SPURT: Scalable Distributed Randomness Beacon with Transparent Setup

Sourav Das
University of Illinois at Urbana-Champaign
souravd2@illinois.edu

Vinith Krishnan
University of Illinois at Urbana-Champaign
vinithk2@illinois.edu

Irene Miriam Isaac
University of Illinois at Urbana-Champaign
irenemi2@illinois.edu

Ling Ren
University of Illinois at Urbana-Champaign
renling@illinois.edu

ABSTRACT

Having shared access to high-quality random numbers is essential in many important applications. Yet, existing constructions of distributed random beacons still have limitations such as imperfect security guarantees, strong setup or network assumptions, or high costs. In this paper, we present SPURT, an efficient distributed randomness beacon protocol that does not require any trusted or expensive setup and is secure against a malicious adversary that controls up to one-third of the nodes in a partially synchronous network. We formally prove that each output of SPURT is unpredictable, bias-resistant, and publicly verifiable. SPURT has an amortized total communication cost of $O(\lambda n^2)$ per beacon output in the fault-free case and $O(\lambda n^2 \log n + n^3)$ in the worst case. We implement SPURT and evaluate it using a network of up to 128 nodes running in geographically distributed AWS instances. Our evaluation shows that SPURT has practical computation and bandwidth costs and can produce beacon outputs every second for a network of 64 nodes, and every 3 seconds for a network of 128 nodes.

1 INTRODUCTION

A reliable source of a continuous stream of shared randomness, also referred to as a *random beacon*, is crucial for many distributed protocols. Applications of random beacon include leader election in proof-of-stake based blockchains [5, 40], blockchain sharding [9, 47, 50, 68], scaling smart contracts [30], annonymous communications [7, 41, 65, 66], solving consensus under asynchrony [36], anonymous browsing [32, 39, 43], publicly auditable auctions and lottery [20], electronic voting [8], cryptographic parameter generations [11, 49], and so on.

The simplest approach is to rely on a single node or organization, such as NIST random beacon or Random.org, to produce the required randomness. This is undesirable or even unreasonable in some scenarios. Incidents such as the backdoor of Dual elliptic curve pseudorandom number generator [14] and 1969 US conscription lottery [1] present arguments against centrally generated randomness. Moreover, in systems such as blockchains where the main objective is to remove centralized authorities, using a trusted party for randomness generation defeats the purpose of the blockchain itself.

A natural approach to remove the trusted third party is to decentralize the process of generating randomness among many nodes using a distributed protocol. This approach guarantees that, as long as a large fraction (majority or supermajority) of nodes faithfully follow the protocol, the protocol will produce shared randomness with desired properties [57]. Briefly, any randomness beacon protocol should be *available* and each beacon output should be *unpredictable, bias-resistant* and *publicly verifiable*. Informally, *unpredictability* guarantees that neither adversary nor honest nodes can compute any non-trivial information about future beacon outputs. Similarly, *bias-resistance* ensures that beacon outputs are sampled from a uniform distribution and are independent of other beacon outputs. Lastly, *public verifiability* enables external clients, i.e., users that do not participate in the beacon generation protocol to validate the correctness of every beacon output.

Existing works. Starting from Blum's two-node coin tossing protocol [17], a long line of works have looked into the problem of generating shared randomness under different system models [2, 10, 17, 22–24, 31, 40, 46, 55, 61, 64]. Due to its use in practical blockchain systems, which typically involves a large number of nodes [30, 47, 50, 68], recent randomness beacon protocols put an emphasis on scalability. Specifically, it is desirable to construct a beacon protocol that has low latency, low communication complexity, and low computation cost per node per beacon output. Also, since many of these protocols are decentralized and are aimed at eliminating trusted entities, it is preferable that the beacon protocol does not rely on a trusted setup.

Despite decades of research and many breakthroughs, state-of-the-art distributed randomness beacon protocols still have scalability issues, require strong cryptographic or network assumptions, or do not provide the full suite of desired properties of a randomness beacon. For example, the random outputs from many blockchain protocols [31, 40, 55] can be biased by a malicious adversary (though the blockchain protocols themselves are often secure). Protocols such as [24, 28, 46] have at least $O(\lambda n^3 \log n)$ total communication cost per beacon output, where λ is the security parameter and n is the number of nodes running the protocol.

A recent protocol Hydrand [61] reduces the communication cost to $O(\lambda n^2)$ but does not provide perfect unpredictability, even in the presence of a semi-honest adversary. Very recently, a concurrent and independent work Brandpiper [15] improves upon Hydrand to provide perfect unpredictibility and increased fault tolerance. As a trade-off, Brandpiper incurs higher worst-case communication and computation costs and makes the q−SDH assumption, which requires a trusted setup to generate the desired public parameters.

Regarding the setup assumption, many protocols [2, 15, 22, 44] assume an initial trusted setup, where a trusted party generates

Table 1: Comparison of existing randomness beacon protocol.

	Network model	Fault Tolerance	Liveness / Availability	Communication Complexity (total)	Unpredictability	Bias-resistance	Computation Complexity	Public Verification Complexity	Cryptographic Assumption	Setup Assumption
Aleph [37]	async.	1/3	✓	$O(\lambda n^2)$	✓	✓	$O(1)$	$O(1)$	Uniq. th-sig.	DKG
Cachin [55]	async.	1/3	✓	$O(\lambda n^2)$	✓	✓	$O(1)$	$O(1)$	Uniq. th-sig.	DKG
RandHerd [64]*	async.	1/3♣	✓	$O(\lambda c^2 \log n)$♣	✓	✓	$O(c^2 \log n)$	$O(1)$	PVSS+CoSi	DKG
Dfinity [44]	sync.	1/2	✓	$O(\lambda n^2)$	✓	✓	$O(1)$	$O(1)$	Uniq. th-sig.	DKG
Drand [2]	sync.	1/2	✓	$O(\lambda n^2)$	✓	✓	$O(1)$	$O(1)$	Uniq. th-sig.	DKG
HERB [28]♠	sync.	1/3	✓	$O(\lambda n^3)$	✓	✓	$O(n)$	$O(n)$	Partial HE	DKG
Algorand [40]	partial sync.	1/3♣	✓	$O(\lambda cn)$♣	$\Omega(t)$	✗	$O(c)$	$O(1)$	VRF	CRS
Proof-of-Work [55]	sync.	1/2	✓	$O(\lambda n)$	$\Omega(t)$	✗	very high	$O(1)$	Hash func.	CRS
Hydrand [61]	sync.	1/3	✓	$O(\lambda n^2)$	$t+1$	✓	$O(n)$	$O(n)$	PVSS	CRS
Ouroboros [46]	sync.	1/3	✓	$O(\lambda n^3)$	✓	✓	$O(n^3)$	$O(n^3)$	PVSS	CRS
Scrape [24]	sync.	1/2	✓	$O(\lambda n^4)$‡	✓	✓	$O(n^2)$	$O(n^2)$	PVSS+Broadcast	CRS
GRandPiper [15]	sync.	1/2	✓	$O(\lambda n^2)$	$t+1$	✓	$O(n^2)$	$O(n^2)$	VSS	q−SDH
BRandPiper [15]	sync.	1/2	✓	$O(\lambda n^3)$	✓	✓	$O(n^2)$	$O(n^2)$	VSS	q−SDH
SPURT	partial sync.	1/3	✓	$O(\lambda n^2 \log n + n^3)$	✓	✓	$O(n)$	$O(n)$	PVSS+Multisig.	CRS

* RandHerd uses RandHound as a one-time setup phase. RandHound is driven by a leader node and hence its liveness requires the leader to be honest. As presented, RandHerd is biasable but can use additional techniques to be unbiasable.

‡ Scrape assumes a broadcast channel and in the protocol every node uses the broadcast channel to share $O(n)$ groups elements. Thus even with the best known protocol for broadcast protocol, its total communication complexity would be $O(\lambda n^4)$.

♣ Algorand and Randherd use a randomly sampled subset of size c to run the protocol. This approach improves scalability and (slightly) reduces fault tolerance. It is thus not instructive to directly compare the efficiency of sampling-based and non-sampling-based protocols.

♠ HERB has three variant of protocol depending upon the underlying broadcast channel. We report their first variant since it uses standard metric for measuring broadcast channel communication cost.

trapdoors based on public parameters and shares them with the nodes. Security of such protocols relies crucially on the adversary's inability to access the trapdoor. Some protocols replace the trusted setup with a Distributed Key Generation (DKG) procedure [37, 64]. But it comes with a high initial setup cost as the best-known DKG protocols in synchronous and asynchronous networks has a communication cost of $O(n^3 \log n)$ and $O(n^4 \log n)$, respectively [37, 38, 64]. Another limitation of using DKG, as observed in [15], is the inability/inefficiency to replace nodes. Whenever a participating node is to be replaced, we would need to run the expensive DKG procedure again.

We summarize existing works in Table 1 and will provide more details about each protocol in §7.

Our result. In this paper, we design SPURT, an efficient distributed random beacon protocol that works in a partially synchronous network [34] and does not require any trusted or expensive setup phase. SPURT guarantees availability, unpredictability, unbiasibility, and public verifiability, against a malicious adversary that controls up to one-third of the nodes.

In a network of n nodes, for every beacon output, SPURT has a total amortized communication cost of $O(\lambda n^2)$ in the fault-free case ($O(1)$ malicious nodes) and $O(\lambda n^2 \log n + n^3)$ in the worst case, where λ is the security parameter. Note that for $\lambda = 256$ which is typical for most elliptic curves, $n^3 < \lambda n^2 \log n$ for $n < 2950$, which is reasonably large for most practical applications. SPURT's amortized computation cost per node per beacon output is only $O(n)$. Thus, we believe SPURT has good scalability and is suitable for applications with a large number of nodes deployed globally across the internet.

Design overview. In designing a beacon protocol with the above mentioned guarantees and properties, we encountered many challenges that resulted from various subtleties. To ensure unpredictability, each beacon output must require contribution from at least $t+1$ nodes. Furthermore, the contribution from honest nodes must remain hidden from the adversary before they become publicly available. In addition, the bias-resistant property requires that the adversary should not be able to force the protocol to abort after observing a beacon output.

Existing protocols that do not rely on trusted setup address these challenges using publicly verifiable secret sharing (PVSS) schemes. We will also start with this design paradigm. Briefly, the idea is that, for every beacon output, each node runs a concurrent instance of PVSS to share a randomly chosen secret with every other node. Once the sharing phase terminates for all nodes, the shares are reconstructed and aggregated to compute the beacon output.

The downside of naïvely using PVSS is that the PVSS schemes assume the existence of broadcast channels. In fact, this will be the major source of high communication complexity. A broadcast channel, when actually implemented using a distributed protocol, has a communication lower bound of $\Omega(n^2)$ [33].

