij} z^j$ and sending a "subshare" $f_i(j)$ to player P_j for all j
2. To satisfy the verifiability portion of the VSS, P_i broadcasts $A_{ik} = g^{a_{ik}}$
3. Upon receiving the subshares and the corresponding commitments (e.g. in the form of a verification vector), P_j can use the verification mechanism per VSS to verify the subshares. If a verification fails, P_j can broadcast a complaint against P_i.
4. If P_i receives at least t complaints, then P_i is disqualified. Otherwise, P_i needs to reveal the subshare $f_i(j)$ per P_j that has broadcasted a complaint. We call *QUAL* the set of non-disqualified players.
5. Once *QUAL* is set, we define $f(z) = \sum_{i \in QUAL} f_i(z) = \sum_{i=0}^{t-1} a_i z^i$ such that each participant P_j in *QUAL* can compute the group public key $y = g^{f(0)} = \prod_{i \in QUAL} A_{i0}$, commitments to f's coefficients $A_k = g^{a_k} = \prod_{i \in QUAL} A_{ik}$, and P_j's share (of the group secret) from subshares $f(j) = \sum_{i \in QUAL} f_i(j)$. Though not computed explicitly, the group secret key x is then equal to both $\sum_{i \in QUAL} a_{i0}$ and the Lagrange interpolation involving the shares $\{f(j)\}_{j \in QUAL}$.

2.2.2 Joint-Pedersen

1. Each participant P_i runs a Pedersen-VSS by choosing two random polynomials $f_i(z) = \sum_{j=0}^{t-1} a_{ij} z^j$ and $f_i'(z) = \sum_{j=0}^{t-1} b_{ij} z^j$ and sending a "subshare" $(f_i(j), f_i'(j))$ to player P_j for all j
2. To satisfy the verifiability portion of the VSS, P_i broadcasts $C_{ik} = g^{a_{ik}} h^{b_{ik}}$
3. Upon receiving the subshares and the corresponding commitments (e.g. in the form of a verification vector), P_j can use the verification mechanism per VSS to verify the subshares. If a verification fails, P_j can broadcast a complaint against P_i.
4. If P_i receives at least t complaints, then P_i is disqualified. Otherwise, P_i needs to reveal the subshare $(f_i(j), f_i'(j))$ per P_j that has broadcasted a complaint. We call *QUAL* the set of non-disqualified players.

5. Once $QUAL$ is set, we define $f(z) = \sum_{i \in QUAL} f_i(z) = \sum_{i=0}^{t-1} a_i z^i$ such that each participant P_j in $QUAL$ can compute the group public key $y = g^{f(0)} = \prod_{i \in QUAL} A_{i0}$, commitments to f's coefficients $A_k = g^{a_k} = \prod_{i \in QUAL} A_{ik}$, and P_j's share (of the group secret) from subshares $f(j) = \sum_{i \in QUAL} f_i(j)$. Though not computed explicitly, the group secret key x is then equal to both $\sum_{i \in QUAL} a_{i0}$ and the Lagrange interpolation involving the shares $\{f(j)\}_{j \in QUAL}$.

2.3 Publicly Verifiable Secret Sharing

While VSS allows verification of the involved shares, the fact of the matter is that such verification can only be done by the involved participants in the secret sharing process. In other words, the verification process of a VSS is not public. Publicly verifiable secret sharing (PVSS), on the other hand, assumes a public setup where the verification process can be achieved publicly (e.g. by bystanders) such that the messages exchanged during the protocol are modified accordingly in order to reflect the fact that some information kept private in a typical VSS can be public in a PVSS. Naturally, this added facet of public verifiability is useful and desirable in models such as the public bulletin board model (e.g. for blockchains). A PVSS can be represented by the following tuple of algorithms.

- $Setup(\lambda) \to pp$. Generates the scheme parameters pp, an implicit input to all other algorithms.
- $KeyGen(\lambda) \to (pk, sk)$. Generate PVSS key-pair (pk, sk) used for share encryption and decryption.
- $Enc(pk, m) \to c$ and $Dec(sk, c) \to m$. The encryption and decryption algorithms used to send shares to all, and obtain private share respectively. The invariant $Pr[Dec(Enc(m)) = m] = 1$ must always hold true for all m in the message domain.
- $ShareGen(s) \to (S, E, \pi)$. Typically, executed by the dealer with secret s to generate secret share vector $S = (s_1, ..., s_n)$ and encryption vector of shares $E = (Enc(s_1), ..., Enc(s_n))$ for all nodes P, and a cryptographic proof π committing to s which guarantees any node with $\geq t$ shares reconstruct a unique s.
- $ShareVerify(E, \pi) \to \{0, 1\}$. Verify if the sharing is correct. A successful verification guarantees that its share is correct and t nodes reconstruct a unique s. 0 indicates a failure whereas 1 indicates a success.
- $Recon(S) \to s$. Reconstruct the shared secret s from the collection of shares $S \subset \{s_1, ..., s_n\}^t$.

