RandPiper – Reconfiguration-Friendly Random Beacons with Quadratic Communication

Adithya Bhat*
bhat24@purdue.edu
Purdue University

Nibesh Shrestha*
nxs4564@rit.edu
Rochester Institute of Technology

Zhongtang Luo
zhtluo@gmail.com
Purdue University

Aniket Kate
aniket@purdue.edu
Purdue University

Kartik Nayak
kartik@cs.duke.edu
Duke University

ABSTRACT

A random beacon provides a continuous public source of randomness and its applications range from public lotteries to zero-knowledge proofs. Existing random beacon protocols sacrifice either the fault tolerance or the communication complexity for security, or ease of reconfigurability. This work overcomes the challenges with the existing works through a novel communication efficient combination of state machine replication and (publicly) verifiable secret sharing (PVSS/VSS).

For a system with n nodes in the synchronous communication model and a security parameter κ, we first design an optimally resilient Byzantine fault-tolerant state machine replication protocol with $O(\kappa n^2)$ bits communication per consensus decision *without* using threshold signatures. Next, we design GRandPiper (Good Pipelined Random beacon), a random beacon protocol with bias-resistance and unpredictability, that uses PVSS and has a communication complexity of $O(\kappa n^2)$ *always*, for a static adversary. However, GRandPiper allows an adaptive adversary to predict beacon values up to $t + 1$ epochs into the future. Therefore, we design BRandPiper (Better RandPiper), that uses VSS and has a communication complexity of $O(\kappa f n^2)$, where f is the *actual* number of faults, while offering a strong unpredictability with an advantage of only a single round even for an adaptive adversary. We also provide reconfiguration mechanisms to restore the resilience of the beacon protocols while still maintaining quadratic communication complexity per round. We implement BRandPiper and compare it against the state of the art practically deployed beacon protocol, *drand*, and show that we are always better than or equal to in performance.

1 INTRODUCTION

Public digital randomness is essential across a large spectrum of security applications ranging from e-voting to blockchains. Its practical relevance further evident from NIST's Randomness Beacons project [21] and from the recent emergence of Drand Organization [25]. In addition, several other proposals [17, 20, 24, 29, 30, 42, 43, 47] and implementations [1, 2, 28] offer *random beacons* [39].

A random beacon emits a new *random* value at intermittent intervals such that the emitted values are bias-resistant, i.e., no entity can influence a future random beacon value, and unpredictable, i.e., no entity can predict future beacon value. Clearly, we cannot trust a single node to offer such a service — the node can easily affect both bias-resistance and unpredictability of the beacon. A series of recent work has instead relied on distributing the trust across multiple nodes such that even if a subset of nodes gets compromised, the beacon is still secure [17, 30, 43, 47].

In a system consisting of n nodes, tolerating t Byzantine faults, with security parameter κ, an ideal distributed randomness beacon protocol, in addition to being bias-resistant and unpredictable, should have the following properties: (i) optimal resilience, (ii) low communication overhead, (iii) reconfiguration friendliness (allowing efficient addition and removal of nodes), and (iv) use efficient cryptographic schemes as opposed to computationally expensive schemes such as Proof-of-Work (PoW) or Verifiable Delay Functions (VDFs). Existing works trade one or the other of the above features expected from a random beacon. For instance, HydRand [43] sacrifices optimal resilience ($t < n/3$) for better communication complexity ($O(\kappa n^2)$ in the best case and $O(\kappa n^3)$ in the worst case) with minimal setup assumptions. Cachin et al. [17] provides a communication complexity of $O(\kappa n^2)$, but it requires a threshold (cryptographic) setup and hence cannot support a reconfiguration of the system without changing the threshold setup through proactive secret sharing techniques. Several other solutions [24, 42] use computationally expensive mechanisms such as VDFs[1] [13] where nodes compute VDF function constantly to ensure security of the beacon.

In this paper, we ask whether we can design an optimally resilient random beacon protocol that achieves good communication complexity while using efficient cryptographic schemes and a re-usable setup, i.e., avoiding setups such as those of threshold signatures where the entire setup needs to be re-generated when a participating node is replaced. To answer this question, we first design an optimally resilient Byzantine fault-tolerant (BFT) state machine replication (SMR) protocol with $O(\kappa n^2)$ communication complexity per consensus decision while requiring a structured reference string (SRS) [34] setup that allows any bounded reconfiguration. Next, we present two random beacon protocols GRandPiper (Good Pipelined Random beacon) and BRandPiper (Better Pipelined Random beacon) using our BFT SMR protocol as a building block and provide similar guarantees. GRandPiper is communication efficient ($O(\kappa n^2)$ in the best and worst case) but allows an adaptive adversary to predict $t+1$ epochs into the future. BRandPiper offers stronger unpredictability guarantees, but has a communication complexity of $O(\kappa f n^2)$ where

*Contributed equally and listed alphabetically

[1]VDFs require computing operations such as squarings, which are energy intensive.

f is the actual number of faults. Finally, we present a communication efficient reconfiguration protocol to add nodes to the system while maintaining quadratic communication complexity per round.

1.1 Efficient State Machine Replication Without Threshold Signatures

There has been a long sequence of work in improving communication complexity of consensus protocols [3, 6, 15, 26, 33, 35, 49]. In the synchronous SMR setting, the optimal communication complexity per consensus decision of an SMR protocol is $O(\kappa n^2)$ bits [3, 5, 35, 46]. However, all of these solutions use threshold signatures. Our first result improves upon the communication complexity in the absence of threshold signatures. Specifically, we show the following:

Theorem 1.1. *Assuming public-key infrastructure and a universal structured reference string setup under q-SDH assumption, there exists a state machine replication protocol with amortized $O(\kappa n^2)$ communication complexity per consensus decision tolerating $t < n/2$ Byzantine faults.*

To be precise, the protocol incurs $O(\kappa n^2)$ communication complexity under q-strong Diffie-Hellman (SDH) assumption [14] (whose parameters can be generated using distributed protocols) or $O(\kappa n^2 \log n)$ without it. Getting rid of threshold signatures allows for efficient reconfiguration of the participating nodes and does not require generating threshold keys each time a new node joins the system. It is in this sense that our system is reconfiguration-friendly. Thus, an efficient BFT protocol in this setting is of independent interest. We reduce communication by making use of efficient erasure coding schemes [40] and cryptographic accumulators [10] to efficiently broadcast large messages at the expense of increase in latency of SMR protocol. As we will see, the increase in latency does not affect our random beacon protocols adversely.

1.2 RandPiper – Random Beacon Protocols

RandPiper is a suite of random beacon protocols that use our SMR protocol as a building block. We present two protocols: GRandPiper (Good Pipelined Random beacon) and BRandPiper (Better Pipelined Random beacon) which differ in unpredictability and communication complexity. In both protocols, we use secret sharing schemes to privately commit random numbers ahead of time. This ensures bias-resistance as the random number once shared cannot be changed. For unpredictability, we ensure that the beacon outputs are generated using inputs from $t+1$ nodes (where t is the threshold of Byzantine nodes) at least one of which is truly random, and therefore the output is truly random.

GRandPiper. In GRandPiper, we explore how to build a communication optimal random beacon protocol with bias-resistance and strong unpredictability, allowing a static adversary to predict up to a security parameter number of epochs into the future. In particular we show the following:

Theorem 1.2 (Informal). *Assuming public-key infrastructure and a universal structured reference string setup under q-SDH assumption, there exists a reconfiguration friendly, bias-resistant, and $O(\min(\kappa, t))$-absolute unpredictable (see Definition 3.2) random beacon protocol tolerating $t < n/2$ Byzantine faults with $O(\kappa n^2)$ communication per beacon output.*

Our GRandPiper protocol outputs a random beacon with $O(\kappa n^2)$ communication complexity per beacon output, where κ is the security parameter. The output of the beacon protocol is bias-resistant and satisfies strong unpredictability against a static adversary, i.e., the probability of a static adversary predicting c rounds into the future is less than 2^{-c} (in expectation this is 2 rounds into the future). After $t+1$ epochs, an adversary cannot predict beacons into the future with non-negligible probability for cases when t is smaller than κ. We concisely term this as $O(\min(\kappa, t))$-absolute unpredictable protocol. We also do not need any threshold setups, which allows nodes to join and leave the system easily without stopping our protocol.

At a high-level, our protocol uses Publicly Verifiable Secret Sharing (PVSS) schemes, and allows a leader to input an $O(\kappa n)$-sized PVSS encryptions $\mathcal{PVSS}.\vec{E}$ into the SMR to share a single secret per epoch. The secret shared by the leader will be reconstructed when the same node is chosen as the leader again. To ensure that eventually there is an honest leader, a leader does not repeat again for the next t epochs. This also ensures that our BFT SMR protocol decides on the proposed shares once we get an honest leader. Our construction ensures that we *always* have a communication complexity of $O(\kappa n^2)$ for the beacon, as the beacon keeps outputting values based on buffered PVSS shares, and we remove Byzantine nodes to avoid the buffer from ever being empty.