This motivates us to revisit the use of broadcast channels in distributed randomness beacons. Previous works such as Scrape [24]

and Hydrand [61] explicitly mentioned that for a beacon protocol to be bias-resistant and available, the sub-protocol invoked for each beacon output must provide *guaranteed output delivery* [29]. The use of broadcast channels then becomes natural as it is standard and commonly assumed/used in the multi-party computation literature [42, 58] to achieve guaranteed output delivery. However, in this paper, we observe that guaranteed output delivery is not necessary, and as a result, we can use an state machine replication protocol (SMR) (cf. §2.5) instead of Byzantine broadcast [48],which enables us to extend our results to a partially synchronous network [34].

The second key technique to reduce communication complexity is to use accumulators such as Merkle trees and additively homomorphic commitment and encryption schemes in PVSS to compress the amount of data sent using the SMR protocol.

Thirdly, we observe that to decide on any beacon output, the SMR requires acknowledgments from a quorum of $2t + 1$ nodes. This implies acknowledgments from at least $t + 1$ honest nodes. This property, along with the properties of threshold secret sharing ensures bias-resistance and achieves optimal fault tolerance. Lastly, we use multi-signature to enforce unpredictability of each beacon output.

Evaluation. We implement SPURT in Golang atop the open-source Quorum [6], which implements the Istanbul BFT [54] state machine replication protocol and is a fork of open source go-ethereum [4] implementation. We then evaluate our prototype for a network of up to 128 nodes running in geographically distributed AWS EC2 instances. We evaluate the throughput of SPURT, measured as the number of beacons generated per minute, the network bandwidth usage and computation time per node per beacon output. Our evaluation illustrates that for a network of size up to 64, SPURT can generate at least one beacon output every second, and has a bandwidth cost (amount of sent plus received data) of about 43 Kilobytes per node per beacon output. For a larger network with 128 nodes, SPURT can generate one beacon output every 3 seconds and the bandwidth cost is 85 Kilobytes per node per beacon output.

Summary. In summary, we make the following contributions:

- We design SPURT, a partially synchronous distributed random beacon protocol with a total amortized communication cost of $O(\lambda n^2 \log n + n^3)$ ($O(\lambda n^2)$ in the fault-free case) per beacon output and does not require any trusted or expensive setup.
- We formally prove that SPURT is available and each beacon output is unpredictable, bias-resistant and publicly-verifiable, against a malicious adversary controlling up to one-third of the nodes in a partially synchronous network.
- We implement a prototype of SPURT and evaluate it in a network of up to 128 nodes. Our evaluation illustrate that SPURT is concretely efficient and can generate output every few seconds.

Paper organization. The rest of the paper is organized as follows. In §2, we discuss relevant background and introduce notations. We then describe the system model and provide an overview of SPURT in §3 followed by the detailed description of the protocol in §4. We analyze the security and performance guarantees of SPURT in §5. We present the details of our prototype implementation and evaluation results in §6. We describe related work in detail in §7 and conclude with a discussion in §8.

2 PRELIMINARIES

Let λ be the security parameter. Let $\mathbb{G}$ be a cyclic abelian group of prime order q and $\mathbb{Z}_q$ the group of integer modulo q. We say a probabilistic event happens *with high probability* or *w.h.p* if that event happens with probability $1-\epsilon(z)$ for some negligible function ϵ in z. We denote an element x sampled uniformly from a finite set $\mathcal{M}$ by $x \leftarrow \mathcal{M}$. We denote vectors using $\mathbf{x}$ and inner product between two vectors $\mathbf{x}, \mathbf{y}$ by $\langle \mathbf{x}, \mathbf{y} \rangle$. For any integer n, we will use $[n]$ to denote the set $\{1, 2, \ldots, n\}$.

We next define the desired properties of a distributed random beacon protocol and then briefly discuss the tools we use in SPURT.

2.1 Randomness Beacon

The two most crucial property for a randomness beacon are *unpredictability* and *bias-resistance*. Unpredictability ensures that a cryptographically bounded malicious adversary controlling up to a threshold fraction of nodes should not be able to predict or compute any function on any future beacon outputs with non-negligible advantage. The bias-resistance property of the beacon protocol requires that every output is chosen uniformly randomly from the intended distribution and independently of other outputs. This too should hold even in the presence of an adversary controlling a threshold fraction of nodes in the system.

In addition to unpredictability and bias-resistance, any beacon protocol should also guarantee *availability*, i.e., an adversary controlling a threshold fraction of nodes should not prevent the protocol from producing new beacon outputs. The final desirable property is *public-verifiability*, which states that each beacon output is efficiently verifiable even by users that do not directly participate in the beacon generation protocol.

2.2 Zero knowledge Proof of Equality of Discrete Logarithm

SPURT has a step that requires nodes to produce zero-knowledge proofs about equality of discrete logarithms for a tuple of publicly known values. In particular, given a group $\mathbb{G}$ of prime order q, two uniformly random generators $g, h \leftarrow \mathbb{G}$ and a tuple (g, x, h, y), a prover $\mathcal{P}$ wants to prove to a probabilistic polynomial time (PPT) verifier $\mathcal{V}$, in zero-knowledge, the knowledge of a witness α such that $x = g^\alpha$ and $y = h^\alpha$. Throughout this paper, we will use the Chaum-Pedersen sigma protocol [27], which assumes the hardness of the Decisional Diffie-Hellman (DDH) problem, and can be made non-interactive using the Fiat-Shamir heuristic [35].

Decisional Diffie–Hellman assumption. Given a group $\mathbb{G}$ with generator $g \in \mathbb{G}$ and uniformly random samples $a, b, c \leftarrow \mathbb{Z}_q$, the Decisional Diffie–Hellman (DDH) hardness assumes that the following two distributions D_0, D_1 are computationally indistinguishable: $D_0 = (g, g^a, g^b, g^{ab})$ and $D_1 = (g, g^a, g^b, g^c)$.

Protocol for equality of discrete logarithm. For any given tuple (g, x, h, y), the Chaum-Pedersen protocol proceeds as follows.

(1) $\mathcal{P}$ samples a random element $\beta \leftarrow \mathbb{Z}_q$ and sends (a_1, a_2) to $\mathcal{V}$ where $a_1 = g^\beta$ and $a_2 = h^\beta$.

(2) $\mathcal{V}$ sends a challenge $e \leftarrow \mathbb{Z}_q$.
(3) $\mathcal{P}$ sends a response $z = \beta - \alpha e$ to $\mathcal{V}$.
(4) $\mathcal{V}$ checks whether $a_1 = g^z x^e$ and $a_2 = h^z y^e$ and accepts if and only if both the equality holds.

As mentioned, this protocol can be made non-interactive in the Random Oracle model using the Fiat-Shamir heuristic [35, 56]. This protocol guarantees completeness, knowledge soundness, and zero-knowledge. The knowledge soundness implies that if $\mathcal{P}$ convinces the $\mathcal{V}$ with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from the prover with non-negligible probability.

Throughout this paper, we will use the non-interactive variant of the protocol described above and denote it using $\mathsf{dleq}(\cdot)$. In particular, for any given tuple (g, x, h, y) where $x = g^s$ and $y = h^s$, $\pi \leftarrow \mathsf{dleq.Prove}(s, g, x, h, y)$ generates the proof π. Given the proof π and (g, x, h, y), $\mathsf{dleq.Verify}(\pi, g, x, h, y)$ verifies the proof.

2.3 Threshold Secret Sharing

A $(n, t+1)$ threshold secret sharing scheme allows a secret $s \in \mathbb{Z}_q$ to be shared among n nodes such that any $t+1$ of them can come together to reconstruct the original secret, but any subset of t shares cannot be used to reconstruct the original secret [16, 63]. We use the common Shamir secret sharing [63] scheme , where the secret is embedded in a random degree t polynomial in the field $\mathbb{Z}_q$ for some prime q. Specifically, to share a secret $s \in \mathbb{Z}_q$, a polynomial $p(\cdot)$ of degree t is chosen such that $s = p(0)$. The remaining coefficients of $p(\cdot)$, $a_1, a_2, \cdots, a_t$ are chosen uniformly randomly from $\mathbb{Z}_q$. The resulting polynomial $p(x)$ is defined as:

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_t x^t$$

Each node is then given a single evaluation of $p(\cdot)$. In particular, the i^{th} node is given $p(i)$ i.e., the polynomial evaluated at i. Observe that given $t+1$ points on the polynomial $p(\cdot)$, one can efficiently reconstruct the polynomial using Lagrange Interpolation. Also note that when s is uniformly random in $\mathbb{Z}_q$, s is information theoretically hidden from an adversary that knows any subset of t or less evaluation points on the polynomial other than $p(0)$ [63].

2.4 Publicly Verifiable Secret Sharing

SPURT crucially relies on publicly verifiable secret sharing (PVSS). In particular, we use the PVSS scheme from Scrape [24], which is an improvement over the Schoenmakers scheme [62]. The scheme allows a node (dealer) to share a secret $s \in \mathbb{Z}_q$ among n nodes, such that any subset of at least $t+1$ nodes can reconstruct h^s. Here, h is one of the independent generators of $\mathbb{G}$. The reconstruction threshold $t+1$ is chosen in a way such that a colluding adversary can not recover h^s without contribution of at least one honest node. A key property of a PVSS scheme is that, not only the recipients but any third party (with access to recipients' public keys) can verify, even before the reconstruction phase begins, that the dealer has generated the shares correctly without having plaintext access to the shares. This property is crucial to SPURT .

The PVSS scheme of Scrape [24] is non-interactive in the random oracle model and has three procedures: PVSS.Share, PVSS.Verify, and PVSS.Recon. A node (dealer) with public-private key pair pk, sk, uses PVSS.Share to share a secret s, other nodes or external users use PVSS.Verify to validate the shares, and PVSS.Recon is used to recover h^s. We describe them in detail in Figure 1.

Let s be the secret a node (the dealer) with public-private key pair (sk, pk) wants to share with set of nodes with public keys $\{pk_j\}_j$ for $j = 1, 2, \ldots, n$.

$\mathsf{PVSS.Share}(s, g, h, sk, \{pk\}_{j,j=1,2,\ldots,n}) \rightarrow (\mathbf{v}, \mathbf{c}, \boldsymbol{\pi})$:

(1) Sample uniform random $a_k \in \mathbb{Z}$ for $k = 1, 2, \ldots, t-1$ and let
$$p(x) = s + a_1 x + \ldots + a_t x^t;$$
(2) Compute $s_j \leftarrow p(j)$; $v_j \leftarrow g^{s_j}$; and $c_j \leftarrow pk_j^{s_j}$, $\forall j \in [n]$.
(3) Compute $\pi_j \leftarrow \mathsf{dleq.Prove}(s, g, v_j, pk_j, c_j)$
(4) Output $\mathbf{v} = \{v_1, v_2, \ldots, v_n\}$; $\mathbf{c} = \{c_1, c_2, \ldots, c_n\}$, and $\boldsymbol{\pi} = \{\pi_1, \pi_2, \ldots, \pi_n\}$.