Definition 2.2 (PVSS security)**.** Let $L \in P$ be the dealer with secret s and λ be the security parameter. A secure PVSS scheme must provide the following guarantees:

1. **Secrecy.** If L is honest, then the adversary's view during the sharing phase reveals no information about the dealer's secret s with probability better than $\mathsf{negl}(\lambda)$.
2. **Correctness.** If L is honest, then the honest nodes output the secret s at the end of the reconstruction phase with high probability $1 - \mathsf{negl}(\lambda)$.
3. **Public Verifiability.** If the check in verification algorithm returns 1, then with high probability $1 - \mathsf{negl}(\lambda)$, the encryptions are valid shares of some secret.

Implementing the above interface, Schoenmakers' PVSS scheme and Scrape's variations of it are delineated in the following.

2.3.1 Schoenmakers

Initialization. The group G_q and the generators g, G are selected using an appropriate public procedure. Participant P_i generates a private key $x_i \leftarrow \mathbb{Z}_q^*$ and registers $y_i = G^{x_i}$ as its public key.

Distribution. The protocol consists of two steps:

1. *Distribution of the shares.* Suppose without loss of generality that the dealer wishes to distribute a secret among participants $P_1, ..., P_n$. The dealer picks a random polynomial p of degree at most $t-1$ with coefficients in $\mathbb{Z}_q$

$$p(x) = \sum_{i=0}^{t-1} a_i x^i$$

and sets $s = a_0$. The dealer keeps this polynomial secret but publishes the related commitments $C_j = g^{a_j}$ for $0 \leq j < t$. The dealer also publishes the encrypted shares $Y_i = y_i^{p(i)}$ for $1 \leq i \leq n$ using the public keys of the participants. Finally, let $X_i = \prod_{j=0}^{t-1} C_j^{i^j}$. The dealer shows that the encrypted shares are consistent by producing a proof of knowledge of the unique $p(i)$ for $1 \leq i \leq n$ satisfying:

$$X_i = g^{p(i)}, \qquad Y_i = y_i^{p(i)}.$$

The non-interactive proof is the n-fold parallel composition of the protocols for $DLEQ(g, X_i, y_i, Y_i)$. Applying Fiat-Shamir's technique, the challenge c for the protocol is computed as a cryptographic hash of $X_i, Y_i, a_{1i}, a_{2i}, 1 \leq i \leq n$. The proof consists of the common challenge c and the n responses r_i.

2. *Verification of the shares.* The verifier computes $X_i = \prod_{j=0}^{t-1} C_j^{i^j}$ from the C_j values. Using $y_i, X_i, Y_i, r_i, 1 \leq i \leq n$ and c as input, the verifier computes a_{1i}, a_{2i} as

$$a_{1i} = g^{r_i} X_i^c, \qquad a_{2i} = y_i^{r_i} Y_i^c$$

and checks that the hash of $X_i, Y_i, a_{1i}, a_{2i}, 1 \leq i \leq n$ matches c.

Reconstruction. The protocol consists of two steps:

1. *Decryption of the shares.* Using its private key x_i, each participant finds the share $S_i = G^{p(i)}$ from Y_i by computing $S_i = Y_i^{1/x_i}$. They publish S_i plus a proof that the value S_i is a correct decryption of Y_i. To this end, it suffices to prove knowledge of an α such that $y_i = G^\alpha$ and $Y_i = S_i^\alpha$, which is accomplished by the non-interactive version of the protocol $DLEQ(G, y_i, S_i, Y_i)$.

2. *Pooling the shares.* Suppose without loss of generality that participants P_i produce correct values for S_i, for $i = 1, ..., t$. The secret G^s is obtained by Lagrange interpolation:

$$\prod_{i=1}^{t} S_i^{\lambda_i} = \prod_{i=1}^{t} \left(G^{p(i)}\right)^{\lambda_i} = G^{\sum_{i=1}^{t} p(i)\lambda_i} = G^{p(0)} = G^s$$

where $\lambda_i = \prod_{j \neq i} \frac{j}{j-i}$ is a Lagrange coefficient.

Note that the participants do not need nor learn the values of the exponents $p(i)$. Only the related values $S_i = G^{p(i)}$ are required to complete the reconstruction of the secret value $S = G^s$. Also, note that participant P_i does not expose its private key x_i; consequently participant P_i can use its key pair in several runs of the PVSS scheme. The type of encryption used for the shares has been optimized for performance; however, if desired, it is also possible to use standard ElGamal encryption instead.

Clearly, the scheme is homomorphic. For example, given the dealer's output for secrets G^{s_1} and G^{s_2}, the combined secret $G^{s_1+s_2}$ can be obtained by applying the reconstruction protocol to the combined encrypted shares $Y_{i1}Y_{i2}$.

2.3.2 Scrape's Variations

Scrape offers two variations of Schoenmakers' PVSS: one that relies on the DDH (decisional Diffie-Hellman) assumption in the random oracle model and one that relies on the DBS (decisional bilinear square) assumption in the plain model. Accordingly, the former utilizes the same NIZK (i.e. *DLEQ* by Chaum and Pedersen) from Schoenmakers' whereas the latter utilizes a bilinear pairing, which serves as a valid size-efficient substitute to the NIZK at the cost of pairing-related computations. Both variations make novel use of Reed-Solomon codes from coding theory in a way that the total number of exponentiations needed to verify the shares distributed by the dealer in a PVSS is decreased from $O(nt)$ (which becomes quadratic if $t = \frac{n}{2}$ for instance) to $O(n)$. In other words, the approaches optimize the verifier's computational complexity before the Reconstruction phase.