However, an adaptive adversary can predict $t+1$ epochs into the future in GRandPiper by simply corrupting the next t leaders and learning their committed secrets. At this point, continuing to use the PVSS scheme to improve the unpredictability leads to a loss of quadratic communication complexity. Hence, we look in a different direction to improve unpredictability.

BRandPiper. In BRandPiper, we explore how to achieve the best possible unpredictability while having the best possible communication complexity and also supporting reconfiguration. In particular, we show the following result:

Theorem 1.3 (Informal). *Assuming public-key infrastructure and a universal structured reference string setup under q-SDH assumption, there exists a reconfiguration-friendly, bias-resistant and 1-absolute unpredictable (see Definition 3.2) adaptively secure random beacon protocol tolerating $t < n/2$ Byzantine faults with $O(\kappa f n^2)$ communication, where $f \leq t$ is the actual number of faults.*

Our second protocol BRandPiper outputs a random beacon with $O(\kappa f n^2)$ communication per output, and guarantees bias-resistance and strong unpredictability against an adaptive adversary. Here, f is the actual number of faults; when $f = O(1)$, the protocol enjoys $O(\kappa n^2)$ communication complexity. BRandPiper uses random inputs from $> t$ nodes in every epoch, ensuring strong unpredictability of only 1 epoch into the future.

As a building block, we first construct an improved VSS (iVSS) protocol by modifying the state-of-the-art VSS scheme eVSS [32]. Compared to eVSS, which requires $O(\kappa n + \kappa f + \kappa f n)$ information on the bulletin board (broadcast channel), iVSS posts only $O(\kappa n)$ bits of information on the bulletin board which in effect improves the amortized communication complexity of the VSS scheme to $O(\kappa f n^2)$ where f is the *actual* number of faults. This may be of independent interest in applications requiring batched VSS sharings.

At a high level, we use round-robin leaders and iVSS in point-to-point channels to secret share n random numbers in every epoch. Since we are producing n shares every epoch, we can now consume n shares in every epoch. Thus, in every epoch, using the homomorphic properties of VSS secret shares, we reconstruct a homomorphic sum of n shares in every epoch, thus eliminating the $t + 1$ epoch advantage held by the adaptive adversary and reducing it to just 1 epoch. We carefully design the protocol so that we have a communication complexity of $O(\kappa f n^2)$. Our key insight in BRandPiper is that a leader can efficiently secret share n shares at once instead of one. These shares are buffered by all nodes, and it ensures that there are always sufficient shares available for reconstruction in the next n epochs so far as leaders are chosen in a round-robin manner. The buffering helps prevent a Byzantine node from biasing by refusing to share new blocks, when the outputs are unfavorable. Without our techniques, while assuming threshold signatures, existing VSS protocols have an optimistic communication complexity of $O(\kappa n^2)$ [32] and a worst case communication complexity of $O(\kappa n^3)$ to perform 1 secret sharing. The difference in the order arises from opening f shares for every node that complains against the leader. BRandPiper shows how to perform $O(n)$ VSS secret sharings with a communication complexity of $O(\kappa f n^2)$ which is quadratic when $f = O(1)$.

1.3 Efficient Reconfiguration

While prior works [17, 43] provide a random beacon protocol with $O(\kappa n^2)$ communication without threshold signatures and claim to be reconfiguration-friendly, they do not provide any reconfiguration mechanisms. In this work, we provide reconfiguration protocols to restore the resilience of our beacon protocol when some Byzantine nodes have been removed from the system. Since we do not rely on threshold signatures, new nodes can join the system without generating new keys for all nodes. Moreover, the reconfiguration protocol is executed while still maintaining quadratic communication complexity per round.

Synchronizing new joining nodes during reconfiguration while keeping low communication overhead is a challenge. Prior clock synchronization protocols [3] incur $O(\kappa n^3)$ communication without threshold signatures, and moreover, the execution cannot be split across rounds to reduce per round complexity. We introduce a new clock synchronization primitive that synchronizes new nodes when a majority of honest nodes are already synchronized while maintaining quadratic communication per round. The protocol utilizes homomorphic addition property of VSS secret shares that yields constant-sized secrets when the secret is opened. The homomorphic secret can be broadcast among all nodes to synchronize all the nodes with only $O(\kappa n^2)$ communication.

Implementation and Evaluation. We implement our protocol and demonstrate the practicality of our random beacon. We show that our BRandPiper protocol is as good as the state of the art practically deployed system: *drand* in terms of beacons per minute. Concretely, we show that choosing a Δ value for BRandPiper such that it always succeeds, we are always better than drand if we assume a similar low Δ value for drand. Giving a benefit of doubt to drand, by choosing slightly relaxed value of 99.9^{th} percentile value of Δ, we show that our protocol is still as practical as drand.

Summary of contributions. To summarize, we make the following contributions in this work:

(1) We present a communication efficient BFT SMR protocol with quadratic communication per consensus decision (Section 4).

(2) We then present two random beacon protocols. Section 5.1 presents GRandPiper, a simple beacon protocol using PVSS with $O(\kappa n^2)$ communication. We then present BRandPiper, a protocol with better unpredictability in Section 5.3.

(3) We evaluate our BRandPiper protocol (Section 8).

(4) We present mechanisms for synchronizing a new node in Section 6 and reconfiguration in Section 7.

Related Work. Due to space constraints, we present related work to our SMR and random beacon protocol in Section 2.

Limitations. Our protocol depends of the synchrony assumption, i.e., messages sent between any two honest nodes in the system are always delivered within a public value Δ.

2 RELATED WORK

2.1 Related Works in the BFT SMR Literature

There has been a long line of work in improving the latency and communication complexity of consensus protocols [3, 6, 7, 15, 26, 33, 35, 46, 49]. The state-of-the-art BFT SMR protocols [3, 5, 7, 46] incur quadratic communication per consensus decision while using threshold signatures. Without threshold signatures, they incur cubic communication per consensus decision. Our BFT SMR protocol makes progress in the setting where threshold signatures are not desirable. Our protocol incurs $O(\kappa n^2)$ communication complexity under the q-SDH assumption or $O(\kappa n^2 \log n)$ without it at the expense of increased latency.

2.2 Related Works in the Random Beacons Literature

In this section, we explore random beacon protocols, sometimes also referred to as coin tossing protocols, in the synchronous setting. Some of the works were originally designed for the asynchronous settings, but in this section, we evaluate them in the synchronous setting.

Cachin et al. [17] use a threshold shared secret x_i of the secret $x \in \mathbb{Z}_q$, where q is the order of a group $\mathbb{G}$. To generate beacons, they create shares g^{cx_i} of the beacon g^{cx}, for some generator $g \in \mathbb{G}$. The beacon value c is some agreed upon coin value , say for instance, a counter. When $> t$ such shares are obtained, all the honest nodes obtain the same beacon value g^{cx}. Drand [25] uses a similar approach by replacing the threshold secret with a threshold BLS key and using signatures on the common coin c (say, a counter). This incurs a communication complexity of $O(\kappa n^2)$ always, but comes with the drawback that it does not support reconfiguration, i.e., if a single node is replaced or joins the system, the threshold shared keys (sharing x_i of x) must be regenerated or new keys (or shares x'_i) for the old key (or secret x) need to be reshared between the new participants.

Homomorphic Encryption Random Beacon (HERB) [20] uses homomorphic threshold ElGamal encryption scheme to generate

Table 1: Comparison of related works on Random Beacon protocols in standard synchrony

Protocol		Res.(t)	Unpred.	Comm. Compl		Adp. Adv.	Re-usable Setup	No DKG?	Assumption
				Best	Worst				
Cachin et al./Drand	[17, 25]	49%	1	$O(\kappa n^2)$	$O(\kappa n^2)$	✗	✗	✗	Threshold Secret/BLS
Dfinity	[4, 30]	49%	$O(\kappa)$	$O(\kappa n^2)$	$O(\kappa n^3)^*$	✗	✗	✗	Threshold BLS
HERB	[20]	33%	1	$O(\kappa n^3)$	$O(\kappa n^3)$	✗	✗	✗	Threshold ElGamal
HydRand	[43]	33%	$O(\min(\kappa, t))^\dagger$	$O(\kappa n^2)$	$O(\kappa n^3)$	✗	✓	✓	PVSS
HydRand (Worst)	[43]	33%	$t + 1$	$O(\kappa n^2)$	$O(\kappa n^3)$	✗	✓	✓	PVSS
RandChain	[29]	33%	$O(\kappa)$	$O(\kappa n^2)$	$O(\kappa n^3)$	✓	✗	✗	PoW
RandHerd	[47]	33%	$O(\kappa)$	$O(\kappa c \log n)^\P$	$O(\kappa n^4)$	✗	✗	✗	Threshold Schnorr
RandHound	[47]	33%	1	$O(\kappa c^2 n)^\P$	$O(\kappa c^2 n^2)^\P$	✗	✓	✓	Client based, PVSS
RandRunner	[42]	49%	$t + 1$	$O(\kappa n^2)$	$O(\kappa n^2)$	✓	✓	✓	VDF
RandShare	[47]	33%	1	$O(\kappa n^3)$	$O(\kappa n^4)$	✓	✓	✓	VSS
GRandPiper		49%	$O(\min(\kappa, t))^\dagger$	$O(\kappa n^2)$	$O(\kappa n^2)$	✗	✓	✓	PVSS, q-SDH
GRandPiper (Worst)		49%	$t + 1$	$O(\kappa n^2)$	$O(\kappa n^2)$	✗	✓	✓	PVSS, q-SDH
BRandPiper		49%	1	$O(\kappa n^2)^\S$	$O(\kappa n^3)$	✓	✓	✓	VSS, q-SDH