$\mathsf{PVSS.Verify}(g, h, \mathbf{v}, \mathbf{c}, \boldsymbol{\pi}, \{pk\}_{j,j=1,2,\ldots,n}) \rightarrow 0/1$:

(1) Sample a random code word $\mathbf{y}^\perp \in C^\perp$ and check whether
$$\prod_{k=1}^{n} v_k^{x_k^\perp} = 1_{\mathbb{G}} \tag{1}$$
where $1_{\mathbb{G}}$ is the identity element of $\mathbb{G}$.
(2) Check whether $\mathsf{dleq.Verify}(g, v_j, pk_j, c_j, \pi_j) = 1$ for all j.
(3) Output 1 if both checks pass, otherwise output 0.

Let T be the set of valid tuples of the form $(\tilde{s}_j, \tilde{\pi}_k)$ where $\tilde{\pi}_k = \mathsf{dleq.Prove}(g, pk_k, \tilde{s}_k, c_k))$ where $|T| = t+1$, then

$\mathsf{PVSS.Recon}(\{\tilde{s}_k\}_{k \in T}) \rightarrow h^s$:

(1) Output
$$\prod_{k \in T} (\tilde{s}_k)^{\mu_k} = \prod_{k \in T} h^{\mu_k \cdot p(k)} = h^{p(0)} \tag{2}$$
where $\mu_k = \prod_{j \neq k} \frac{j}{j-k}$ for $k \in T$ are Lagrange coefficients.

Figure 1: Scrape's PVSS scheme.

The verification procedure of Scrape's PVSS uses properties of error correcting code, specifically the Reed Solomon code [59]. They use the observation by McEliece and Sarwate [51] that sharing of a secret using a degree t polynomial among n nodes is equivalent to encoding the message $(x, a_1, a_2, \cdots, a_t)$ using a $[n, t+1, n-t]$ Reed Solomon code [59].

Let C be a $[n, k, d]$ linear error correcting code over $\mathbb{Z}_q$ of length n and minimum distance d. Also, let $C^\perp$ be the dual code of C i.e., $C^\perp$ consists vectors $\mathbf{y}^\perp \in \mathbb{Z}_q^n$ such that for all $\mathbf{x} \in C$, $\langle \mathbf{x}, \mathbf{y}^\perp \rangle = 0$. Here, $\langle \cdot, \cdot \rangle$ is the inner product operation. Scrape's PVSS.Verify uses the following basic fact (Lemma 2.1) of linear error correcting code. We refer readers to [24, Lemma 1] for its proof, and §A for a brief description of the Reed Solomon and its dual code.

Lemma 2.1. *If $\mathbf{x} \in \mathbb{Z}_q^n \setminus C$, and $\mathbf{y}^\perp$ is chosen uniformly at random from $C^\perp$, then the probability that $\langle \mathbf{x}, \mathbf{y}^\perp \rangle = 1$ is exactly $1/q$.*

The PVSS scheme of Scrape provides the secrecy property stated in Theorem 2.2. We refer reader to Appendix B for the definition of IND1-Secrecy and to [24] for the proof of the theorem.

Theorem 2.2. (IND1-Secrecy [24, Theorem 1]) *Under the decisional Diffie-Hellman assumption, the PVSS protocol in [24] is IND1-secret against a static probabilistic polynomial time adversary that can collude with up to t nodes.*

Remark. SPURT does not use the Scrape's PVSS scheme in a black-box manner because their PVSS scheme requires a broadcast channel over which the dealer sends a large amount of data. Instead, we customize the use of PVSS procedures in SPURT to minimize the amount of data sent over the broadcast channel. Then, in the actual implementation, we use a state machine replication protocol to serve as the broadcast channel.

2.5 State Machine Replication

A State Machine Replication is a distributed protocol run by a network of n nodes to decide on a sequence of values, one for each height. It provides the following properties.

- **Agreement/Safety.** If an honest node decides some value v in height r, then for height r, no honest node decides on a value v' such that $v' \neq v$ for height r.
- **Validity/Liveness.** If an honest node broadcasts a value v, every honest node eventually decides v in some height.
- **Verifiability.** Whenever a node decides on a value, it can prove to other nodes and external parties the correctness of the decided value.

Note that unlike regular SMR [26, 67], in our case only participating nodes propose values and all decided values must meet a certain external valid predicate M.

We will use Istanbul BFT (IBFT) [54]. It is a variant of the popular PBFT [26] protocol and tolerates up to one third malicious nodes in a partially synchronous network (which is optimal). IBFT is an *epoch* based protocol, where each epoch has a leader. In every epoch, the IBFT protocol finalizes a value in three steps: *Propose, Prepare*, and *Commit*. We present a simplified description of the IBFT protocol in Figure 2, and refer the reader to [54] for more details.

Let r be the current epoch and L be its leader. Also, let $\mathsf{ht}-1$ be the latest finalized height.

Propose. L proposes a value z to be finalized at height ht by sending $\langle propose, z, r, \mathsf{ht}, X\rangle$ message to all the nodes. Here X is the view change certificate (if any) that validates that the proposal is safe.

Prepare. Each node P_j, upon receiving the proposal checks whether the proposal is consistent with IBFT specifications using X, and $M(z)$ is true for an external predicate $M(\cdot)$. If both checks pass, P_j multi-casts $\langle prepare, z, r, \mathsf{ht}\rangle$ to all nodes.

Commit. Upon receiving $2f + 1$ *prepare* messages for the proposal z at height ht and epoch r, P_j multi-casts $\langle commit, z, r, \mathsf{ht}\rangle$ message to every node.

Upon receiving $2f+1$ *commit* messages for (z, r, ht), **decide** z for height ht.

Figure 2: Steady state of Istanbul BFT [54] SMR protocol.

3 SYSTEM MODEL AND OVERVIEW

3.1 System Model

We consider a network of n nodes connected via pair-wise authenticated channels. We assume a standard public-key infrastructure, i.e., every node in the system is aware of every other node's public key in the system. Let pk_i be the public key of node i. We assume that at most $t < n/3$ nodes can be malicious. A single adversary, $\mathcal{A}$ controls all malicious nodes. The remaining nodes are honest and they strictly follow the specified protocol. We also assume that at the start of the protocol, all honest nodes agreed on a pair (g, h) of randomly and independently chosen generators of $\mathbb{G}$. This is a common reference string (CRS) setup. We assume that $\mathcal{A}$ cannot break standard cryptographic constructions such as hash functions, signatures schemes and the ones specified in §2.

We adopt the standard partially synchronous network model [34], i.e., a network that oscillates between periods of synchrony and periods of asynchrony. During periods of synchrony all messages sent by honest replicas adhere to a known delay bound Δ. During periods of asynchrony messages, messages can be delayed arbitrarily. (In theoretical works, the partial synchrony model [34] is often stated differently (e.g., using an unknown Global Standardization Time, GST) for rigor or convenience, but the essence is to capture the practical oscillating timing model mentioned above.) A beacon protocol in the partially synchronous model is secure if it ensures that every beacon output is unpredictable, bias-resistant, and publicly verifiable even during periods of asynchrony, and guarantees availability during periods of synchrony.

3.2 Overview

We now give an overview of SPURT to describe our core ideas. For ease of exposition, we will first explain how SPURT generates a single beacon output assuming that nodes have access to a broadcast channel. Strictly speaking, a single-value broadcast channel [48] is impossible in a partially-synchronous network; we merely assume its existence to simplify this overview and aid intuitive understanding. Later in §4, we will replace it with an BFT SMR protocol while preserving the overall efficiency of the protocol. Throughout this paper, we will assume that all messages exchanged between honest nodes are digitally signed by the sender, and recipients validate them before processing them further.

SPURT proceeds in epochs where each epoch has a designated leader. The leader of a given epoch is chosen using any deterministic algorithm. For concreteness, we assume leaders are chosen in a round-robin order, i.e., the leader of epoch r is node $i = r \bmod n$ with public key pk_i. We will use L_r to denote the leader of epoch r. The beacon generation process in every epoch has four phases: *Commitment, Aggregation, Agreement* and *Reconstruction* phase. We illustrate the communication pattern of in all four in Figure 3 and describe the message contents (symbols over the arrows) in §4.

Commitment phase. During the commitment phase, each node chooses a uniformly random secret and computes shares for the chosen secret using the PVSS.Share primitive described in §2.4. Each node then sends all these shares to L_r. Here on, we refer to the messages sent by nodes to L_r as the PVSS tuples of epoch r. We remark that, despite having access to PVSS messages from

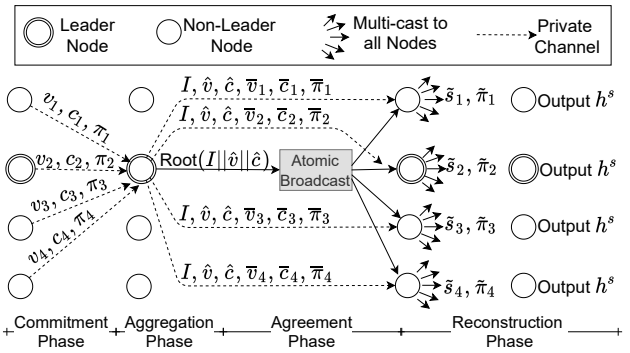

Figure 3: Messages sent during each phase of the SPURT. We describe contents of the messages i.e., the notations over the arrows in §4. We use Root(·) as the shorthand for Merkle root.

all nodes, L_r can not 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. We will give more details on this in §5.

Aggregation phase. In the aggregation phase, L_r upon receiving PVSS tuple from a node, validates them using PVSS.Verify from §2.4. 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 $p_1, p_2, \ldots, p_{t+1}$ are the underlying polynomials from the $t+1$ valid polynomial commitments, L_r aggregates them to obtain the commitment to the *aggregated polynomial* $\hat{p}(x) = \sum_{j=1}^{t+1} p_j(x)$. Moreover, L_r aggregates the $t+1$ encrypted shares to obtain encrypted shares corresponding to the aggregated polynomial $\hat{p}(\cdot)$.