The idea is that the verifier does not have to compute t exponentiations via $X_i = \prod_{j=0}^{t-1} C_j^{i^j}$ per share, before which the dealer publishes the commitments C_j to the polynomial coefficients. Instead, the dealer publishes $v_i = g^{p(i)}$ directly alongside the dual code $\mathcal{C}^\perp$ of the $[n, k, n-k+1]$ Reed-Solomon code $\mathcal{C}$ corresponding to the shares calculated by the dealer in the form $\mathcal{C} = \{(p(1), ..., p(n)) \mid p(x) \in \mathbb{Z}_q[x], \deg(p(x)) \leq t-1\}$, in which case the verifier can randomly sample a codeword $c^\perp = (c_1^\perp, ..., c_n^\perp)$ from $\mathcal{C}^\perp$, indirectly compute the inner product of $c^\perp$ and the share vector $(s_1, ..., s_n)$, and verify that $\prod v_i^{c_i^\perp} = g^{\sum s_i c_i^\perp} = g^0 = 1$ with $O(n)$ exponentiations.

2.4 Verifiable Delay Function

A verifiable delay function (VDF) is a function that takes a prescribed time to compute, even on a parallel computer. However once computed, the output can be quickly verified by anyone. Moreover, every input must have a unique valid output as per a function. A VDF can be represented by a tuple of three algorithms:

- $Setup(\lambda, T) \rightarrow pp$ is a randomized algorithm that takes a security parameter λ and a time bound T and outputs public parameters pp.
- $Eval(pp, x) \rightarrow (y, \pi)$ takes an input x and outputs y and a proof π.
- $Verify(pp, x, y, \pi) \rightarrow \{accept, reject\}$ outputs $accept$ if y is the correct evaluation of the VDF on input x.

A VDF must satisfy the following three properties:

- ε-evaluation time. $Eval(pp, x)$ runs in time at most $(1+\varepsilon)T$, for all x and all pp output by $Setup(\lambda, T)$.
- Sequentiality. A parallel algorithm $\mathcal{A}$, using at most $poly(\lambda)$ processors, that runs in time less than T cannot compute the function. Specifically, for a random x and pp output by $Setup(\lambda, T)$, if $(y, \pi) \leftarrow Eval(pp, x)$ then $Pr[\mathcal{A}(pp, x) = y]$ is negligible.
- Uniqueness. For an input x, exactly one y will be accepted by $Verify$ with negligible error probability. Specifically, let $\mathcal{A}$ be an efficient algorithm that given pp as input, outputs (x, y, π) such that $Verify(pp, x, y, \pi) = accept$. Then $Pr[Eval(pp, x) \neq y]$ is negligible.

Intuitively, one can understand VDF as a computational task that cannot be sped up by parallelism as a result. Exponentiation in a group of unknown order is believed to have this property and was used by Rivest, Shamir, and Wagner to construct a time-lock puzzle. The two well-regarded VDF proposals, i.e. one due to Pietrzak and the other due to Wesolowski, similarly make use of the serial nature of this task.

3 Protocols

3.1 PVSS-based

3.1.1 Scrape

Scrape is run between n parties $P_1, ..., P_n$ who have access to a public ledger (a la public bulletin board model) where

they can post information for later verification. At a high level, the protocol is an improvement over commit-reveal such that the concept of reconstruction can prevent the last revealer attack in case any committed inputs are not revealed later. A PVSS protocol is used as a subprotocol, and it is assumed that the Setup phase is already done and the public keys pk_i of each party P_i are already registered in the ledger. In detail, the protocol proceeds as follows:

1. **Commit:** For $1 \leq j \leq n$, party P_j executes the Distribution phase of the PVSS subprotocol as the Dealer with threshold $t = \frac{n}{2}$, publishing the encrypted shares and the verification information $PROOF_D^j$ on the public ledger, and also learning the random secret h^{s^j} and s^j. P_j also publishes a commitment to the secret exponent $\mathsf{Com}(s^j, r_j)$ (with fresh randomness $r_j \leftarrow \mathbb{Z}_q$), for $1 \leq j \leq n$.

2. **Reveal:** For every set of encrypted shares and the verification information $PROOF_D^j$ published in the public ledger, all parties run the Verification phase of the PVSS subprotocol. Let $\mathcal{C}$ be the set of all parties who published commitments and valid shares. Once $\frac{n}{2}$ parties have posted their commitments and valid shares on the ledger, party P_j opens its commitment, posting $\mathsf{Open}(s^j, r_j)$ on the ledger, for $j \in \mathcal{C}$.

3. **Recovery:** For every party $P_a \in \mathcal{C}$ that does not publish $\mathsf{Open}(s^a, r_a)$ in the Reveal phase, party P_j runs the Reconstruction phase of the PVSS protocol posting $\tilde{s}_j^a$ and $PROOF_j^a$ to the public ledger, for $1 \leq j \leq n$. Once $\frac{n}{2}$ valid decrypted shares are published, every party reconstructs h^{s^a}.