κ is the security parameter denoting maximum of sizes of signatures, hashes, and other components used in the protocol. **Res.** refers to the number of Byzantine faults tolerated in the system. **Unpred.** refers to the unpredictability of the random beacon, in terms of the number of future rounds an adaptive rushing adversary can predict. A rushing adversary can always obtain outputs before correct nodes, and hence, the minimum is 1. **Adp. Adv** refers to *Adaptive Adversary* whether the adversary can pick its t corruptions at any point in the protocol. **Reusable Setup** refers to a setup that can be reused when a node is replaced in the system. *probabilistically $O(\kappa n^3)$ when $O(n)$ consecutive leaders are bad. $^\P c$ is the average (constant) size of the groups of server nodes. †In expectation it is 2 rounds, the probability of an adversary predicting c epochs into the future is 2^{-c}, with a security parameter κ it is $\min(\kappa, t) + 1$ epochs. §In the optimistic case, when the leader is honest and $f = O(1)$ nodes are Byzantine.

random numbers. The system tolerates $n > 3t$ faults. Each node in the system encrypts a random share and posts it on the bulletin board. The protocol uses $t+1$ such encryptions to produce the final beacon output. The work requires the use of a Byzantine Agreement protocol whose inputs are $O(\kappa n)$ sized, and therefore trivially has a communication complexity of $O(\kappa n^3)$ in the best and worst cases. It also uses a variant of the threshold setup, thereby not permitting a re-usable setup.

RandChain [29] builds a DRB - Decentralized Random Beacon that assumes sequential Proof-of-Work (Seq-PoW), and VDFs along with Nakamoto consensus for consistency. Constructions using these assumptions are not energy-efficient. In PoW, an adversary with more hash power can neglect unfavorable random numbers by forking, and to avoid this requires the total honest hash power to be greater than $1/2$. The work uses existing Byzantine Agreement techniques which makes the protocol have a communication complexity of $O(\kappa n^2)$ in the best case, while inheriting the $O(\kappa n^3)$ communication complexity from BA [22] in the worst case.

Drake *et al.* [24] proposed a minimal bias-resistant VDF-based random beacon scheme, that assumes the existence of a VDF [13] and that the adversary has an advantage A_{max} in terms of speed over the honest nodes, in computing the VDF. The VDF is used to determine the beacon output for a round, and sufficiently old beacon outputs are used to select leaders for the Ethereum Proof-of-Stake protocol. The system tolerates $n > 3t$ faults, and is designed for partial synchrony.

RandRunner [42] builds a random beacon protocol using VDFs. Therefore, it has a setup that can be re-used. It uses trapdoor Verifiable Delay Functions - VDFs with strong uniqueness properties that produces unique values efficiently for the node that has the trapdoor, but takes time T to produce an output for the nodes that do not have the trapdoor. This allows the beacon to output bias-resistant outputs in every round. It is not immediately unpredictable as an adaptive adversary can corrupt the next $t < n/2$ leaders to know the outputs for the next t epochs. RandRunner uses reliable broadcasts which results in communication complexity of $O(\kappa n^2)$ for every round.

RandShare [47] is a strawman protocol for where the beacon output is generated using n independent Byzantine Agreement instances are run in a system tolerating $n > 3t$ faults. Running n BA instances incurs a communication complexity of $O(\kappa n^3)$ in the best case and $O(\kappa n^4)$ in the worst case. RandHerd [47] is an improved version of RandShare, driven by a client seeking a random beacon value. The client splits the system into groups of size c which internally use RandShare, leading to a communication complexity of $O(\kappa c^2 n)$, even in the worst case. However, even though c is a constant, it depends on n as the randomness of the beacon output is determined by c. RandHound [47] goes beyond RandHerd by using a stable-leader approach and dividing the system into groups of nodes with group leaders in a tree structure during the setup. This incurs RandHound, a communication complexity of $O(\kappa c^2 \log n)$ when the leader and the group leaders are honest. However, when the leader is bad, it uses a *view-change* protocol which is analogous to Byzantine Agreement, and incurs a cost of $O(\kappa n^3)$ communication complexity when $t < n/3$ consecutive leaders are Byzantine.

HydRand [43] is a random beacon protocol in the synchrony model which tolerates t faults, with $n > 3t$, with a communication complexity of $O(\kappa n^2)$, and $O(\kappa n^3)$ communication complexity in the worst case. It uses PVSS schemes (in particular SCRAPE [18]) and tolerates an adversary which can predict up to $t + 1$ epochs into the future.

Summary of limitations. Before, we set out to show what we achieve in this work, we show what the current works show what is achievable and what is not currently possible. The state of the art SMR protocols [3, 5, 7, 49] have a lower bound of $O(\kappa n^2)$ on the communication complexity in the best case, which hints that we cannot do better than $O(\kappa n^2)$ without improving SMR first. State of the art random beacon protocols [17, 20, 47] show that we cannot achieve an unpredictability advantage better than 1 epoch, since a rushing adversary can always know one epoch output before the rest of the honest nodes. State of the art Random beacon protocols [17, 20, 47] also show that a random beacon not in lock-step cannot avoid giving a time advantage of less than 2Δ to a rushing adversary. Our work aims to bridge these gaps in existing random beacon protocols.

Insights from existing works. RandPiper uses some insights from HydRand [43] and non-trivially improves upon them for optimal fault tolerance ($t < n/2$ unlike $t < n/3$ from HydRand) and better communication complexity (recall that HydRand has a communication complexity of $O(\kappa n^2)$ in the best case and $O(\kappa n^3)$ in the worst case). We first observe that HydRand secret shares one value and uses this shared value the next time the same node becomes a leader again. We observe that this is buffering of shares, and that this buffering can be done for more than one share, i.e., every time a node becomes a leader, we can use the value from the last d^{th} time it was a leader to buffer d shares. Next, we observe that HydRand cannot tolerate more than $t > n/3$ because it fails to deliver the PVSS encryptions to all correct nodes, if the leaders fails to send it to them. We solve this concern using extension techniques from recent works [36]. However, these works assume threshold signatures which we avoid in our protocol. Thus, in RandPiper, we achieve an optimal fault tolerance of $t < n/2$ and also improved communication complexity.

3 MODEL AND DEFINITIONS

We consider a system $\mathcal{P} := \{p_1, \ldots, p_n\}$ consisting of n nodes out of which at most $t = \lfloor n - 1/2 \rfloor$ nodes can be Byzantine which we term as a t-bounded adversary. The Byzantine nodes may behave arbitrarily. When we assume an adaptive adversary $\mathcal{A}$, the nodes can be corrupted to behave arbitrarily at any time during execution. When we assume a static adversary $\mathcal{A}$, the nodes to be corrupted must be chosen by the adversary before the start of the protocol execution. We also use the term t-bounded adversary. A node that is not faulty throughout the execution is considered to be honest and executes the protocol as specified.

We assume the network between nodes consists of point-to-point secure (authenticated and confidential) synchronous communication channels. Messages between nodes may take at most Δ time before they arrive, where Δ is a known maximum network delay. To provide safety under adversarial conditions, we assume that the adversary is capable of delaying the message for an arbitrary time upper bounded by Δ. In addition, we assume all honest nodes have clocks moving at the same speed. They also start executing the protocol within Δ time from each other. This can be easily achieved by using the clock synchronization protocol [3] once at the beginning of the protocol.

We make use of digital signatures and a public-key infrastructure (PKI) to prevent spoofing and replays and to validate messages. Message x sent by a node p is digitally signed by p's private key and is denoted by $\langle x \rangle_p$. In addition, we use $H(x)$ to denote the invocation of the random oracle H on input x.