Agreement phase. After aggregation, L_r computes a cryptographic digest of the commitment of the aggregated polynomial $\hat{p}(\cdot)$, the identities (or indices) of nodes whose polynomial are aggregated into $\hat{p}(\cdot)$, and the encrypted shares of the secret embedded in $\hat{p}(\cdot)$. L_r then sends this cryptographic digest to all of the nodes via a broadcast channel. Note again that our actual protocol will use an SMR protocol instead.

Additionally, to each node i, L_r sends node i the entire commitment to the aggregated polynomial $\hat{p}(\cdot)$, and the encrypted shares corresponding to $\hat{p}(\cdot)$ 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 $\hat{p}(\cdot)$, 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 $O(n\lambda)$ 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 epoch with the next leader and the cycle continues.

Reconstruction phase. When the agreement phase terminates, i.e., all honest nodes agree on the cryptographic digest broadcast by the L_r, every honest node who received valid shares from L_r

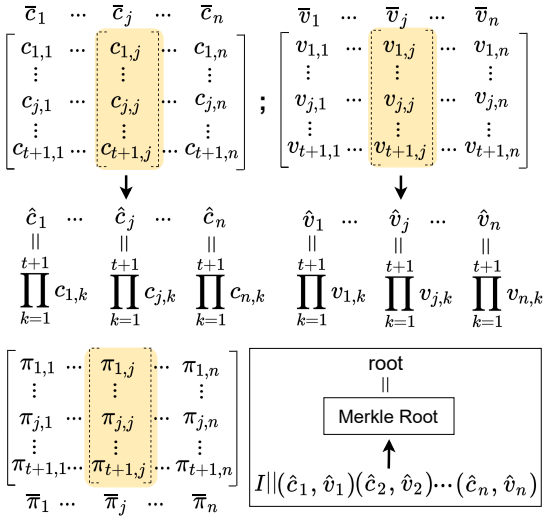

Figure 4: Aggregation phase at the leader.

multicasts its aggregated share along with the NIZK proof of its correctness. As we show in §5, if the agreement phase terminates successfully, then at least $t+1$ honest nodes hold valid shares of the aggregated polynomial $\hat{p}(\cdot)$. Also, all nodes will be able to prove the correctness of their aggregated shares. Moreover, all these nodes starts the reconstruction process within three message transmission delays. Hence, during the reconstruction phase, every honest node will receive at least $t+1$ valid shares of $\hat{p}(\cdot)$, along with their correctness proofs. As a result, every honest node will be able to successfully reconstruct the polynomial $h^{\hat{p}(\cdot)}$, and hence the output of the beacon for this epoch.

4 DESIGN

In this section, we present the detailed design of SPURT. As discussed in previous sections, SPURT proceeds in epochs and each epoch has four phases. We next describe each phase in detail.

Let g and h be two independently chosen generators of a group $\mathbb{G}$ of order q (this is the common reference string). We will model hash(·) as a random oracle. Recall from §A, C is the linear error correcting code corresponding to $(t, n+1)$ Shamir secret sharing and $C^{\perp}$ is the dual code of C. Let $sk_i \leftarrow \mathbb{Z}_q$ and $pk_i = h^{sk_i}$ be the secret and public keys of node i.

4.1 Commitment Phase

For any given epoch r, let L_r be its leader. Each node i samples a uniformly random secret $s_i \leftarrow \mathbb{Z}_q$ and computes the PVSS tuples using the PVSS.Share primitive described in §2.4:

$$\mathbf{v}_i, \mathbf{c}_i, \boldsymbol{\pi}_i \leftarrow \text{PVSS.Share}(s_i, g, h, sk_i, \{pk\}_{j,j=1,2,\ldots,n}) \tag{3}$$

Recall from Figure 1, $\mathbf{v}_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,n}\}$, $\mathbf{c}_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,n}\}$, and $\boldsymbol{\pi}_i = \{\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,n}\}$. Node i then sends the tuple $(\mathbf{v}_i, \mathbf{c}_i, \boldsymbol{\pi}_i)$ to L_r.

4.2 Aggregation Phase

The leader L_r, on receiving each PVSS tuple $(\mathbf{v}_i, \mathbf{c}_i, \boldsymbol{\pi}_i)$, validates them using PVSS.Verify$(\mathbf{v}_i, \mathbf{c}_i, \boldsymbol{\pi}_i)$. Upon receiving $t+1$ valid PVSS

tuples, L_r aggregates them as follows. Let $I \subseteq [n]$ be the set of nodes that send valid PVSS tuples. L_r computes the commitment $\hat{\mathbf{v}} = (\hat{v}_1, \hat{v}_2, ..., \hat{v}_n)$ of the aggregated polynomial $\hat{p}(\cdot) = \sum_{i \in I} p_i(\cdot)$, where

$$\hat{v}_\ell = \prod_{i \in I} v_{i,\ell} \tag{4}$$

Similarly, L_r aggregates the encryptions using their additive homomorphism property, i.e., L_r computes the vector $\hat{\mathbf{c}} = (\hat{c}_1, \hat{c}_2, ..., \hat{c}_n)$, where

$$\hat{c}_\ell = \prod_{i \in I} c_{i,\ell} \tag{5}$$

Figure 4 illustrates this step using $I = \{1, 2, \ldots, t+1\}$ as an example. Observe that the $t+1$ PVSS tuples received and validated by L_r can be represented as three matrices shown in Figure 4. Here on, we refer to these matrices as the commitment matrix $\{v_{i,j}\}$, the ciphertext matrix $\{c_{i,j}\}$, and the proof matrix $\{\pi_{i,j}\}$. Let $\bar{\mathbf{c}}_j, \bar{\mathbf{v}}_j$ and $\bar{\boldsymbol{\pi}}_j$ be the j^{th} column of the ciphertext, commitment, and proof matrix respectively. Stated differently, $\bar{\mathbf{c}}_j$ is the set of encryptions sent by nodes in I that are encrypted under the public key of node j. Similarly, $\bar{\mathbf{v}}_j$ and $\bar{\boldsymbol{\pi}}_j$ are j^{th} coordinate of commitments and dleq proofs sent by nodes in I, respectively. Without loss of generality, let $I = \{1, 2, \ldots, t+1\}$, then $\bar{\mathbf{c}}_j = \{c_{1,j}, c_{2,j}, \ldots, c_{t+1,j}\}$, $\bar{\mathbf{v}}_j = \{v_{1,j}, v_{2,j}, \ldots, v_{t+1,j}\}$, and $\bar{\boldsymbol{\pi}}_j = \{\pi_{1,j}, \pi_{2,j}, \ldots, \pi_{t+1,j}\}$.

Then, $\hat{c}_j$ is the product of all elements in $\bar{\mathbf{c}}_j$ and $\hat{v}_j$ is the product of all elements in $\bar{\mathbf{v}}_j$.

Next, L_r computes `root`, the Merkle root [52] that commits $I, \hat{\mathbf{v}}$, and $\hat{\mathbf{c}}$, also shown in Figure 4. In the agreement phase, `root` will be the only value that is sent using the SMR protocol. $I, \hat{\mathbf{v}}, \hat{\mathbf{c}}$ themselves and the original PVSS tuples will be sent privately to corresponding nodes.

4.3 Agreement Phase

Let ht be the height chosen by L_r according to SMR. Then, to each node j, L_r sends $(\texttt{root}, \hat{\mathbf{v}}, \hat{\mathbf{c}}, I, \bar{\mathbf{v}}_j, \bar{\mathbf{c}}_j, \bar{\boldsymbol{\pi}}_j, \text{ht})$ and proposes `root` using the SMR protocol for height ht.

Observe that in the above message, only $\bar{\mathbf{v}}_j, \bar{\mathbf{c}}_j$, and $\bar{\boldsymbol{\pi}}_j$ are recipient specific and everything else is common to all nodes. Essentially, the tuple L_r sends to each node corresponds to the BFT SMR proposal on `root` for epoch r and height ht.

Upon receiving $(\texttt{root}, \hat{\mathbf{v}}, \hat{\mathbf{c}}, I, \bar{\mathbf{v}}_j, \bar{\mathbf{c}}_j, \bar{\boldsymbol{\pi}}_j, \text{ht})$ from L_r, node j validates them by checking:

(1) The proposal is safe according to SMR,
(2) `root` is a valid Merkle root of $I, \hat{\mathbf{v}}$, and $\hat{\mathbf{c}}$; and
(3) For a randomly chosen code word $\mathbf{y}^\perp = \{y_1^\perp, y_2^\perp, ..., y_n^\perp\}$ from the dual code $C^\perp$

$$\prod_{k=1}^{n} \hat{v}_k^{y_k^\perp} = 1_{\mathbb{G}}; \text{ and} \tag{6}$$

(4) Every tuple $(v_{i,j}, c_{i,j}, \pi_{i,j}) \in (\bar{\mathbf{v}}_j, \bar{\mathbf{c}}_j, \bar{\boldsymbol{\pi}}_j)$ is valid dleq proof according to §2.2; and
(5) $\hat{c}_j = \prod_{i \in I} \bar{c}_{i,j}$ and $\hat{v}_j = \prod_{i \in I} \bar{v}_{i,j}$.

If all of the above mentioned checks are satisfied, node j multicasts the $\langle \textit{prepare}, \texttt{root}, r, \text{ht} \rangle$ to all other nodes. Alternatively, if any of the above checks fails or if j does not receive the required private information from L_r, j does not send the *prepare* message in the prepare step of the SMR protocol (cf. §2.5).

As discussed in Figure 2, every honest node upon receiving $2t+1$ $\langle \textit{prepare}, \texttt{root}, r, \text{ht} \rangle$ messages multi-casts $\langle \textit{commit}, \texttt{root}, r, \text{ht} \rangle$ to all other nodes. Next, each honest node upon receiving $2t+1$ $\langle \textit{commit}, \texttt{root}, r, \text{ht} \rangle$ messages decide on `root` at height ht.

4.4 Reconstruction Phase

Every honest node that *decides* `root` (cf. §2.5) and receives valid messages from the leader during the agreement phase starts the reconstruction phase for the beacon at height ht. In particular, these nodes compute the reconstruction share $\tilde{s}_j$ and its correctness proof $\tilde{\pi}_j$ as in equation (7) and multicasts $(\tilde{s}_j, \tilde{\pi}_j)$ them to all other nodes.

$$\tilde{s}_j = \hat{c}_j^{\frac{1}{sk_j}} = h^{\sum_{i \in I} s_{i,j}}; \text{and } \tilde{\pi}_j = \text{dleq.Prove}(sk_j, h, pk_j, \tilde{s}_j, \hat{c}_j) \tag{7}$$

Let H be the set of honest nodes and let $V \subseteq H$ be the set of honest nodes that received valid messages from L_r during the proposal step of the SMR, but are yet to decide on `root`. Upon hearing a reconstruction message $(\tilde{s}_j, \tilde{\pi}_j)$ from a node j, nodes in V requests node j for the proof of the decision on `root`, to which node j responds by sending the multi-signature of the proof of decision. Upon receiving the multi-signature, nodes in V validate it for correctness, and on successful validation multi-casts their reconstruction shares all other nodes. This implies all honest nodes who received valid shares from L_r start the reconstruction within three message delays from the instant an honest node decides on `root`.