The final randomness is computed as $\rho = \prod_{j \in \mathcal{C}} h^{s^j}$.

3.1.2 HydRand

HydRand can be interpreted as an improvement over Scrape, where at most one operation of PVSS distribution or reconstruction is conducted every round such that we are able to reduce the protocol's communication complexity by a factor of n, i.e. from $O(n^3)$ to $O(n^2)$. Basically, this is made possible by introducing the concept of a round leader (which is nonexistent in the case of Scrape) such that one of the following two scenarios is to happen every round: the round leader reveals his secret committed in his previous round (i.e. the last time he was the round leader as per some predefined, deterministic leader selection mechanism) and commits to a new secret to be utilized in a future round by distributing the corresponding PVSS shares via a message of size $O(n)$ (hence at most one PVSS distribution), or the leader fails to reveal his secret committed in his previous round in a timely manner, in which case the non-leader nodes are able to reconstruct the secret by aggregating the shares that are received previously (hence at most one PVSS reconstruction). This indeed contrasts with Scrape, where there are $O(n)$ fresh operations of PVSS distribution and reconstruction every round due to the fact that each node has to perform one each round.

Again, the difference is that PVSS shares distributed by a leader in HydRand are used not in the current round (which is the case for Scrape), but the next time the same leader is selected in some future round. Combined with the fact that HydRand's randomness each round is defined as the hash of the previous round's randomness as well as the round leader's secret corresponding to the current round (but committed previously), this can potentially cause the beacon's unpredictability property to fail without any remedy, as the leader selection mechanism can theoretically select only the adversarial nodes as the round leaders such that it becomes possible for the adversarial nodes to predict the entire randomness chain in advance.

To remedy this, HydRand takes the approach that a round leader is not allowed to be selected again for the next f rounds (where f denotes the number of Byzantine nodes). As a result, the protocol guarantees that there exists at least one honest node in $f+1$ consecutive rounds such that it is impossible to achieve predictability for up to $f+1$ rounds. Nonetheless, it remains possible to achieve predictability for up to f rounds with a low probability whenever f Byzantine nodes are consecutively selected as leaders, and this remains a tradeoff incurred by HydRand.

Employing Scrape's PVSS as a subprotocol, HydRand's rounds are implemented in three distinct phases: propose, acknowledge, and vote.

I. Propose. During this phase, the round leader ℓ reveals his previously committed value s_ℓ and provides a new commitment $Com(s_\ell^*)$. For this purpose, it is the leader's task to propose a new dataset D_r for the current round r. As a performance optimization, we split a dataset into two parts: a header and a body. For certain operations, we only require sending the header of the dataset. The header $header(D_r)$ of dataset D_r contains:

- the hash of the dataset's body $H(body(D_r))$
- the current round index r
- the round's random beacon value R_r
- the revealed secret value s_ℓ
- the round index $\tilde{r}$ of the previous dataset $D_{\tilde{r}}$
- the hash $H(D_{\tilde{r}})$ of the previous dataset $D_{\tilde{r}}$ if $\tilde{r} > 0$
- a list of random beacon values $\{R_k, R_{k+1}, ...\}$ for all recovered rounds between $\tilde{r}$ and r (if any)
- the Merkle tree root hash M_r over all encrypted shares in the new commitment $Com(s_\ell^*)$

Then the body $body(D_r)$ of dataset D_r contains:

- a confirmation certificate $CC(D_{\tilde{r}})$ which confirms that $D_{\tilde{r}}$ was previously accepted as a valid dataset
- a recovery certificate $RC(k)$ for all rounds $k \in \{\tilde{r}+1, \tilde{r}+2, ..., r-1\}$ which confirms that there exists a recovery for all rounds between $\tilde{r}$ and r
- the commitment $Com(s^*_\ell)$ to a new randomly chosen secret s^*_ℓ

Note that the leader selects $\tilde{r} < r$ as the most recent regular round, for which the leader is not aware of any successful recovery.

After the construction of the above dataset, a correct leader ℓ broadcasts a signed *propose* message to all nodes. Each node i, which receives such a message from the leader before the end of the propose phase, checks the validity of the dataset D_r. For this purpose i verifies that D_r is constructed as previously defined and properly signed. This includes a check that the revealed secret s_ℓ corresponds to the previous commitment of the current leader. Additionally, the validity of the confirmation and recovery certificates is checked. A *confirmation certificate* for dataset $D_{\tilde{r}}$ is valid iff it consists of $f+1$ signed messages from $f+1$ different senders. Similarly, a *recovery certificate* for some round k is a collection of $f+1$ signed messages from $f+1$ different senders.

II. Acknowledge. If a node i receives a valid dataset D_r from the round's leader ℓ during the propose phase, it constructs and broadcasts a signed acknowledge message thereby also forwarding the revealed secret value s_ℓ as part of the header. Further, each node i collects and validates acknowledge messages from other nodes.

III. Vote. Each node i checks the following conditions:

- During the current propose phase a valid dataset D_r was received.
- During the current acknowledge phase at least $2f+1$ valid acknowledge messages from different senders have been received.
- All acknowledge messages received refer to the dataset's hash $H(D_r)$. Valid acknowledge messages for more than one value of $H(D_r)$ form a cryptographic proof of leader equivocation.