3.1 Definitions

We consider a state machine replication protocol defined as follows:

Definition 3.1 (Byzantine Fault-tolerant State Machine Replication [44])**.** *A Byzantine fault-tolerant state machine replication protocol commits client requests as a linearizable log to provide a consistent view of the log akin to a single non-faulty server, providing the following two guarantees: (i) Safety. Honest nodes do not commit different values at the same log position. (ii) Liveness. Each client request is eventually committed by all honest nodes.*

We define d-absolute unpredictability as follows:

Definition 3.2 (d-absolute unpredictability)**.** *Consider an epoch based protocol. Let the fastest honest node be at epoch e. The protocol is said to be unpredictable with absolute bound d for $d \geq 1$, if the the probability of an adversary $\mathcal{A}$ predicting the honest output for any epoch $e' \geq e + d$ is* $\mathsf{negl}(\kappa)$.

We define the security requirements for a random beacon protocol $\mathcal{RB}$ as follows:

Definition 3.3 (Secure random beacon protocol)**.** *An epoch based protocol $\mathcal{RB}$ is said to be a d-secure random beacon protocol if it satisfies the following conditions:*

*(1) **Bias-resistance.** Let O be the output of the beacon for some epoch e. No adversary $\mathcal{A}$ can bias the output of the beacon, i.e., fix some c bits of O for any epoch $e > 1$ with probability better than* $\mathsf{negl}(c) + \mathsf{negl}(\kappa)$.

*(2) **Unpredictability.** The protocol is d-absolute unpredictable.*

*(3) **Guaranteed Output Delivery.** For every epoch $e \geq 1$, the protocol outputs a value.*

3.2 Primitives

In this section, we present several primitives used in our protocols.

Linear erasure and error correcting codes. We use standard $(t + 1, n)$ Reed-Solomon (RS) codes [40]. This code encodes $t + 1$ data symbols into codewords of n symbols and can decode the $t + 1$ elements of codewords to recover the original data.

- ENC. Given inputs $m_1, \ldots, m_{t+1}$, an encoding function ENC computes $(s_1, \ldots, s_n) = \mathsf{ENC}(m_1, \ldots, m_{t+1})$, where $(s_1, \ldots, s_n)$ are codewords of length n. A combination of any $t+1$ elements of n codewords uniquely determines the input message and the remaining of the codeword.
- DEC. The function DEC computes $(m_1, \ldots, m_{t+1}) = \mathsf{DEC}(s_1, ..., s_n)$, and is capable of tolerating up to c errors and d erasures in codewords $(s_1, \ldots, s_n)$, if and only if $t \geq 2c + d$.

Cryptographic accumulators. A cryptographic accumulator scheme constructs an accumulation value for a set of values and produces a witness for each value in the set. Given the accumulation value and a witness, any node can verify if a value is indeed in the set. Formally, given a parameter k, and a set D of n values $d_1, \ldots, d_n$, an accumulator has the following components:

- $\mathsf{Gen}(1^k, n)$: This algorithm takes a parameter k represented in unary form 1^k and an accumulation threshold n (an upper bound on the number of values that can be accumulated securely), returns an accumulator key a_k. The accumulator key a_k is part of the q-SDH setup and therefore is public to all nodes.
- $\mathsf{Eval}(a_k, \mathcal{D})$: This algorithm takes an accumulator key a_k and a set $\mathcal{D}$ of values to be accumulated, returns an accumulation value z for the value set $\mathcal{D}$.
- $\mathsf{CreateWit}(a_k, z, d_i, \mathcal{D})$: This algorithm takes an accumulator key a_k, an accumulation value z for $\mathcal{D}$ and a value d_i, returns $\perp$ if $d_i \in \mathcal{D}$,and a witness w_i if $d_i \in \mathcal{D}$.
- $\mathsf{Verify}(a_k, z, w_i, d_i)$: This algorithm takes an accumulator key a_k, an accumulation value z for $\mathcal{D}$, a witness w_i and a value d_i, returns true if w_i is the witness for $d_i \in \mathcal{D}$, and false otherwise.

In this paper, we use *collision free bilinear accumulators* from [37] as cryptographic accumulators.

Verifiable Secret Sharing and Commitments. We assume the existence of a secure Verifiable secret sharing scheme $\mathcal{VSS}$ with commitments, satisfying the security properties in Definition 3.4. We use the interfaces to a secure VSS scheme $\mathcal{VSS}$ as described in Table 2 (Appendix A).

Definition 3.4 (VSS Security [8]). *A VSS protocol consists of two phases: sharing and reconstruction. We call an $n-$node VSS protocol, with $t-$bounded adversary $\mathcal{A}$ and security parameter κ, an $(n-t)$-VSS protocol if it satisfies the following conditions:*

*(1) **Secrecy.** If the dealer L is honest, then the probability of $\mathcal{A}$ learning any information about the dealer's secret s in the sharing phase is* $\mathsf{negl}(\kappa)$.

*(2) **Correctness.** If L is honest, then the honest nodes output the secret s at the end of the reconstruction phase with a high probability of* $1 - \mathsf{negl}(\kappa)$.

*(3) **Commitment.** If L is Byzantine, then at the end of the sharing phase there exists a value $s^\star$ in the input space including $\perp$, such that at the end of the reconstruction phase all honest nodes output $s^\star$ with high probability* $1 - \mathsf{negl}(\kappa)$.

In our work, we implicitly assume that the VSS scheme used is $(n/2 + 1)$-secure.

Publicly Verifiable Secret Sharing — PVSS. PVSS schemes consist of communication such as broadcasts, posts on the bulletin board, as well as computational components such as share generation, encryption, etc. We separate the two components and present interfaces to computational algorithms that we will use in our protocols. We use the interfaces to a secure PVSS scheme $\mathcal{PVSS}$ as described in Table 3 (Appendix A).

We assume the existence of a secure PVSS algorithm $\mathcal{PVSS}$ as defined in Definition 3.5.

Definition 3.5 (PVSS security [8, 18]). *Let $L \in \mathcal{P}$ be the dealer with secret s and κ be the security parameter. A PVSS scheme $\mathcal{PVSS}$ is a secure VSS scheme (see Definition 3.4) and must provide the following guarantees:*

*(1) **Public Verifiability.** If the check in share verification algorithm ($\mathcal{PVSS}$.ShVrfy, see Table 3) returns* 1, *i.e., succeeds, then with high probability* $1 - \mathsf{negl}(\kappa)$, *the encryptions are valid shares of some secret.*

Normalizing the length of cryptographic building blocks. Let λ denote the security parameter, $\kappa_h = \kappa_h(\lambda)$ denote the hash size, $\kappa_a = \kappa_a(\lambda)$ denote the size of the accumulation value and witness of the accumulator and $\kappa_v = \kappa_v(\lambda)$ denote the size of secret share along with the associated proofs (both for PVSS and VSS). Further, let $\kappa = \max(\kappa_h, \kappa_a, \kappa_v)$; we assume $\kappa = \Theta(\kappa_h) = \Theta(\kappa_v) = \Theta(\kappa_a) = \Theta(\lambda)$. Throughout the paper, we will use the same parameter κ to denote the hash size, signature size, accumulator size and secret share size for convenience.

4 BFT SMR PROTOCOL

In this section, we present a simple BFT SMR protocol as a basic building block for the random beacon protocols discussed in following sections. Our SMR protocol achieves $O(\kappa n^2)$ bits communication complexity with a universal structured reference string (SRS) setup under the q-SDH assumption, or $O(\kappa n^2 \log n)$ bits communication complexity without the q-SDH setup assumption. In particular, we do not use threshold signatures, and thus avoid any distributed key generation during the setup or proactive secret sharing during reconfiguration. We note that prior synchronous BFT SMR protocols [5, 19, 46] with honest majority incur $O(\kappa n^3)$ communication per consensus decision without threshold signatures.

Epochs. Our protocol progresses through a series of numbered *epochs* with each epoch coordinated by a distinct leader. Epochs are numbered by integers starting with 1. The leaders for each epoch are rotated irrespective of the progress made in each epoch. For simplicity, we use round-robin leader election in this section and the leader of epoch e, represented as L_e, is determined by $e \bmod n$. Later in the beacon protocols, we introduce different leader election rules. Each epoch lasts for 11Δ time.

Blocks and block format. An epoch leader's proposal is represented as a *block*. Each block references its predecessor with the exception of the genesis block which has no predecessor. We call a block's position in the chain as its height. A block B_h at height h has the format, $B_h := (b_h, H(B_{h-1}))$ where b_h denotes the proposed payload at height h, B_{h-1} is the block at height $h-1$ and $H(B_{h-1})$ is the hash digest of B_{h-1}. The predecessor for the genesis block is $\perp$. A block B_h is said to be *valid* if (1) its predecessor block is valid, or if $h = 1$, predecessor is $\perp$, and (2) the payload in the block meets the application-level validity conditions. A block B_h *extends* a block B_l $(h \geq l)$ if B_l is an ancestor of B_h. Note that a block's height h and its epoch e need not necessarily be the same.