Every node i, upon receiving a tuple $(\tilde{s}_j, \tilde{\pi}_j)$, validates $\tilde{\pi}_j$ using dleq.Verify. Note that some honest nodes who did not receive valid messages from the leader, may not have $\hat{\mathbf{v}}$ and/or $\hat{\mathbf{c}}$. In such a situation, these nodes query the sender of $(\tilde{s}_j, \tilde{\pi}_j)$ for $(\hat{v}_j, \hat{c}_j)$ and the Merkle path from $(\hat{v}_j, \hat{c}_j)$ to `root`, and validate them using dleq.Verify

Let T be the set of nodes from which node i receives valid $(\tilde{s}_j, \tilde{\pi}_j)$ tuples. Upon receiving $t+1$ such valid tuples, i.e., when $|T| \geq t+1$, i outputs the beacon output for height ht as h^s. Recall from §4.1, $s = \hat{p}(0) = \sum_{i \in I} s_i$. The honest nodes constructs h^s using the Lagrange interpolation:

$$\prod_{k \in T} (\tilde{s}_k)^{\mu_k} = \prod_{k \in T} h^{\mu_k \cdot \hat{p}(k)} = h^{\hat{p}(0)} \tag{8}$$

where $\mu_k = \prod_{j \neq k} \frac{j}{j-k}$ are the Lagrange coefficients.

4.5 Optimizations

In this section, we will describe few optimizations we employ to improve the computation efficiency of nodes and amortize the computation and bandwidth usage of the leader.

Pre-aggregating data. Recall from §4.2, during the aggregation phase, the leader validates a total of $O(n^2)$ NIZK proofs. Moreover, the leader aggregates polynomial commitments from $t+1$ nodes. As a result, the leader performs $O(n^2)$ group exponentiations. For a large n, this may introduce delay in the agreement phase, and hence delay the beacon generation process.

SPURT addresses this by enabling leaders to pre-compute the messages of aggregation phase. In particular, at any epoch r, every node sends their PVSS shares for epoch $r+\tau$ to $L_{r+\tau}$. Here, τ is a system parameter. Since the leader selection rule in SPURT is deterministic, $L_{r+\tau}$ is fixed, and known to all nodes in advance. $L_{r+\tau}$,

upon receiving the shares for epoch $r + \tau$, immediately starts aggregating them as in §4.2, and sends the aggregated messages as well as the private messages to each node. By doing so, SPURT amortizes the leader's higher usage of computation and communication across τ epochs. As a result, during epoch $r + \tau$, $L_{r+\tau}$ only sends λ bits of information to every node, incurring a total bandwidth usage of $O(n\lambda)$ bits (instead of $O(n^2\lambda)$ bits), which is comparable to non-leader nodes.

Multi-exponentiation. We further reduce the computation cost using the multi-exponentiation technique [53]. For any given group $\mathbb{G}$, let $\mathbf{g} = [g_1, g_2, \cdots, g_m]$ be a vector of m elements in $\mathbb{G}$, and let $\mathbf{a} = [a_1, a_2, \cdots, a_m]$ be a vector of m scalars in $\mathbb{Z}_q$. Given $\mathbf{a}$ and $\mathbf{g}$, the multi-exponentiation technique computes more efficiently:

$$g' = \prod_{k=1}^{m} g_k^{a_k} \tag{9}$$

In SPURT, nodes need to compute an expression of this form to: (*i*) validate the polynomial commitments sent during commitment phase; (*ii*) validate the aggregated polynomial sent by the leader; and (*iii*) compute the beacon output from reconstruction shares. In our implementation, we use the Interleaved window method of computing multi-exponentiation from Strauss [13].

5 ANALYSIS

In this section, we will first argue that SPURT is available in a partially synchronous network. Next, we will prove that every output of SPURT is unpredictable, bias-resistant, and publicly-verifiable. We then analyze the computation and communication complexity of each epoch. Lastly, we will discuss the latency involved in generating a beacon output. Throughout our analysis, we will assume that the dleq protocol is sound and complete.

5.1 Reconstructability and Availability

We will first argue that at any given epoch r and height ht, if an honest node decides on a digest `root`, then at least $t+1$ honest nodes must have possessed and validated the private messages associated with `root`. We then use this property, the safety property of the BFT SMR, and the fact that the beacon output for every decided value is reconstructible by all honest nodes to prove the bias-resistance property of SPURT.

Lemma 5.1. *At any given height* ht *and epoch r, if an honest node decides on* `root`*, then at least $t+1$ honest nodes received valid messages from L_r at epoch r.*

Proof. An honest node decides `root`, only if it receives a quorum Q_2 of $2t + 1$ *commit* messages. Since, there are at most t malicious nodes in the system, at least $t + 1$ of the nodes in Q_2 are honest. Also, since an honest node sends a *commit* message only upon receiving $2t + 1$ *prepare* messages, by a similar argument this implies that at least $t + 1$ honest nodes sent *prepare* message.

Recall from §4.3, an honest node sends a *prepare* message only if it received a valid message from L_r, this implies that at least $t + 1$ honest nodes received valid messages for epoch r. □

Next, we will argue that whenever honest nodes output any value due to the SMR then *w.h.p* the degree of the underlying polynomial is at most t.

Lemma 5.2. *For any given epoch r and height* ht*, if an honest node outputs* `root` *and $\hat{p}(\cdot)$ is the polynomial whose commitment is embedded in the leaves of* `root`*, i.e.,*

$$\hat{\mathbf{v}} = (g^{\hat{p}(1)}, g^{\hat{p}(2)}, ..., g^{\hat{p}(n)}) \tag{10}$$

then with probability at least $1 - 1/q$, $\deg(\hat{p}(\cdot)) \leq t$, where $\deg(\cdot)$ denote the degree of the polynomial.

Proof. When an honest node outputs `root` in any epoch r, from Lemma 5.1 we know that at least $t + 1$ nodes have validated the messages sent by L_r and found them to be valid. This implies that the check (6) was successful for at least $t + 1$ nodes.

Recall from Lemma (2.1), for any polynomial of degree greater than t, at any honest node, the check in equation (6) passes with probability exactly $1/q$. Let A be the event that $\deg(\hat{p}(\cdot)) > t$ and the check (6) was successful for at least $t + 1$ honest nodes. Since every honest node checks this condition independently,

$$\Pr[A] \leq \binom{2t+1}{t+1} \frac{1}{q^{t+1}} \leq \frac{1}{q}; \quad (\text{since } t \ll q) \qquad \square$$

An immediate consequence of Lemma 5.1 and 5.2 is that for every `root` output by the SMR the corresponding beacon output $h^{\hat{p}(0)}$ is reconstructible by the honest parties.

Lemma 5.3. *For any given epoch r, if an honest node decides* `root`*, then $h^{\hat{p}(0)}$ is reconstructible by the honest nodes.*

Proof. From Lemma 5.2 and 5.1 *w.h.p* the degree of $\hat{p}(\cdot)$ is at most t and at least $t + 1$ honest nodes received valid messages from L_r. Let T be the set of indices of such honest nodes, then by the security guarantees of dleq, each node P_j for $j \in T$ holds $h^{sk_j \cdot \hat{p}(j)}$. Also, these nodes can compute the decrypted share $h^{\hat{p}(j)}$ and construct a NIZK proof of its correctness using their secret key sk_j. Honest nodes can use these shares to reconstruct $h^{\hat{p}(0)}$ using Lagrange interpolation. □

Next, we will argue that during periods of synchrony, when an honest node becomes leader SPURT is available.

Theorem 5.4. (Availability) *During periods of synchrony, if the leader L_r of an epoch r is honest, w.h.p* SPURT *will produce an output and that output will be available at every honest node.*

Proof. When L_r is honest, all the checks described in §4.3 will be successful at every honest node. Thus, the SMR will proceed normally. Hence, during periods of synchrony, due to the liveness property of the SMR honest nodes will decide on the value proposed by L_r. Next, from Lemma 5.3, we know that whenever the SMR decides, the corresponding beacon output is reconstructible. Moreover, from §4.4, we know that every node that received a valid message during the agreement phase will multi-cast their decrypted shares along with its proof of correctness to every other node. This implies that every honest node will receive at least $t + 1$ valid decrypted shares to reconstruct the beacon output for epoch r. □

5.2 Unpredictability and Bias-Resistance

It follows from Theorem 2.2, Proposition 1 of [46], and knowledge soundness of dleq protocol that *w.h.p* the polynomials chosen by the adversarial nodes are independent of the polynomials chosen by the honest nodes. We will use this to argue that every beacon output includes an input of at least one honest node.

Proposition 5.5. *For any epoch r, if honest nodes decide on* `root` *and $\hat{p}(\cdot)$ be the underlying polynomial, then there exists an honest node such that*

$$\hat{p}(x) = p_i(x) + q(x)$$

where $p_i(\cdot)$ is the polynomial chosen by node i and $q(\cdot)$ is a polynomial of degree t independent of $p_i(\cdot)$. As a result, $\hat{p}(0)$ is uniformly random and independent of aggregated polynomial of any other epoch.

Proof. When an honest node decides `root`, by Lemma 5.1 and the collision resistance property of the hash function, at least $t+1$ honest nodes validate that contribution from at least $t+1$ nodes are included elements of the commitment vector $\hat{\mathbf{v}} = (\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n)$. Since, there are at most t malicious nodes, this implies that contribution from at least one honest node is included in at least $t+1$ elements of $\hat{\mathbf{v}}$.

Without loss of generality, let i be the honest node whose contribution is included in $t+1$ elements of $\hat{\mathbf{v}}$ and $p_i(\cdot)$ be the polynomial chosen by node i. By construction $\deg(p_i(\cdot)) = t$. Thus the $t+1$ evaluation points validated by the $t+1$ honest nodes fix the polynomial $p_i(\cdot)$ and hence all other evaluation points of $p_i(\cdot)$. Moreover, since the polynomials chosen by adversarial nodes are independent of $p_i(\cdot)$, this implies that evaluation of $p_i(\cdot)$ must be included at all other remaining elements $\hat{\mathbf{v}}$ as well. Hence, $\hat{p}(x) = p_i(x) + q(x)$. This immediately implies $\deg(q(\cdot)) \leq t$. □

Next we will use Proposition 5.5, the safety property of SMR and the fact that beacon output for every SMR decision is reconstructible by all nodes to prove that SPURT is bias-resistant.