If all conditions are met, node i broadcasts a signed confirmation message. Otherwise, node i broadcasts a recover message. Note that a Merkle tree root hash is required to enable nodes which are not in possession of $Com(s_\ell)$ to verify the share decryption proof. R_{r-1} is included for efficient external verification.

At the end of this phase, each node i can obtain the round's random beacon value R_r. We distinguish between the following two cases: (i) node i already knows the secret value s_ℓ, because it received the dataset D_r or an acknowledge message for D_r, and (ii) node i has received at least $f+1$ valid recover messages which include at least $f+1$ decrypted secret shares for s_ℓ. In the latter case, the reconstruction procedure of Scrape's PVSS can be executed to produce the value h^{s_ℓ}. In both cases, R_r is then obtained by computing:

$$R_r \leftarrow H(R_{r-1}||h^{s_\ell})$$

3.1.3 RandPiper

RandPiper offers two protocols: GRandPiper (Good Pipelined Random beacon) and BRandPiper (Better Pipelined Random beacon). While GRandPiper roughly follows HydRand's protocol, it improves upon it via its idiosyncratic use of BFT SMR (state machine replication) protocol, which uses erasure coding and cryptographic accumulators (either of the bilinear type incurring a trusted setup tradeoff or of the Merkle tree type) to implement the public bulletin board model in a Byzantine broadcast fashion such that HydRand's worst-case communication complexity of $O(n^3)$ is improved to $O(n^2)$ in GRandPiper.

The key idea is that RandPiper's SMR protocol allows a round leader to Byzantine broadcast a message of size $O(n)$ within a total communication complexity of $O(n^2)$ rather than $O(n^3)$. While this improvement benefits the side of communication complexity indeed, the same issue of predictability present in HydRand still remains in GRandPiper, where an adversary can predict up to f number of rounds (albeit with low probability that becomes trivial when n is large). The trusted setup assumption due to the initial q-SDH parameters is also a tradeoff. While using Merkle trees as accumulators as opposed to bilinear ones would eliminate the trusted setup assumption, Merkle trees increase the total communication complexity from $O(n^2)$ to $O(n^2 \log n)$.

The following delineates RandPiper's SMR protocol, which happens largely in 4 steps (preceded by an epoch advancement) per epoch (i.e. round): propose, vote, vote certificate, and commit.

0. Epoch advancement. When $\mathsf{epoch\text{-}timer}_{e-1}$ expires, node p_i enters epoch e. After entering epoch e, node p_i sends its highest ranked certificate to leader L_e. In addition, it also sets $\mathsf{epoch\text{-}timer}_e$ to 11Δ and starts counting down.

1. Propose. After entering epoch e, leader L_e waits for 2Δ to collect highest ranked certificates from all honest nodes. The 2Δ wait before a proposal ensures that leader L_e can collect highest ranked certificates from all honest nodes when leader L_e enters epoch e Δ time before other honest nodes. After the 2Δ wait, leader L_e proposes a block B_h by extending the highest ranked block certificate known to L_e. The proposal

for B_h, conceptually, has the form $\langle \text{propose}, B_h, e, \mathcal{C}, z_{pe} \rangle_{L_e}$ where z_{pe} is the accumulation value for the pair $(B_h, \mathcal{C})$. In order to facilitate efficient equivocation checks, the leader signs the tuple $\langle \text{propose}, H(B_h, \mathcal{C}), e, z_{pe} \rangle_{L_e}$ and sends B_h and $\mathcal{C}$ separately. This size of the signed message is $O(k)$ and hence can be broadcast during equivocation or while delivering p_e without incurring cubic communication overhead.

2. Vote. If a node receives the first valid proposal p_e when $\mathsf{epoch\text{-}timer}_e \geq 7\Delta$ and block B_h extends the highest ranked certificate known to the node, it invokes $\mathsf{Deliver}(\text{propose}, p_e, z_{pe}, e)$. In addition, the node sets its $\mathsf{vote\text{-}timer}_e$ to 2Δ and starts counting down. When $\mathsf{vote\text{-}timer}_e$ reaches 0 and detects no epoch e equivocation, the node sends vote to L_e. If block B_h does not extend the highest ranked certificate known to the node or receives proposal p_e when its $\mathsf{epoch\text{-}timer}_e < 7\Delta$, the node simply ignores the proposal and does not vote for B_h.

3. Vote certificate. When leader L_e receives $t+1$ vote messages for the proposed block, L_e broadcasts $\mathsf{vote\text{-}cert}$ to all nodes. Similar to the proposal, the hash of the certificate is signed to allow for efficient equivocation checks. It is important to note that two different certificates for the same value is still considered an equivocation in this step.

4. Commit. When node p_i receives the block certificate v_e when epoch timer is large enough (3Δ), it invokes $\mathsf{Deliver}(\text{vote-cert})$ and sets $\mathsf{commit\text{-}timer}_e$ to 2Δ. When $\mathsf{commit\text{-}timer}_e$ reaches 0, if no equivocation for epoch e has been detected, node p_i commits B_h and all its ancestors. The $\mathsf{Deliver}()$ message ensures that all honest nodes have received $\mathcal{C}$ before quitting epoch e. We note that it is not necessary for node p_i to vote for B_h to commit it. A block certificate on B_h implies that at least one honest node voted for B_h and if node p_i had a different highest ranked certificate due to which it did not vote, then the corresponding block was not committed.