Certified blocks, and locked blocks. A block certificate on a block B_h consists of $t + 1$ distinct signatures in an epoch e and is represented by $C_e(B_h)$. Block certificates are ranked by epochs, i.e., blocks certified in a higher epoch has a higher rank. During the protocol execution, each node keeps track of all certified blocks and keeps updating the highest ranked certified block to its knowledge. Nodes will lock on highest ranked certified blocks and do not vote for blocks that do not extend highest ranked certified blocks to ensure safety of a commit.

Equivocation. Two or more messages of the same *type* but with different payload sent by an epoch leader are considered an equivocation. In this protocol, the leader of an epoch e sends propose and vote-cert messages (explained later) to all other nodes. In order to facilitate efficient equivocation checks, the leader sends the payload along with the signed hash of the payload. When an equivocation is detected, broadcasting the signed hash suffices to prove equivocation by L_e.

4.1 Protocol Details

We first describe a simple function that is used by an honest node to forward a long message received from the epoch leader.

Deliver function. The Deliver() function (refer Figure 2) implements efficient broadcast of long messages using erasure coding techniques and cryptographic accumulators. The input parameters to the function are message type mtype, long message b, accumulation value z_e corresponding to object b and epoch e in which the deliver function is invoked. The input message type mtype corresponds to message *type* containing large message b sent by leader L_e of epoch e. In order to facilitate efficient leader equivocation checks, the input message type mtype, hash of object b, accumulation value z_e and epoch e are signed by leader L_e.

When the function is invoked using the above input parameters, the message b is partitioned into $t + 1$ data symbols. The $t + 1$ data symbols are then encoded into n codewords $(s_1, \ldots, s_n)$ using ENC function (defined in Section 3). Then, the cryptographic witness w_i is computed for each codewords $(s_1, \ldots, s_n)$ using CreateWit (defined in Section 3). Then, the codeword and witness pair (s_j, w_j) is sent to the node $p_j \in \mathcal{P}$ along with the accumulation value z_e, message type mtype, and L_e's signature on the message.

When a node p_j receives the first valid codeword s_j for an accumulation value z_e such that the witness w_j verifies the codeword s_j (using Verify function defined in Section 3), it forwards the codeword and witness pair (s_j, w_j) to all nodes. Note that node p_j forwards only the first codeword and witness pair (s_j, w_j). Thus, it is required that all honest nodes forward the codeword and witness pair (s_j, w_j) for long message b; otherwise all honest nodes may not receive $t + 1$ codewords for b. When a node p_i receives $t + 1$ valid codewords 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.

The Deliver function contains two communication steps and hence requires 2Δ time to ensure all honest nodes can receive at least $t + 1$ codewords sufficient to reconstruct the original input b. Invoking Deliver on a long message of size ℓ incurs $O(n\ell+(\kappa+w)n^2)$ bits where κ is the size of accumulator and w is the size of the accumulator *witness*. The witness size is $O(\kappa)$ and $O(\kappa \log n)$ when bilinear accumulators and Merkle trees are respectively used as witnesses. Thus, the total communication complexity to broadcast a single message of size ℓ is $O(n\ell + \kappa n^2)$ bits, or $O(n\ell + \kappa n^2 \log n)$ bits without the q-SDH assumption.

BFT SMR Protocol. Our BFT SMR protocol is described in Figure 1. Consider an epoch e and its epoch leader L_e. To ensure an honest leader can always make progress, leader L_e first collects highest ranked certificate $C_{e'}(B_h)$ from all honest nodes. In each epoch, at a high level, there are two "rounds" of communication from the epoch leader. The first round involves leader making a proposal and the second round involves sending certificates to aid in committing the proposal.

Efficient propagation of proposal. In the first round, the leader proposes a block B_h to every node (step 2) by extending the highest ranked certificate $C_{e'}(B_h)$. The proposal for B_h, conceptually, has the form $\langle \text{propose}, B_h, C_{e'}(B_l), z_{pe}, e\rangle_{L_e}$ where z_{pe} is the accumulation value for the pair $(B_h, C_{e'}(B_l))$. In order to facilitate efficient equivocation checks, the leader signs the tuple $\langle \text{propose}, H(B_h, C_{e'}(B_l)), z_{pe}, e\rangle$ and sends B_h and $C_{e'}(B_l)$ separately. The size of this signed message is $O(\kappa)$ bits. In case of equivocation, all-to-all broadcast of this signed message incur only $O(\kappa n^2)$ in communication.

If the received proposal is valid and it extends the highest ranked certificate known to a node p_i, node p_i forwards the proposal. Forwarding the received proposal is required to ensure all honest nodes receive a common proposal; otherwise only a subset of the nodes may receive the proposal if the leader is Byzantine. Observe that the size of the proposal is linear as it contains certificate $C_{e'}(B_l)$ (which is linear in the absence of threshold signatures). A naïve approach of forwarding the entire proposal incurs $O(\kappa n^3)$ when all nodes broadcast their proposal. In order to save communication, nodes forward the proposal by invoking Deliver function. For linear sized proposal, invoking Deliver incurs $O(\kappa n^2)$ bits (or $O(\kappa n^2 \log n)$ bits under q-SDH assumption) in communication.

Observe that the Deliver primitive requires 2Δ time. In particular, we need to ensure all honest nodes forward their codeword and witness pair for the proposal. Thus, our protocol waits for 2Δ time (i.e., vote-timer$_e$) before voting to check for equivocation. Hence, if no equivocation is detected at the end of 2Δ wait, all honest nodes forwarded their codeword and witness pair for the proposal and all honest nodes can reconstruct the proposal. At the end of 2Δ wait, if there no equivocation is detected, nodes vote for the proposed block B_h (step 3).

Ensuring the receipt of a certificate efficiently. Observe that a vote message is $O(\kappa)$ sized and hence, it can be broadcast using all-to-all communication with communication complexity of $O(\kappa n^2)$. However, if every node that commits needs to ensure that all honest parties receive a certificate for the block being committed, this can result in $O(\kappa n^3)$ complexity again. This is because, all-to-all broadcast of linear sized certificate incurs $O(\kappa n^3)$. One might try to invoke Deliver to propagate the certificate. However, this does not save communication. This is because, in general, there can be exponentially many combinations of $t + 1$ signatures forming a certificate depending on the set of signers, and each node may invoke Deliver on a different combination.

Let e be the current epoch and L_e be the leader of epoch e. For each epoch e, node p_i performs following operations:

(1) **Epoch advancement.** When epoch-timer$_{e-1}$ reaches 0, enter epoch e. Upon entering epoch e, send highest ranked certificate $C_{e'}(B_l)$ to L_e. Set epoch-timer$_e$ to 11Δ and start counting down.

(2) **Propose.** L_e waits for 2Δ time after entering epoch e and broadcasts $\langle \text{propose}, B_h, C_{e'}(B_l), z_{pe}, e\rangle_{L_e}$ where B_h extends B_l. $C_{e'}(B_l)$ is the highest ranked certificate known to L_e.

(3) **Vote.** If epoch-timer$_e \geq 7\Delta$ and node p_i receives the first proposal $p_e = \langle \text{propose}, B_h, C_{e'}(B_l), z_{pe}, e\rangle_{L_e}$ where B_h extends a highest ranked certificate, invoke Deliver(propose, p_e, z_{pe}, e). Set vote-timer$_e$ to 2Δ and start counting down. When vote-timer$_e$ reaches 0, send $\langle \text{vote}, H(B_h), e\rangle_i$ to L_e.

(4) **Vote cert.** Upon receiving $t+1$ votes for B_h, L_e broadcast $\langle \text{vote-cert}, C_e(B_h), z_{ve}, e\rangle_{L_e}$.

(5) **Commit.** If epoch-timer$_e \geq 3\Delta$ and node p_i receives the first $v_e = \langle \text{vote-cert}, C_e(B_h), z_{ve}, e\rangle_{L_e}$, invoke Deliver(vote-cert, v_e, z_{ve}, e). Set commit-timer$_e$ to 2Δ and start counting down. When commit-timer$_e$ reaches 0, if no equivocation for epoch-e has been detected, commit B_h and all its ancestors.

(6) **(Non-blocking) Equivocation.** Broadcast equivocating hashes signed by L_e and stop performing epoch e operations.

Figure 1: BFT SMR Protocol with $O(\kappa n^2)$ bits communication per epoch and optimal resilience

Deliver(mtype, b, z_e, e):

(1) Partition input b into $t+1$ data symbols. Encode the $t+1$ data symbols into n codewords $(s_1, \ldots, s_n)$ using ENC function. Compute witness $w_j\ \forall s_j \in (s_1, \ldots, s_n)$ using CreateWit function. Send $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_r, r\rangle_r$ to node $j\ \forall j \in [n]$.

(2) If node p_j receives the first valid codeword $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_e, e\rangle_*$ for the accumulator z_e, forward the codeword to all the nodes.