Theorem 5.6. (Bias-Resistant) *Every* SPURT *output is uniformly random and is independent of any other beacon output.*

Proof. For any epoch r and height ht, from Proposition 5.5, we know that for every SMR decision on digest `root` corresponding to aggregated polynomial $\hat{p}(\cdot)$, $\hat{p}(0)$ is uniformly random and independent of aggregated polynomials of any other height. Moreover, from Lemma 5.3, a beacon output corresponding to every digest finalized by the atomic broadcast is reconstructible. This implies that SPURT is bias resistant. □

Theorem 5.7. (Unpredictability) SPURT *ensures unpredictability in the sense that as soon an adversary $\mathcal{A}$ learns any function of a beacon output, every honest party learns the beacon output within three communication delays.*

Proof. From Proposition 5.5, every beacon output includes a secret from at least one honest party. Let s^* be the secret of the honest node. Then, from Theorem 2.2, till an honest node starts reconstruction of the aggregated secret, s^* is indistinguishable from a uniformly randomly chosen element in $\mathbb{Z}_q$. Whenever an honest node starts the reconstruction phase, after a round trip delay all honest nodes start the reconstruction phase. Hence, all honest nodes will reconstruct the corresponding beacon output at most three communication delays later since the time the adversary learns the output. This implies that SPURT ensures unpredictability. □

Table 2: Summary of communication and computation cost of each epoch of SPURT. — indicate the no cost for the corresponding phase.

	Communication		Computation	
Protocol Phase	Leader	Non-leader	Leader	Non-leader
Commitment	$O(\lambda n^2)$	$O(\lambda n)$	—	$O(n)$
Aggregation	—	—	$O(n^2)$	—
Agreement	$O(\lambda n^2)$	$O(\lambda n)$	—	$O(n)$
Reconstruction	—	$O(\lambda n)$; $O(\lambda n \log n + n^2)$	—	$O(n)$

5.3 Public Verifiability

Public verifiability for beacon protocols producing true random numbers differs from beacon protocols producing pseudorandom numbers [2, 22, 44, 64]. In pseudorandom beacons, each beacon output is some deterministic function of the secret key generated during the initial setup phase. Hence, the output of such protocols can be efficiently verified given only the verification/public key corresponding to the secret key used for beacon generation. Contrary to this, truly random beacon protocols such as Scrape [24], Hydrand [61], and SPURT do not have a trusted setup phase, so their outputs are verified using the transcript of the interaction between nodes.

At any given height ht, let `root` be digest decided by honest nodes, and o_{ht} be the output of SPURT. Then, to verify the validity of o_{ht}, a user (need not be one of the nodes) queries up to $t+1$ nodes, either in sequence or in parallel, for the SMR decision certificate C_{ht} for height ht, leaves of the Merkle tree corresponding to `root`, and the $t+1$ valid reconstruction shares along with their proofs of correctness. Note that, from Theorem 5.4, every honest node will have these information at the end of the reconstruction phase. Upon receiving these information the external client can validate them by checking that:

- C_{ht} is a multi-signature of at least $2t+1$ nodes.
- `root` is a valid Merkle root of the received leaves.
- The dleq proofs of all the reconstruction shares are valid and consistent with `root` and the quorum certificate C_{ht}.

5.4 Performance

In this section, we analyze the communication cost of each epoch and the amortized cost of generating every beacon output. We will report the communication complexity in number of bits each node needs to send in every epoch. We then analyze the computation complexity of each node measured in of number of exponentiations each node needs to perform every epoch the amortized computation cost of each beacon output. Also, throughout this section, we will assume that signatures and multi-signatures are $O(\lambda)$ and $n + O(\lambda)$ bits long, respectively. Also, we assume that a node needs to perform $O(1)$ exponentiation to compute and validate a single signature, and $O(k)$ exponentiations to create and validate a multi-signature of k nodes. We summarize our performance analysis in Table 2.

Communication cost. During the commitment phase of an epoch r, each node sends $O(n)$ group elements to L_r. Thus, the commitment phase's total communication cost is $O(\lambda n^2)$. Next, during the agreement phase L_r sends back $O(n)$ group elements to every node. During the agreement phase, every node multi-casts a constant number of group elements to all other nodes. Hence, the agreement phase's total communication complexity is $O(\lambda n^2)$. Finally, during the reconstruction phase, in the fault-free case, every honest node sends only a constant number of group elements to every other honest node. In other scenarios, nodes might have to send the Merkle path ($O(\log n)$ group elements) from their share to the Merkle root, the aggregated signature, and the list of signers ($O(n)$ bits) to all other nodes. Hence, SPURT incur a total communication cost $O(\lambda n^2 \log n + n^3)$ bits in the worst case and $O(\lambda n^2)$ in the fault-free case.

Observe that during periods of synchrony, for every n epochs, there will be at least $\lceil 2n/3 \rceil$ honest leaders. From Theorem 5.4, for every honest leader, SPURT will produce an output. Hence, in every sequence of n epochs, SPURT will output at least $\lceil 2n/3 \rceil$ outputs. This implies that the *amortized* communication complexity of each beacon output is $O(\lambda n^2)$ in the fault-free case and $O(\lambda n^2 \log n + n^3)$ in the worst case.

Computation cost. During the commitment phase of an epoch r, each node performs $O(n)$ exponentiation to evaluate PVSS.Share for their chosen secret, and to sign the PVSS shares. In the aggregation phase, only L_r verifies the PVSS shares from all nodes. Since, verification of each node requires $O(n)$ exponentiations [24], L_r performs $O(n^2)$ exponentiation to verify all the PVSS shares. Computing the aggregated commitment and aggregated encryption requires $O(n^2)$ multiplications. Lastly, L_r performs $O(n)$ operations to construct the required Merkle tree. Overall, during the aggregation phase, the leader of the epoch performs $O(n^2)$ exponentiations and the remaining nodes do not perform any computation.

During the agreement phase, each node performs $O(n)$ exponentiations to validate signatures, and the aggregated polynomial. Finally, in the reconstruction phase, every node verifies $O(n)$ dleq proofs and possibly reconstructs a Merkle tree of size $O(n)$. Moreover, nodes might also need to aggregate $O(n)$ signatures. Hence, the computation cost per node in both agreement and reconstruction phase is $O(n)$.

In summary, in every epoch, the leader of the epoch performs $O(n^2)$ exponentiations whereas every other node performs $O(n)$ exponentiations. However, in a sequence of n epochs, each node becomes the leader only once, and due to pre-aggregation optimization (cf. §4.5), the leader gets $\tau = \Theta(n)$ rounds to compute $O(n^2)$ exponentiations. As a result, during periods of synchrony, the *amortized* computation cost of each beacon output is $O(n)$ exponentiation per node.

Public verification. Recall from §5.3, to validate a beacon output, external clients need to download the SMR BFT decision certificate, the leaves of the Merkle tree, the reconstruction shares, and the corresponding dleq proofs. Each of these messages is $O(n)$ group elements. Hence, the communication cost of verifying a beacon output is $O(\lambda n)$. Moreover, verifying all these messages require $O(n)$ exponentiations. Verifying $O(n)$ signatures and dleq proofs, constructing the Merkle tree, and reconstruction of the secret all have a computation cost of $O(n)$, so the computation complexity of public verification is $O(n)$.

Latency. During periods of synchrony, when an honest node is chosen as the leader of an epoch, SPURT produces a beacon output. Thus, in the fault-free case, in practice, SPURT would only require five message delays. However, there might be a sequence of t malicious leaders in the worst case, and all of them may decide to abort their epochs. In such cases, SPURT will take $O(t)$ message delay to produce the next output. Nevertheless, since in a sequence of n epochs, SPURT will produce at least $2n/3$ beacon outputs, the amortized latency of SPURT is 1.5 epochs.

6 IMPLEMENTATION & EVALUATION

We have implemented a prototype of SPURT using the go programming language version 1.9.0. Our implementation builds atop the open-source Quorum client version 2.4.0. Quorum is a fork of Ethereum go client and implements the Istanbul BFT protocol as one of its consensus protocol. Istanbul BFT [54] is a variant of PBFT protocol with a total quadratic communication complexity both during view change and steady-state.

Throughout our implementation, we have used the ed25519 elliptic curve for our cryptographic primitive and have used the filippo.io/edwards25519 [3] package for primitive elliptic curve operations. Thus, throughout our evaluation, the security parameter λ is 256 bits. For multi-exponentiations, we have used the native implementation of [3], which implements the interleaved window method. Also, we disable the IBFT parameter to artificially control the time between two consecutive proposals and modify the underlying implementation such that the next leader proposes its value as soon as the previous beacon output is finalized.

6.1 Experimental Setup

We evaluate our implementation of SPURT with varying nodes, i.e., 16, 32, 64, and 128. We run all nodes on Amazon Web Services (AWS) *t3a.medium* virtual machine (VM) with one node per VM. All VMs have two vCPUs, 4GB RAM and 5.0 GB/s network bandwidth. The operating system for each VM is Ubuntu 20.04.

To simulate an execution over the internet, we pick eight different AWS regions, namely, Canada, Ireland, London, N. California, N. Virginia, Paris, Singapore, and Tokyo. For any given choice of the total number of nodes, we distribute the nodes evenly across all eight regions. We create an overlay network among nodes where all nodes are pair-wise connected, i.e., they form a complete graph or any given network size.

6.2 Evaluation Results

All our evaluation results are averaged over three runs for each value of number of nodes.

Throughput. We report the throughput of SPURT as the number of beacon output generated per minute in Figure 5. With 16, 32, and 64 nodes, SPURT on average can generate 4, 2, and 1 beacon output per second. With 128 nodes, SPURT takes about 3 seconds to generate one beacon output.

One interesting thing we observe is that, despite the fact that Hydrand [61] and SPURT have similar communication cost in the fault-free case operation, i.e., without any adversarial attacks, throughput

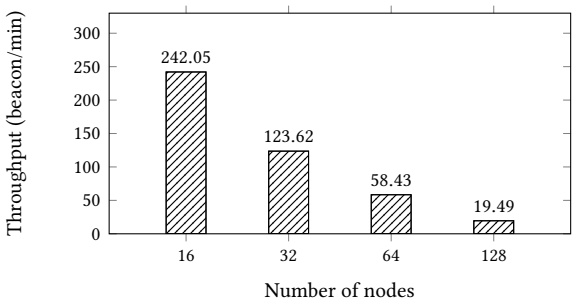

Figure 5: Average number of random beacon generated per minute with varying number of nodes.