A. Equivocation. At any time in epoch e, if a node p_i detects an equivocation, it broadcasts equivocating message signed by L_e. As mentioned earlier, the message signed by L_e are $O(1)$ sized and does not incur large communication. Node p_i also stops participating in epoch e at that time.

B. Deliver function. The $\mathsf{Deliver}()$ function implements efficient broadcast of long messages using erasure coding techniques and cryptographic accumulators. When the function is invoked using the input parameters, the message b is first divided into n coded shares $(s_1, ..., s_n)$ using Reed-Solomon codes and a cryptographic witness w_j is computed for each value s_j. Then, the share (s_j, w_j) is sent to the node $p_j \in P$ along with the accumulation value z_e, keyword key, and L_e's signature on the message. When a node p_j receives the first valid shares for an accumulation value z_e such that the witness w_j verifies the share s_j, it forwards the share (s_j, w_j) to all nodes. When a node p_i receives $n-t$ valid shares corresponding to the first accumulation value z_e it receives, it reconstructs the object b. Note that node p_i reconstructs object b for the first valid share even though it detects equivocation in an epoch.

While GRandPiper is essentially an improvement over HydRand employing the above SMR protocol, BRandPiper offers a slightly different approach to distributed randomness, motivated to provide a guarantee of absolute unpredictability rather than probabilistic unpredictability. The key difference is that BRandPiper makes use of n secrets (one from each node) per round as opposed to one (from the round leader) like GRandPiper or HydRand. Similar to Scrape, this in fact requires n^2 secret shares (n shares per secret) per round. To perform better than Scrape's quartic worst-case communication complexity, however, BRandPiper fine-tunes its public bulletin board during the protocol such that the number of messages posted on the public bulletin board (realized by the BFT SMR protocol) is minimized.

The main idea is that a leader commits to n secrets (corresponding to the next n rounds) in one round at once as opposed to committing to one secret on a per-round basis. It is in this way that each node (leader or non-leader) should have access to n secret shares (each corresponding to a different secret, one from each node) per round such that a homomorphic sum of those shares followed by a reconstruction phase with other nodes should yield the randomness shared by the entire group in that round a la DKG. Nonetheless, the tradeoff is an additional multiplicative factor of $O(f)$ in total communication complexity, i.e. $O(fn^2)$, which becomes cubic in the worst case when $f = O(n)$ but remains quadratic in the best case when $f = O(1)$.

3.1.4 SPURT

While similar to BRandPiper in achieving absolute unpredictability (rather than probabilistic) by utilizing n secrets (whose sum becomes the group secret) per round, SPURT takes a different approach where those n secrets every round are taken not from a leader (from any rounds including previous rounds), but from all the non-leader nodes in a round. A leader, on the other hand, effectively functions to facilitate the overall communication among nodes, lowering the total communication complexity from that in Scrape where all nodes need to broadcast messages of size $O(n)$ every round.

Given a round leader selected in a round-robin fashion, SPURT proceeds in four phases: commitment, aggregation, agreement, and reconstruction.

1. Commitment. During the commitment phase, each node chooses a uniformly random secret and computes shares for

the chosen secret using PVSS. Each node then sends all these shares to the leader L_r. Here on, we can denote the messages sent by nodes to L_r by PVSS tuples of round r. Despite having access to PVSS messages from all nodes, L_r cannot break unpredictability of SPURT. Intuitively, this is because each share is encrypted using the public key of the recipient node, and a zero-knowledge scheme is used for proving consistency.

2. Aggregation. In the aggregation phase, L_r upon receiving PVSS tuples from a node, validates them. Upon receiving and validating PVSS messages from $t+1$ nodes, L_r aggregates them using the additive homomorphic property of the underlying polynomial commitment and encryption schemes. If $p_1, p_2, ..., p_{t+1}$ are the underlying polynomials from the $t+1$ valid polynomial commitments, L_r aggregates them to obtain the commitment to the aggregated polynomial $\tilde{p}(x) = \sum_{j=1}^{t+1} p_j(x)$. Moreover, L_r aggregates the $t+1$ encrypted shares to obtain encrypted shares corresponding to the aggregated polynomial $\tilde{p}(x)$.

3. Agreement. After aggregation, L_r computes a cryptographic digest of the commitment of the aggregated polynomial $\tilde{p}(x)$, the identities (or indices) of nodes whose polynomial are aggregated into $\tilde{p}(x)$, and the encrypted shares of the secret embedded in $\tilde{p}(x)$. L_r then sends this cryptographic digest to all of the nodes via a broadcast channel. Note that this theoretical concept of a broadcast channel is realized by some SMR protocol.