(3) Upon receiving $t+1$ valid codewords for the accumulator z_e, decode b using DEC function.

Figure 2: Deliver function

This issue can be addressed if we ensure that there is a single certificate for a block. Hence, we use the leader to collect signatures and form a single certificate (step 3). The leader forwards this certificate via $\langle \text{vote-cert}, C_e(B_h), z_{ve}, e\rangle_{L_e}$ to all nodes (step 4) where z_{ve} is the accumulation value of $C_e(B_h)$. 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.

To ensure that every honest node receives this certificate, we again resort to the Deliver primitive which yields a communication complexity of $O(\kappa n^2)$ when all honest parties are invoking it using the same certificate. Again, to tolerate malicious behaviors such as sending multiple different certificates for the same block (due to which none of them may be delivered), we treat the vote-cert message similar to the proposal and perform equivocation checks. Thus, nodes commit only if they observe no equivocation 2Δ time after they invoke Deliver (step 5).

Epoch timers. Observe that we set the epoch timer epoch-timer$_e$ for each epoch e to be 11Δ. This is the maximum time required for an epoch when the leader is honest and all messages take Δ time. Similarly, in different steps, we make appropriate checks w.r.t. epoch-timer$_e$ to ensure that the protocol is making sufficient progress within the epoch.

Latency. We note that all honest nodes commit in the same epoch when the epoch leader is honest. However, when the epoch leader is Byzantine, only some honest nodes may commit in that epoch. Due to the round-robin leader selection, there will be at least one honest leader every $t+1$ epochs and all honest nodes commit common blocks up to the honest epoch. Thus, our protocol has a worst-case commit latency of $t+1$ epochs.

4.2 Safety and Liveness for BFT SMR

We say a block B_h is committed directly in epoch e if it is committed as a result of its own commit-timer$_e$ expiring. We say a block B_h is committed indirectly if it is a result of directly committing a proposal B_ℓ ($\ell > h$) that extends B_h.

Fact 4.1. *If an honest node delivers an object b at time τ in epoch e and no honest node has detected an epoch e equivocation by time $\tau + \Delta$, then all honest nodes will receive object b by time $\tau + 2\Delta$ in epoch e.*

Proof. Suppose an honest node p_i delivers an object b at time τ in epoch e. Node p_i must have sent valid codewords and witness $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_e, e\rangle_i$ computed from object b to every $p_j \in \mathcal{P}$ at time τ. The codewords arrive at all honest nodes by time $\tau + \Delta$.

Since no honest node has detected an epoch e equivocation by time $\tau + \Delta$, it must be that either honest nodes will forward their codeword $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_e, e\rangle$ when they receive the codewords sent by node p_i or they already sent the corresponding codeword when they either delivered object b or received the codeword from some other node p_j. In any case, all honest nodes will forward their epoch e codeword corresponding to object b by time $\tau + \Delta$. Thus, all honest nodes will have received $t+1$ valid codewords for a common accumulation value z_e by time $\tau + 2\Delta$ sufficient to decode object b by time $\tau + 2\Delta$. □

Fact 4.2. *If an honest node votes for a block B_h at time τ in epoch e, then all honest nodes receive B_h by time τ.*

Proof. Suppose an honest node p_i votes for a block B_h at time τ in epoch e. Node p_i must have received proposal p_e for B_h by time $\tau - 2\Delta$ and detected no epoch e equivocation by time τ. This implies no honest node detected an epoch e equivocation by time $\tau - \Delta$. Node p_i must have invoked Deliver(propose, p_e, z_{pe}, e) at time $\tau - 2\Delta$. By Fact 4.1, all honest nodes receive p_e by time τ. Thus, all honest nodes must have received B_h by time τ. □

Lemma 4.3. *If an honest node directly commits a block B_h in epoch e, then (i) no equivocating block certificate exists in epoch e, and (ii) all honest nodes receive $C_e(B_h)$ before quitting epoch e.*

Proof. Suppose an honest node p_i commits a block B_h in epoch e at time τ. Node p_i must have received a vote-cert for B_h at time $\tau - 2\Delta$ such that its epoch-timer$_e \geq 3\Delta$ and did not detect an equivocation by time τ. This implies no honest node detected an epoch e equivocation by time $\tau - \Delta$. In addition, some honest node p_j must have voted for B_h by time $\tau - 2\Delta$. By Fact 4.2, all honest nodes would receive B_h by time $\tau - 2\Delta$.

For part (i), observe that no honest node received an equivocating proposal by time $\tau - 2\Delta$; otherwise, all honest nodes would have received a codeword for equivocating proposal by time $\tau - \Delta$ and node p_i would not commit. And, no honest node would vote for an equivocating block after time $\tau - 2\Delta$ (since they have received B_h by time $\tau - 2\Delta$). Thus, an equivocating block certificate does not exist in epoch e.

For part (ii), observe that node p_i must have invoked Deliver(vote-cert, v_e, z_{ve}, e) for $v_e = C_e(B_h)$ at time $\tau - 2\Delta$ and did not detect epoch e equivocation by time τ. By Fact 4.1, all honest nodes receive v_e by time τ. Note that node p_i must have its epoch-timer$_e \geq 3\Delta$ at time $\tau - 2\Delta$. Since, all honest nodes are synchronized within Δ time, all other honest nodes must have epoch-timer$_e \geq 2\Delta$ at time $\tau - 2\Delta$. Thus, all nodes are still in epoch e at time τ and receive $C_e(B_h)$ before quitting epoch e. □

Lemma 4.4 (Unique Extensibility)**.** *If an honest node directly commits a block B_h in epoch e, then any certified blocks that ranks higher than $C_e(B_h)$ must extend B_h.*

Proof. The proof is by induction on epochs $e' > e$. For an epoch e', we prove that if a $C_{e'}(B_{h'})$ exists then it must extend B_h.

For the base case, where $e' = e + 1$, the proof that $C_{e'}(B_{h'})$ extends B_h follows from Lemma 4.3. The only way $C_{e'}(B_{h'})$ for $B_{h'}$ forms is if some honest node votes for $B_{h'}$. However, by Lemma 4.3, there does not exist any equivocating block certificate in epoch e and all honest nodes receive and lock on $C_e(B_h)$ before quitting epoch e. Thus, a block certificate cannot form for a block that does not extend B_h.

Given that the statement is true for all epochs below e', the proof that $C_{e'}(B_{h'})$ extends B_h follows from the induction hypothesis because the only way such a block certificate forms is if some honest node votes for it. An honest node votes in epoch e' only if $B_{h'}$ extends a valid certificate $C_{e''}(B_{h''})$. Due to Lemma 4.3 and the induction hypothesis on all block certificates of epoch $e < e'' < e'$, $C_{e'}(B_{h'})$ must extend B_h. □

Theorem 4.5 (Safety). *Honest nodes do not commit conflicting blocks for any epoch e.*

Proof. Suppose for the sake of contradiction two distinct blocks B_h and B'_h are committed in epoch e. Suppose B_h is committed as a result of $B_{h'}$ being directly committed in epoch e' and B'_h is committed as a result of $B'_{h''}$ being directly committed in epoch e''. Without loss of generality, assume $h' < h''$. Note that all directly committed blocks are certified. By Lemma 4.4, $B'_{h''}$ extends $B_{h'}$. Therefore, $B_h = B'_h$. □

Fact 4.6. *Let B_h be a block proposed in epoch e. If the leader of an epoch e is honest, then all honest nodes commit B_h and all its ancestors in epoch e.*

Proof. Suppose leader L_e of an epoch e is honest. Let τ be the earliest time when an honest node p_i enters epoch e. Due to Δ delay between honest nodes, all honest nodes enter epoch e by time $\tau + \Delta$. Some honest nodes might have received a higher ranked certificate than leader L_e before entering epoch e; thus, they send their highest ranked certificate to leader L_e.

Leader L_e might have entered epoch e at time τ while some honest nodes enter epoch e only at time $\tau + \Delta$. The 2Δ wait in the Propose step ensures that the leader can receive highest ranked certificates from all honest nodes. However, leader L_e may enter epoch e Δ time after the earliest honest nodes. Due to 2Δ wait after entering epoch e, leader L_e collects the highest ranked certificate $C_{e'}(B_l)$ by time $\tau + 3\Delta$ and sends a valid proposal $p_e = \langle \text{propose}, B_h, e, C_{e'}(B_l), z_{pe} \rangle_{L_e}$ for a block B_h that extends $C_{e'}(B_l)$ which arrives all honest nodes by time $\tau + 4\Delta$.