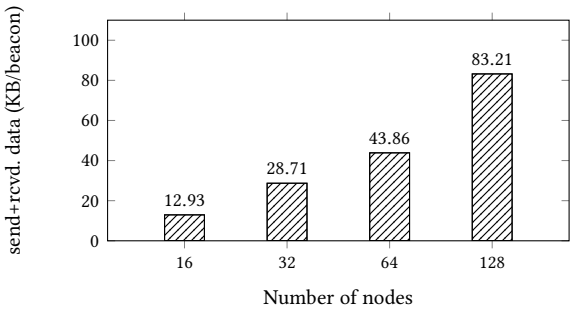

Figure 6: Average bandwidth usage (send + received data) measured in Kilobytes per beacon output with varying number of nodes.

	Data sent (bits)		n = 64 (KBytes)	
Protocol	Leader	Non-leader	Leader	Non-leader
Scrape [24]	—	BB($4\lambda n + 2\lambda$)	—	—
Hydrand [61]	$(10/3)\lambda n^2 + 7\lambda n$	$14\lambda n$	450	28
SPURT	$(10/3)\lambda n^2 + 5\lambda n$	$9\lambda n$	442	18

Table 3: Bandwidth usage per node per beacon output. Here BB(x) is the bandwidth usage of byzantine broadcast of a message of size x.

	Time taken (in milliseconds)			
Protocol Phase	n = 16	n = 32	n = 64	n = 128
Commitment	6.0	12.14	24.6	51.18
Aggregation	48.63	180.57	718.22	2811.18
Agreement	7.63	13.86	27.33	54.16
Reconstruction	7.83	14.11	27.87	55.67

Table 4: Time taken (in milliseconds) to compute different cryptographic functions required in the different phases of SPURT. All measurements performed using a *t3a.medium* AWS EC2 instance.

of SPURT is about 3 or more times higher than Hydrand [61]. We believe this is because SPURT is partially synchronous, i.e., it does not require any network delay parameter and hence can make progress at the speed of true network delay. In the literature, this is also referred to as the *responsiveness* property of protocols [67].

Bandwidth usage. We report the bandwidth usage measured as the amount of bytes sent and received per node per beacon output in Figure 6. Recall from §5 that at every epoch, each node sends and receives a total of $O(n\lambda)$ bits of information to and from other nodes. Hence, with an increase in the number of nodes, we observe an approximately linear increase in the bandwidth usage per node per beacon output. For example, from 64 to 128 nodes, the average bandwidth usage per node per beacon output increases from 43 to 83 Kilobytes.

In Table 3, we present the number of bits sent by each node during the fault-free case operation of SPURT, and compare it against Scrape and Hydrand. We calculate these numbers based on their protocol specifications, and we measure the cost of sending both scalars and group-elements as λ bits to avoid any discrepancies in implementation details. Since Scrape uses a broadcast channel in a black-box manner, we measure its cost as the number of bits each node needs to send over the broadcast channel. Hence, we denote it as BB(·). For concreteness, we calculate this value for the 64 node network. We note that each node in SPURT sends 18 Kilobytes of data in comparison to 28 kilobytes in Hydrand; the leader in SPURT sends 442 Kilobytes of data, whereas the leader in Hydrand sends 450 Kilobytes of data.

Computation cost. Table 4 presents the time required for each of the four phases. Except for the aggregation phase, the other three phases take less than 0.1 seconds. Furthermore, for these phases, the computation time increases linearly with the number of nodes. The aggregation phase requires the leader to perform a quadratic number of exponentiations, so the computation time is higher than the other phases. Nevertheless, since we pipeline the aggregation phase by sending the commitments to the leader in advance (cf. §4.5), the aggregation phase is not the bottleneck in the critical path.

	# exponentiations		n = 64	
Protocol	Leader	Non-leader	Leader	Non-leader
Scrape [24]	—	$(15/4)n^2 + 13n + 3$	—	16848
Hydrand [61]	$4n + 4$	$3n$	262	192
SPURT	$n^2 + 3n$	$5n + 5$	4416	327

Table 5: Computation cost measured in number of exponentiations per node per beacon output in the normal case operation.

In Table 5, we present the number of exponentiations each node needs to perform for any given epoch during a fault-free operation and compare it with the number of exponentiations required for fault-free operation in Scrape and Hydrand. In Scrape for every beacon output, each node performs a quadratic number of exponentiations. In SPURT only the leader performs a quadratic number of exponentiation whereas all other nodes perform a linear number of exponentiations. In Hydrand, both leader and the remaining nodes perform a linear number of exponentiations, and it is about 40% less than what non-leader performs in SPURT. Nevertheless, we believe this higher computational cost in SPURT is a favorable trade-off for tolerating partially synchronous network, offering strong unpredictability and avoiding the need for an expensive initialization (cf. §7).

7 RELATED WORK

Based on the setup assumption, existing distributed protocols can be classified into two categories; protocols with *trusted* setup and with *transparent* setup.

The protocols with trusted setup involve generation of public parameters that embed a secret trapdoor. These parameters can either be generated by a *trusted* third party (hence the name trusted setup) or can be generated by running a maliciously secure multi-party computation protocol, often a Distributed Key Generation (DKG) step. The security of the protocol with a trusted setup relies crucially on the assumption that the adversary does not have access to the secret trapdoor. Existing protocols in this category include [2, 15, 22, 28, 37, 44, 64]. Note that even protocols without a trusted setup assumption may also require an initial step to generate public parameters. The subtlety is that the public parameters are uniformly random elements from certain groups. This is a milder assumption than the trusted setup assumption described above, where the public parameters hide information about the trapdoor. Protocols in the transparent setup category include [10, 24, 31, 40, 46, 55, 61].

The protocols due to Cachin et al [22], Dfinity [44], Drand [2], and Aleph [37] require a threshold signing key [19] to be shared among all the nodes as per the shamir threhsold secret sharing scheme [63]. Cachin et al. and Aleph can tolerate asynchronous network whereas Drand and Dfinity require the network to be synchronous. The random beacon's output at any given epoch is the BLS signature on the hash of the epoch number. Cachin et al., and Dfinity do not provide specific details of the DKG phase of the protocol. Drand uses the DKG protocol of Gennaro et al. [38], which requires a Byzantine broadcast channel over which each node sends $O(\lambda n)$ bits to information. Hence, the overall communication complexity of the setup phase of Drand is $O(\lambda n^3 \log n)$. After the DKG step, the communication complexity for generating one beacon output for all three protocol is $O(n^2\lambda)$. Aleph [37] presents their own DKG protocol with a total communication complexity of $O(\lambda n^4 \log n)$.

Most relevant to our protocol are Scrape [24] and Hydrand [61]. Both Scrape and Hydrand assume that the underlying network is synchronous. Scrape [24] improves the computation complexity of PVSS protocol of [62] from $O(n^2)$ exponentiation to $O(n)$ exponentiation per PVSS, and uses their PVSS along with a broadcast channel to generate distributed randomness. In particular, for every beacon output in Scrape, each node uses the broadcast channel to share their secret. Once $t+1$ nodes share their secret, nodes reconstruct the secrets using the reconstruction phase of the PVSS and combine them to produce the beacon output. In Scrape, for each beacon, each node uses the broadcast channel to share $O(n\lambda)$ size message to all other nodes. Thus, the total communication cost of at least $O(n^3 \log n)$. Also, for every beacon output, each node requires to perform $O(n^2)$ exponentiations.

Hydrand [61] modifies the Scrape protocol to remove the implicit assumption of a broadcast channel. Unlike Scrape, in each epoch of Hydrand, only one node, a designated leader shares a secret using Scrape's PVSS scheme. Hydrand has an initialization step where each node shares a secret using PVSS scheme, which costs $O(\lambda n^3 \log n)$ communication wise. After the setup phase, for every beacon, Hydrand has an amortized total communication and computation cost of $O(\lambda n^2)$ and $O(n)$, respectively. One major disadvantage of Hydrand is that it only provides probabilistic unpredictability, i.e., at any epoch, an adversary can predict beacon output for up to t future epochs. In contrast, SPURT has a fault-free complexity of $O(\lambda n^2)$, provides perfect unpredictability, and does not require an initialization step.

A concurrent and independent work Bhat et al., GRandPiper [15] improves upon Hydrand and increases its fault tolerance to one-half, but requires a trusted setup to generate q-SDH parameters. To overcome the lack of perfect unpredictability in GRandPiper (and Hydrand), Bhat et al [15] presents BRandPiper, where the leader share n secrets in a single epoch and has the nodes reconstruct a random value accumulating secrets chosen by $t+1$ distinct nodes. Similar to GRandPiper, BRandPiper also requires a trusted setup to generate q-SDH parameters, and has a worst-case communication complexity of $O(\lambda n^3)$.

Randherd [64] uses Randhound in a one-time setup to partition nodes into smaller subgroups of size c, and additionally setup keys for threshold signatures. The total complexity of Randherd is $O(\lambda c^2 \log n)$. Randherd, as presented, is not bias-resistant as a malicious leader can abort the protocol after observing the beacon output, and will require additional mechanisms to make it bias-resistant. Also, in general, although the protocols that rely on random committee selection increase efficiency; they typically tend to have (slightly) reduced fault tolerance and can be applied to improve many other protocols described in this section. It is thus not instructive to directly compare the efficiency of sampling-based and non-sampling-based protocols.

In addition to the above mentioned protocols, other beacon protocols include Proof-of-Work (PoW) [55], Proof-of-Delay [21], Algorand [40], Ouroboros [46], Ouroboros Praos [31], etc. The PoW, Algorand and Ouroboros Praos are not bias-resistant as a malicious adversary can decide to discard undesirable beacon outputs. The Proof-of-Delay based protocol relies on strong and new assumptions about verifiable time-lock puzzles [12, 25] or Verifiable Delay Functions [18]. The Ouroboros [46] protocol requires every node to perform PVSS over a broadcast channel, and hence has high communication complexity.

8 CONCLUSION AND FUTURE DIRECTIONS

We have presented SPURT, an efficient distributed randomness beacon protocol with transparent setup, i.e., trapdoor-free public parameters. SPURT guarantees that each beacon output is unpredictable, bias-resistant and publicly verifiable, and provides these properties in a partially synchronous network against a malicious adversary controlling up to one third of the total nodes. In the fault-free case of operation, SPURT has amortized total communication of $O(\lambda n^2)$. In the worst case, the amortized total communication cost is $O(\lambda n^2 \log n + n^3)$. (Note that for $\lambda = 256$ and $n < 2950$, $n^3 < \lambda n^2 \log n$.) Computation wise, each node performs $O(n)$ group exponentiations per beacon output.