Additionally, to each node i, L_r sends node i the entire commitment to the aggregated polynomial $\tilde{p}(x)$, and the encrypted shares corresponding to $\tilde{p}(x)$ using the pair-wise channel between i and L_r. Moreover, L_r also sends the encrypted shares for i of the original $t+1$ polynomials aggregated into $\tilde{p}(x)$, and the corresponding NIZK proofs. Note that these shares are encrypted under the public key of i. In total, during the agreement phase, L_r sends $O(\lambda)$ bits of data via the broadcast channel and $O(\lambda n)$ bits of data to each node using pair-wise private channels.

Each node i, upon receiving the cryptographic digest over the broadcast channel and private messages from L_r, validates them to ensure that L_r did the aggregation phase correctly. For this step, node i relies on the properties of linear error-correcting code and NIZK proofs forwarded by L_r. Upon successful validation, node i starts the reconstruction phase. Else, node i moves to the next round with the next leader and the cycle continues.

4. Reconstruction. When the agreement phase terminates, i.e. all honest nodes agree on the cryptographic digest broadcast by L_r, every honest node who received the valid shares from L_r multicasts its aggregated share along with the NIZK proof of its correctness. If the agreement phase terminates successfully, then at least $t+1$ honest nodes hold valid shares of the aggregated polynomial $\tilde{p}(x)$. Also, all nodes will be able to prove the correctness of their aggregated shares. Moreover, all these nodes start the reconstruction phase within three messages transmission delays. Hence, during the reconstruction phase, every honest node will receive at least $t+1$ valid shares of $\tilde{p}(x)$, along with their correctness proofs. As a result, every honest node will be able to successfully reconstruct the polynomial via $h^{\tilde{p}(x)}$, and hence the output of the randomness beacon for this round.

Besides the fact that SPURT takes a different approach (secrets are received from non-leader nodes rather than leader nodes) to achieve the same goal (absolute unpredictability via n secrets per round) as BRandPiper, it is noteworthy that bits of data that are broadcasted (as opposed to sent privately) are minimized, i.e. $O(\lambda)$ instead of $O(\lambda n)$, in SPURT. As a result, the total communication complexity is $O(\lambda n^2)$ in the best case. In the worst case, the additional cost in communication complexity can be incurred during the reconstruction phase, e.g. when some honest nodes might not receive the correct messages from the round leader, in which case other non-leader nodes might have to be queried for a Merkle path of size $O(\log n)$ and the list of signers of size $O(n)$. Thus, the worst-case communication complexity is $O(\lambda n^2 \log n + n^3)$.

3.2 DVRF-based

Distributed verifiable random function (DVRF) is a distributed version of VRF, where the VRF's secret key is distributed among a group of participants. In the current setting, such distribution can be done via DKG such that a threshold t number of participants is needed to recover the DVRF output from t DVRF shares. Then the idea is that each DVRF output from a DVRF phase (preceded by a DKG phase) is defined to be each round's randomness, where the input to the DVRF can be a hyperparameter (e.g. which can be set to the current timestamp, DVRF output from the previous round, etc.) to the protocol.

Note that DVRF generalizes the idea of unique (i.e. at most one signature is accepted by the verification algorithm for every message and public key) threshold signature schemes such that protocols like Dfinity (which involves threshold BLS) can be bucketed under this interpretation. The reason is that a unique digital signature is a VUF (verifiable unpredictable function), which can be hashed to yield a VRF in the random oracle model.

3.2.1 drand

drand and Dfinity adopt a similar mechanism to output random numbers. As outlined above, the idea is the following. After the DKG phase, nodes enter the beacon phase, each round of which involves each node creating a signature on the common message (i.e. derived from the previous round's beacon output) under its corresponding secret share. Then

these individual signatures are interpolated into one group signature (verifiable by group public key) via Lagrange interpolation, after which the group signature is hashed to yield the round's output. BLS signature scheme is used throughout.

Afterwards, nodes proceed to the next round and reiterate the above process such that the resulting architecture of the protocol is basically a chain of DVRFs.

3.2.2 DDH-DVRF

3.3 VDF-based

Lastly, one could devise a distributed randomness beacon using VDFs. As aforementioned, the idea of VDF is that computing the function takes time while verifying it can happen quickly. It is precisely this aspect that can prevent the last revealer attack in a typical commit-reveal. Additionally, VDFs can be employed to extend some preexisting public randomness that can be manipulated into one that cannot be manipulated, or to construct an entirely new protocol (like RandRunner) altogether. These three approaches are summarized in the following.

3.3.1 Extending Commit-Reveal

In commit-reveal (like RANDAO), the issue is that the last revealer is able to compute the final output earlier than others and hence can choose to not reveal his share to his benefit in order to bias the randomness. The idea to prevent this biasing attack via VDF is simple. Namely, the protocol could make it a requirement that the output from the reveal phase of the protocol has to be run through a VDF before declaring it as the final randomness. Because of the fact that computing a VDF takes substantial time, the last revealer would not be able to precompute the final randomness of the round before others and hence would have to conform to the protocol (i.e. cooperate by revealing his share that was previously committed).