Thus, all honest nodes satisfy the constraint epoch-timer$_e \geq 7\Delta$. In addition, B_h extends the highest ranked certificate. So, all honest nodes will invoke Deliver(propose, p_e, z_{pe}, e) and set vote-timer$_e$ to 2Δ which expires by time $\tau + 6\Delta$. All honest nodes send vote for B_h to L_e which arrives L_e by time $\tau + 7\Delta$. Leader L_e forwards $C_e(B_h)$ which arrives all honest nodes by time $\tau + 8\Delta$. Note that all honest nodes satisfy the constraint epoch-timer$_e \geq 3\Delta$ and honest nodes set their commit-timer$_e$ to 2Δ which expires by time $\tau + 10\Delta$. Moreover, no equivocation exists in epoch e. Thus, all honest nodes will commit B_h and its ancestors in epoch e before their epoch-timer$_e$ expires. □

Theorem 4.7 (Liveness). *All honest nodes keep committing new blocks.*

Proof. For any epoch e, if the leader L_e is Byzantine, it may not propose any blocks or propose equivocating blocks. Whenever an honest leader is elected in epoch e, by Fact 4.6, all honest nodes commit in epoch e. Since we assume a round-robin leader rotation policy, there will be an honest leader every $t + 1$ epochs, and thus the protocol has a commit latency of $t + 1$ epochs.

□

Lemma 4.8 (Communication complexity)**.** *Let ℓ be the size of block B_h, κ be the size of accumulator and w be the size of witness. The communication complexity of the protocol is $O(n\ell + (\kappa + w)n^2)$ bits per epoch.*

Proof. At the start of an epoch e, each node sends a highest ranked certificate to leader L_e. Since, size of each certificate is $O(\kappa n)$, this step incurs $O(\kappa n^2)$ bits communication. A proposal consists of a block of size ℓ and block certificate of size $O(\kappa n)$. Proposing $O(n + \ell)$-sized object to n nodes incurs $O(\kappa n^2 + n\ell)$. Delivering $O(\kappa n + \ell)$-sized object has a cost $O(n\ell + (\kappa + w)n^2)$, since each node broadcasts a codeword of size $O((n + \ell)/n)$, a witness of size w and an accumulator of size κ.

In Vote cert step, the leader broadcasts a certificate for block B_h which incurs $O(\kappa n^2)$ communication. Delivering $O(\kappa n)$-sized $C_e(B_h)$ incurs $O((\kappa + w)n^2)$ bits. Hence, the total cost is $O(n\ell + (\kappa + w)n^2)$ bits.

□

5 RANDOM BEACON PROTOCOLS

In this section, we present two random beacon protocols while tolerating $f \leq t < n/2$ Byzantine faults. The first protocol GRandPiper outputs a random beacon with $O(\kappa n^2)$ communication complexity *always*, per beacon output, where κ is the security parameter, guarantees bias-resistance, and $O(\min(\kappa, t))$-absolute unpredictability against a static adversary, but $t+1$-absolute unpredictability against an adaptive adversary. The second protocol BRandPiper outputs a random beacon with $O(\kappa f n^2)$ communication complexity per output after amortization where κ is the security parameter, and guarantees bias-resistance and 1-absolute unpredictability. When the actual number of faults $f = O(1)$, the communication complexity is quadratic.

A key aspect of both of our protocols is their reconfiguration-friendliness. A protocol is said to be reconfiguration-friendly if it allows changing protocol parameters such as the scheme and nodes, without stopping the old instance, and starting a new one. Such reconfiguration is possible if the setup used for the protocol does not bind to the system, as such a binding will force a new setup to change any parameter in the system. This is true, for instance, when using threshold signatures in a protocol which is used by many existing permissioned systems [9, 15, 16, 27]. Neither of our protocols use setups for threshold signatures, but a setup based on the q-SDH assumption. This allows for easy reconfiguration.[2]

5.1 RandPiper – GRandPiper Protocol

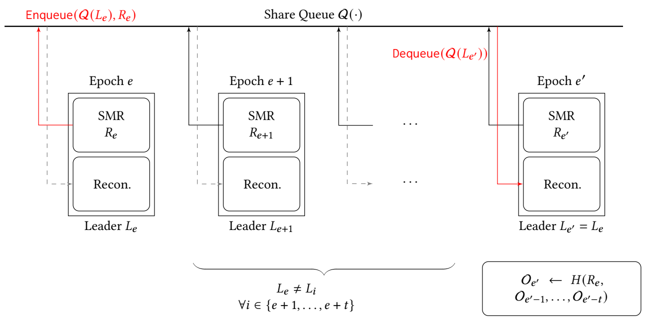


Figure 3: Overview of RandPiper – GRandPiper. In every epoch, a PVSS sharing of some random value is secret shared. At the same time, a reconstruction protocol is used to reconstruct the random value committed by the leader of this epoch, the last time it was a leader. $O_{e'}$ is generated using the random value R_e, shared in epoch e, reconstructed in epoch $e' > e + t$, and outputs $\{O_{e'-1}, \ldots, O_{e'-t}\}$ from previous epochs by using them as inputs to the random oracle H.

We use the SMR protocol (refer Figure 1) described in Section 4 as a building block. At a high-level, consider using the SMR protocol such that the leader outputs a number chosen uniformly at random in each epoch. The random beacon output can be a function of the outputs of the last $t+1$ epochs, allowing for the presence of at least one honest input (chosen uniformly at random) which is potentially sufficient to obtain a *random* output. This argument holds *only if* each leader chooses their input in the SMR protocol independently of other inputs. Otherwise, if a Byzantine leader can choose an input after knowing the outputs of the previous t instances then it can bias the output. A separate concern with using the SMR protocol as is, is that in an epoch with a Byzantine leader, honest nodes may not all output the same value or output at all.

To fix both of these concerns, we require each node to send a commitment of a random value more than t epochs before it will be reconstructed and used in the beacon protocol. To ensure the secrecy of this value (for unpredictability and bias-resistance), the values are shared with the nodes using a publicly verifiable secret sharing (PVSS) scheme (refer Appendix A for detailed PVSS interface). Committing a secretly chosen value ahead of time helps us solve both of our previous concerns. First, if the same leader is not chosen twice in any span of $t+1$ epochs, it ensures that the $t+1$ values that will be used to construct the beacon protocol are chosen independently of one another. Thus, when nodes reconstruct a value in an epoch, it corresponds to a value committed more than t epochs before. Moreover, the nodes can reconstruct this value independent of the participation of the leader in this epoch. Second, waiting for $t+1$ epochs before opening allows for the value to be committed by the SMR protocol. Thus, all honest nodes will open the same value in an epoch.

A graphical description of this approach is presented in Figure 3. In epoch e, a leader L_e inputs PVSS shares corresponding to a random value R_e to the SMR protocol. Conceptually, when the block is committed, this value is added to a queue $Q(L_e)$ corresponding to this leader. When the same node is chosen the next time as a leader, say in epoch e', the committed shares of R_e is dequeued and reconstructed by all honest nodes to obtain R_e. The output $O_{e'}$ of epoch e' can be computed as $H(R_e, O_{e'-1}, \ldots, O_{e'-t})$. To allow for unpredictability in leader selection while disallowing repetition within $t+1$ epochs, the leader for the next epoch $e'+1$ is chosen based on $O_{e'}$ and by removing the leaders $L_{e'}, \ldots, L_{e'-t}$.

A remaining concern is when no values are added to the chain at epoch e. Observe that the reconstruction in epoch e is not affected, since nodes reconstruct values previously committed. However, nodes may not have shares in epoch $e' > e + t$ where e' is the first epoch where L_e is chosen as the leader again. To fix this concern, we ensure that such a malicious leader who does not commit in epoch e can be removed by all nodes by $e + t < e'$. Subsequently, we can ensure that L_e is never chosen as the leader again. To allow for reconstruction the first time a node is chosen as the leader, we ensure a setup where each node has an agreed upon share buffered for every other node.

5.1.1 *Protocol details.*

We now explain the protocol in detail (described in Figure 4). We use a Publicly Verifiable Secret Sharing (PVSS) scheme $\mathcal{PVSS}$ with threshold t to generate encrypted shares and an associated proof that guarantees that any $> t$ nodes will reconstruct a unique secret.

Setup. We establish PVSS parameters $\mathcal{PVSS}$.pp, and public keys $\mathcal{PVSS}.\text{pk}_i$ for every node $p_i \in \mathcal{P}$. We also buffer shares for one random value for every node p_i, i.e., fill $Q(p_i)$ for $p_i \in \mathcal{P}$. We start with epoch $e = 1$, and use seed random values for R_e and

[2]We can use Merkle trees instead of q-SDH at the expense of $O(\log n)$ multiplicative communication complexity.