An interesting question for future work is whether it is possible to design a randomness beacon protocol with optimal fault tolerance and *sub-quadratic* communication complexity (possibly with a trusted setup). Note that protocols that sample subsets can

be easily made sub-quadratic in the trusted setup phase. But such protocols come with reduced fault tolerance. It is interesting to study whether we can design a sub-quadratic protocol that does not resort to subset sampling. On the flip side, it would also be very interesting to show some sort of communication lower bound for randomness beacon. Similar lower bounds for Byzantine agreement or multiparty computation may be good starting points towards that direction.

One may also try to extend SPURT to fully asynchronous networks. The major hurdle we encounter is that consensus (SMR) protocols in fully asynchronous network require shared randomness [36], which creates a circularity.

ACKNOWLEDGMENTS

The authors would like to thank Amit Agarwal, Jong Chan Lee, Zhuolun Xiang, and Tom Yurek for numerous discussions related to the paper.

REFERENCES

[1] 1969. Draft lottery (1969). (1969). https://en.wikipedia.org/wiki/Draft_lottery_(1969)
[2] 2020. Drand - A Distributed Randomness Beacon Daemon. (2020). https://github.com/drand/drand.
[3] 2020. filippo.io/edwards25519. (2020). https://pkg.go.dev/filippo.io/edwards25519
[4] 2020. Go Ethereum: Official Golang implementation of the Ethereum protocol. (2020). https://github.com/ethereum/go-ethereum
[5] 2020. Proof of Stake (PoS). (2020). https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/
[6] 2020. Quorum: A permissioned implementation of Ethereum supporting data privacy. (2020). https://github.com/ConsenSys/quorum
[7] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2019. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 317–326.
[8] Ben Adida. Helios: Web-based Open-Audit Voting.
[9] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A sharded smart contracts platform. (2017).
[10] Sarah Azouvi, Patrick McCorry, and Sarah Meiklejohn. 2018. Winning the caucus race: Continuous leader election via public randomness. *arXiv preprint arXiv:1801.07965* (2018).
[11] Thomas Baigneres, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrede Lepoint, and Matthieu Rivain. 2015. Trap Me If You Can-Million Dollar Curve. *IACR Cryptol. ePrint Arch.* 2015 (2015), 1249.
[12] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. 2020. CRAFT: Composable Randomness and Almost Fairness from Time. Cryptology ePrint Archive, Report 2020/784. (2020). https://eprint.iacr.org/2020/784.
[13] Richard Bellman and E. G. Straus. 1964. 5125. *The American Mathematical Monthly* 71, 7 (1964), 806–808. http://www.jstor.org/stable/2310929
[14] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. 2016. Dual EC: A standardized back door. In *The New Codebreakers*. Springer, 256–281.
[15] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2020. RandPiper–Reconfiguration-Friendly Random Beacons with Quadratic Communication. (2020).
[16] George Robert Blakley. 1979. Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*. IEEE, 313–318.
[17] Manuel Blum. 1983. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News* 15, 1 (1983), 23–27.
[18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.
[19] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.
[20] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. 2015. On Bitcoin as a public randomness source. *IACR Cryptol. ePrint Arch.* 2015 (2015), 1015.
[21] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. 2017. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)* (2017).
[22] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.
[23] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 42–51.
[24] Ignacio Cascudo and Bernardo David. 2017. SCRAPE: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*. Springer, 537–556.
[25] Ignacio Cascudo and Bernardo David. 2020. ALBATROSS: publicly AttestabLe BATched Randomness based On Secret Sharing. Cryptology ePrint Archive, Report 2020/644. (2020). https://eprint.iacr.org/2020/644.
[26] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. 173–186.
[27] David Chaum and Torben Pryds Pedersen. 1992. Wallet databases with observers. In *Annual International Cryptology Conference*. Springer, 89–105.
[28] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. 2019. Homomorphic Encryption Random Beacon. (2019).
[29] Ran Cohen and Yehuda Lindell. 2017. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology* 30, 4 (2017), 1157–1186.
[30] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. 2019. YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*.
[31] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 66–98.
[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The second-generation onion router*. Technical Report. Naval Research Lab Washington DC.
[33] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.
[34] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.
[35] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 186–194.
[36] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.
[37] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.
[38] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2007. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology* 20, 1 (2007), 51–83.
[39] Mainak Ghosh, Miles Richardson, Bryan Ford, and Rob Jansen. 2014. *A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays*. Technical Report. NAVAL RESEARCH LAB WASHINGTON DC.
[40] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.
[41] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. 2003. *Herbivore: A scalable and efficient protocol for anonymous communication*. Technical Report. Cornell University.
[42] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 218–229.
[43] David Goulet and George Kadianakis. 2015. Random number generation during Tor voting. *Tor's protocol specifications-Proposal* 250 (2015).
[44] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548* (2018).
[45] Somayeh Heidarvand and Jorge L Villar. 2008. Public verifiability from pairings in secret sharing schemes. In *International Workshop on Selected Areas in Cryptography*. Springer, 294–308.
[46] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.
[47] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 583–598.
[48] LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.
[49] Arjen K Lenstra and Benjamin Wesolowski. 2015. A random zoo: sloth, unicorn, and trx. (2015).

[50] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 17–30.

[51] Robert J. McEliece and Dilip V. Sarwate. 1981. On sharing secrets and Reed-Solomon codes. *Commun. ACM* 24, 9 (1981), 583–584.

[52] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[53] Bodo Möller. 2001. Algorithms for multi-exponentiation. In *International Workshop on Selected Areas in Cryptography*. Springer, 165–180.

[54] Henrique Moniz. 2020. The Istanbul BFT Consensus Algorithm. *arXiv preprint arXiv:2002.03613* (2020).

[55] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[56] David Pointcheval and Jacques Stern. 1996. Security proofs for signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 387–398.

[57] Michael O Rabin. 1983. Transaction protection by beacons. *J. Comput. System Sci.* 27, 2 (1983), 256–267.

[58] Tal Rabin and Michael Ben-Or. 1989. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 73–85.

[59] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[60] Alexandre Ruiz and Jorge L Villar. 2005. Publicly verifiable secret sharing from Paillier's cryptosystem. In *WEWoRC 2005–Western European Workshop on Research in Cryptology*. Gesellschaft für Informatik eV.

[61] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. 2020. Hydrand: Practical continuous distributed randomness.

[62] Berry Schoenmakers. 1999. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*. Springer, 148–164.

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

[64] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. 2017. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 444–460.

[65] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 137–152.

[66] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in numbers: Making strong anonymity scale. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 179–182.

[67] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 347–356.

[68] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 931–948.

A LINEAR ERROR CORRECTING CODE

Let C be a $[n, k, d]$ linear error correcting code over $\mathbb{Z}_q$ of length n and minimum distance d. Also, let $C^\perp$ be the dual code of C i.e., $C^\perp$ consists vectors $\mathbf{y}^\perp \in \mathbb{Z}_q^n$ such that for all $\mathbf{x} \in C$, $\langle \mathbf{x}, \mathbf{y}^\perp \rangle = 0$. Here, $\langle \cdot, \cdot \rangle$ is the inner product operation. SPURT uses the basic fact from coding theory. Refer to Lemma 1 of [24] for its proof.

Lemma A.1. *If $\mathbf{x} \in \mathbb{Z}_q^n \setminus C$, and $\mathbf{y}^\perp$ is chosen uniformly at random from $C^\perp$, then the probability that $\langle \mathbf{x}, \mathbf{y}^\perp \rangle = 1$ is exactly $1/q$.*

Throughout this paper, we will use C to be the $[n, k, n-k+1]$ Reed-Solomon Code of the form

$$C = \{p(1), p(2), ..., p(n) : p(x) \in \mathbb{Z}_q[x]; \text{and } \deg(p(\cdot)) \leq k-1\}$$

Table 6: Notations used and their descriptions

Notation	Description
$g, h \in \mathbb{G}$	Two uniform random generators of $\mathbb{G}$
λ	security parameter
n	Total number of nodes
t	Maximum number of byzantine nodes
pk_i, sk_i	Public and Secret key of i^{th} node.
r, ht	epoch and height
s_i	Secret chosen by i^{th} node at epoch r
$p_i(\cdot)$	Polynomial chosen by i^{th} node to share s_i
$s_{i,j}$	$p_i(j)$, i.e, $p_i(\cdot)$ evaluated at j
$v_{i,j}$	Commitment of $s_{i,j}$ computed as $g^{s_{i,j}}$
$c_{i,j}$	Encryption of $s_{i,j}$ under pk_j computed as $pk_i^{s_{i,j}}$
dleq$(\cdot)$	NIZK proof for equality of discrete logarithm
$\langle \mathbf{x}, \mathbf{y} \rangle$	Inner product of two vectors x and y
$[n]$	Set $\{1, 2, \ldots, n\}$
$C^\perp$	Dual of error correcting code C
L_r	Leader of epoch r

where $\deg(p(\cdot))$ is the degree of the polynomial $p(\cdot)$. Thus its $[n, n-k, k+1]$ dual code $C^\perp$ can be written as

$$C^\perp = \{(\mu_1 f(1), \mu_2 f(2), ..., \mu_n f(n); f(x) \in \mathbb{Z}_q[x]; \text{and } \deg(f(\cdot)) \leq n-k+1\}$$

where the coefficients $\mu_i = \prod_{i=1, i\neq j}^{n} \frac{1}{i-j}$. This implies that random elements from $C^\perp$ of interest is efficiently samplable.

B INDISTINGUISHABILITY OF SECRETS

Intuitively, for any $(n, t+1)$ PVSS scheme, IND1-secrecy ensures that prior to the reconstruction phase, the public information together with the secret keys sk_i of any set of at most t players gives no information about the secret. Formally this is stated as in the following indistinguishability based definition adapted from [45, 60]:

Definition B.1. (IND1-Secret) A $(n, t+1)$ PVSS is said to be IND1-secret if for any probabilistic polynomial time adversary $\mathcal{A}$ corrupting at most t parties, $\mathcal{A}$ has negligible advantage in the following game played against an challenger.

(1) The challenger runs the Setup phase of the PVSS as the dealer and sends all public information to $\mathcal{A}$. Moreover, it creates secret and public keys for all honest nodes, and sends the corresponding public keys to $\mathcal{A}$.
(2) $\mathcal{A}$ creates secret keys for the corrupted nodes and sends the corresponding public keys to the challenger.
(3) The challenger chooses values s_0 and s_1 at random in the space of secrets. Furthermore it chooses $b \leftarrow \{0, 1\}$ uniformly at random. It runs the phase of the protocol with s_b as secret. It sends $\mathcal{A}$ all public information generated in that phase, together with s_b.

The advantage of $\mathcal{A}$ is defined as $|\Pr[b = b'] - 1/2|$.