3.3.2 Extending Public Randomness

VDFs can also be utilized to extend a preexisting public source of randomness that by itself could be subject to manipulation. Namely, we can consider some closing stock price (or some block hash of Bitcoin) as our unextended randomness, in which case it is possible for malicious traders (or Bitcoin miners) to temporarily manipulate the market (or the Bitcoin network via block withholding attack) in order to manipulate the randomness. Similar to applying VDF to commit-reveal, the solution to this problem is to run the output through a VDF before declaring it as our final randomness. In other words, the output from the final VDF layer is defined to be the randomness of the protocol as opposed to the original output (closing price or block hash) itself. The reason is that a malicious actor would not have enough time to decide whether or not to manipulate the protocol in the first place due to the fact that VDF computation must take substantial time.

3.3.3 RandRunner

RandRunner represents a novel way to utilize VDFs (aside from simply attaching a VDF layer to some preexisting protocol) to derive distributed randomness. Namely, it involves a chain of trapdoor VDFs with the strong uniqueness property. A trapdoor VDF is a VDF where the initial Setup algorithm also outputs a trapdoor (secret key) *sk* such that the party initiating it is able to use the trapdoor to compute the VDF in a more timely manner compared to others. Anyone knowing *sk* can in fact achieve the same. On the other hand, strong uniqueness not only implies uniqueness (which is true by definition for a VDF), but also uniqueness given the possibility of malicious generation of VDF's public parameters by an adversary – a property that holds true in the case of Pietrzak's VDF but not Wesolowski's.

Definition 3.1 (Trapdoor verifiable delay function)**.** A verifiable delay function is a *trapdoor verifiable delay function* if a slight modification to $Setup$ and an addition of $TrapdoorEval$ are made as follows.

- $Setup(\lambda, T) \rightarrow (pp, sk)$ is a randomized algorithm that takes a security parameter λ and a time bound T and outputs public parameters pp and a trapdoor secret key sk.
- $Eval(pp, x) \rightarrow (y, \pi)$ takes an input x and outputs y and a proof π.
- $TrapdoorEval(sk, pp, x) \rightarrow (y, \pi)$ takes an input x with the knowledge of trapdoor sk and outputs y and a proof π such that the algorithm takes less than time T to complete unlike $Eval$.
- $Verify(pp, x, y, \pi) \rightarrow \{accept, reject\}$ outputs $accept$ if y is the correct evaluation of the VDF on input x.

Definition 3.2 (Strong uniqueness)**.** For each input x and all public parameters pp, exactly one y will be accepted by $Verify$ with negligible error probability even if the public parameters have been adversarially generated. Specifically, let $\mathcal{A}$ be an efficient algorithm that outputs (pp, x, y, π) such that $Verify(pp, x, y, \pi) = accept$. Then $Pr[Eval(pp, x) \neq y]$ is negligible.

Achieving the strong uniqueness property with Pietrzak's VDF as a result, RandRunner comprises a chain of VDFs such that there are n VDFs involved given n participants (each offering a VDF that is independently set up). The chain is essentially an "interleaved" (or juxtaposed) one involving n VDFs where each round of randomness is generated by one VDF, led by one node that leads the round, among n VDFs. In other words, the protocol proceeds with a leader

corresponding to a VDF on a per-round basis. A leader can be selected either in a round-robin style (i.e. taking turns in some permuted order) or in a way that involves randomized sampling (i.e. based on the previous round's randomness output).

The idea is that each round, the leader node computes its VDF using $TrapdoorEval(sk, pp, x)$ with the knowledge of sk that corresponds to the VDF (whose $Setup$ was originally run by the very node) and broadcasts the resulting pair (y, π) to all other nodes, which then perform $Verify$ to either accept or reject the supposed evaluation of the VDF from the leader node. The input x is a function of the previous round's randomness output. Given that the protocol outputs R_r in round r, in other words, input x is equal to $H_{in}(R_{r-1})$ in round r where H_{in} denotes a hash function mapping R_{r-1} to the VDF's input space.

Input and output values of round r in RandRunner

- $x = H_{in}(R_{r-1})$
- $y = TrapdoorEval(sk, pp, x)$ if a node is the round leader or otherwise receives (y, π) from the round leader where $Verify(pp, x, y, \pi) = accept$
- $y = Eval(pp, x)$ if a node is a non-leader not having received any message from the round leader
- $R_r = H_{out}(y)$

Then the output of round r is the hash (denoted by H_{out}) of y such that the relationship $R_r = H_{out}(y) = H_{out}(TrapdoorEval(sk, pp, H_{in}(R_{r-1}))[0])$ holds true for any r in the default case of the protocol (where the notation [0] denotes the first element of a tuple). In the non-default case, what could happen is that the leader node may fail to successfully broadcast the pair (y, π) to all nodes, e.g. due to network failure or intentionally, in which case the non-leader nodes compute $Eval(pp, x)$ to achieve it. Notably, this can create a potential asymmetry between the leader and non-leaders as the leader node can compute R_r substantially before others using $TrapdoorEval$. Due to the fact that the leader changes every round (with certainty in the round-robin style RandRunner whereas with high probability in the randomized sampling RandRunner), however, such temporal asymmetry is remedied as long as the number of Byzantine nodes $f < n/2$, allowing the protocol to achieve unpredictability to a reasonable extent though not as optimally as other aforementioned VDF-based constructions. Details regarding unpredictability as well as the protocol's algorithm itself can be found in the appendix.