All nodes $p_i \in \mathcal{P}$ running the SMR protocol do the following:

- **Setup.** Set $e = 1$. All nodes agree upon seed random values for R_e and $\{O_{e-1}, \ldots, O_{e-t}\}$. Set $\mathcal{L}_{Last} \leftarrow \{p_n, \ldots, p_{n-t}\}$, $\mathcal{P}_r \leftarrow \emptyset$. Run $\mathcal{PVSS}$.Setup and agree on the public parameters $\mathcal{PVSS}$.pp. Then every node generates a key pair $(\mathcal{PVSS}.\mathsf{sk}, \mathcal{PVSS}.\mathsf{pk}) \leftarrow \mathcal{PVSS}.\mathsf{KGen}(\kappa)$, and all nodes agree on each others public keys.
- **Leaders.** Choose leaders for an epoch e using Definition 5.1 instead of a round-robin order.
- **Blocks.** The leader L_e of an epoch e, creates a PVSS sharing $(\mathcal{PVSS}.\vec{S}, \mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi) \leftarrow \mathcal{PVSS}.\mathsf{ShGen}(R)$ of a random value chosen from the input space of $\mathcal{PVSS}$, and creates a block B_h with block contents b_h as $b_h := (\mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi) \leftarrow \mathcal{PVSS}.\mathsf{ShGen}(R)$. (We drop the individual shares in $\mathcal{PVSS}.\vec{S}$.)
- **Update.** When committing a block B_h sent by leader $L_{e'}$ for some epoch e', $\mathsf{Enqueue}(Q(L_{e'}), b_h)$. At the end of epoch e, if no block was committed for epoch $e - t$ by L_{e-t}, then remove L_{e-t} from future proposals, i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{e-t}\}$ from epoch $e + 1$.
- **Reconstruct.** When the epoch timer epoch-timer$_{e-1}$ for epoch $e - 1$ ends, obtain the $(\mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi)$ corresponding to the committed block in $\mathtt{Dequeue}(Q(L_e))$. Send $s \leftarrow \mathcal{PVSS}.\mathsf{Dec}(\mathcal{PVSS}.\mathsf{sk}, \mathcal{PVSS}.\vec{E}_i)$ to all the nodes in the system. On receiving share s' from another node p_j, ensure that $\mathcal{PVSS}.\mathsf{Enc}(\mathcal{PVSS}.\mathsf{pk}_j, s') = \mathcal{PVSS}.\vec{E}_j$. On receiving $t + 1$ valid shares in $\mathcal{PVSS}.\vec{S}$, reconstruct $R_e \leftarrow \mathcal{PVSS}.\mathsf{Recon}(\mathcal{PVSS}.\vec{S})$.
- **Output.** After reconstructing R_e for epoch e, output the beacon value O_e by computing $O_e \leftarrow H(R_e, O_{e-1}, \ldots, O_{e-t})$

Figure 4: RandPiper – GRandPiper beacon protocol description.

$\{O_{e-1}, \ldots, O_{e-t}\}$. We also arbitrarily assign $\mathcal{L}_{Last} \leftarrow \{p_n, \ldots, p_{n-t}\}$ and set $\mathcal{P}_r \leftarrow \emptyset$.

Leader selection. The leader for epoch e is chosen based on the following rule:

Definition 5.1 (Leader selection rule)**.** *Let e be the current epoch, $\mathcal{L}_{Last} := \{L_{e-1}, \ldots, L_{e-t}\}$ be the leaders of the last t epochs, $\mathcal{P}_r$ be the set of nodes that are removed (due to misbehavior), and $\mathcal{L}_e = (\mathcal{P} \setminus \mathcal{L}_{Last}) \setminus \mathcal{P}_r := \{l_0, \ldots, l_{w-1}\}$, be a set of candidate leaders for epoch e ordered canonically, with $0 < w < n - t$ and $\mathcal{L}_e \subseteq \mathcal{P}$. Then the leader L_e of epoch e, is derived from output O_{e-1}, as*

$$L_e \leftarrow l_{(O_{e-1} \mod w)}$$

Blocks. The leader of an epoch chooses R uniformly at random from the input space of the $\mathcal{PVSS}$ algorithm (which could be a cyclic additive/multiplicative group, or pairing groups). The leader uses the $\mathcal{PVSS}$.ShGen algorithm to generate share $\mathcal{PVSS}.\mathsf{s}_i$ for node p_i which are encrypted using $\mathcal{PVSS}.\mathsf{pk}_i$, and all shares for the nodes are stored in $\mathcal{PVSS}.\vec{E}$. The $\mathcal{PVSS}$.ShGen algorithm also outputs the proof $\mathcal{PVSS}.\pi$ that any $> t$ shares will reconstruct a unique secret, which implies that the degree of the polynomial cannot be more than t. Finally, the block in our SMR protocol consists of the outputs of the $\mathcal{PVSS}$.ShGen algorithm, i.e., $b_h := (\mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi) \leftarrow \mathcal{PVSS}.\mathsf{ShGen}(R_e)$. An honest nodes acknowledges B_h if b_h meets the validity condition $\mathcal{PVSS}$.ShVrfy algorithm. Note that despite the blocks being $O(\kappa n)$ sized, due to our usage of Deliver primitive, we retain a communication complexity of $O(\kappa n^2)$ per epoch.

Commit, reconstruct, and output beacon value. In each epoch, nodes commit the shares sent by the leader. They also reconstruct the last sharing sent by the leader at the start of the epoch. Note that each node can separately maintain the last time a node was elected as the leader, and thus, be able to appropriately invoke $\mathtt{Dequeue}(Q(L_e))$. Moreover, since a leader does not repeat in any consecutive $t + 1$ epochs, and we ensure that the set of leaders are consistently known to all honest nodes (as will be shown in the next subsection), the value being reconstructed is agreed upon by all the honest nodes. When the nodes reconstruct R_e, they already have access to $\{O_{e-1}, \ldots, O_1\}$. Hence, they can compute a consistent output O_e. Observe that since all nodes enter epoch e within a delay of Δ, they also output the beacon value within Δ time of each other.

Remove misbehaving leaders. Finally, at the end of an epoch e, if no block was committed in epoch $e - t$ by L_{e-t}, L_{e-t} is removed from all future proposals. Since this operation is performed after $t + 1$ epochs, all nodes will perform this action consistently.

5.2 GRandPiper Security Analysis

Theorem 5.1 (Consistent beacon). *Let $L = L_e$ denote the leader of epoch e. Then the following properties hold:*

(i) Block consistency*: if an honest node commits a block B proposed in epoch $e' \leq e - t$, then all the honest nodes commit block B by epoch e.*

(ii) Leader validity*: all the honest nodes have a block in $Q(L_e)$.*

(iii) Output consistency*: all the honest nodes output the same randomness R_e, output O_e, and*

(iv) Leader consistency*: all the honest nodes choose the same leader for epoch $e + 1$.*

Proof. We prove the theorem by induction on epochs.

Base case for epoch $e = 1$ to $e = t + 1$. (ii) should hold for the first $t + 1$ leaders because we fill $Q(p_i)$ with $m = 1$ values for all nodes $p_i \in \mathcal{P}$ during the setup phase. Additionally, from Definition 5.1, no leaders repeat in the first $t + 1$ epochs, thereby proving (ii) for the base case. (iii) and (iv) hold since the first $t + 1$ outputs only use the seed values, and pre-agreed upon shares from Q from the setup phase. At epoch $e = t + 1$, from the proof for (iv) for the base cases, we know that all nodes agree on the leaders for epochs $1 \leq e' \leq t + 1$. Therefore, if some honest node commits block B_1 from epoch $e = 1$, then all honest nodes will commit B_1 by epoch $e = t + 1$, because at least one leader in some epoch $1 \leq e' \leq t + 1$ must be honest, and from Fact 4.6 all honest nodes commit the block proposed in e' and therefore directly or indirectly commit B_1. Therefore, by epoch $e = t + 1$ all honest nodes commit B_1, thereby proving (i) for the base cases.

Induction hypothesis. The statements hold until epoch $e - 1$.

Induction step. *Proof for (i).* From the induction hypothesis for (iv), we have that all the leaders until epoch e are consistent and at epoch $e - 1$, and from the induction hypothesis for (i) all honest nodes would have committed all the blocks for epoch $e' < e - 1 - t$

by consistent leaders up to epoch $e-1$. Now, at epoch e all honest nodes need to decide on the block B proposed in epoch $e-t-1$. In epochs $e-t-1 \leq e' < e$, there is one epoch e' whose leader $L_{e'}$ is honest, from Fact 4.6, all honest nodes commit B in epoch e', thus proving the hypothesis for (i).
Proof for (ii). By the induction hypothesis for (iv), the leader of epoch e and all previous epochs is agreed upon. Let L_e be the leader for epoch e. Then L_e must have committed a block in some epoch $< e-t$, or during the setup. If L_e was never the leader, then the hypothesis (ii) is trivially satisfied. Let $e' < e-t$ be the latest epoch in which L_e was the leader last. If L_e proposed a block in some epoch $e' < e-f$, then from the proof for (i) for epoch e, all nodes agree on the same block for epoch e'. If no block proposed in epoch e' is added to the chain by epoch $e'+t < e$, then from our leader selection rule (in Definition 5.1), no honest node will derive L_e as the leader as $L_e \in \mathcal{P}_r$. Therefore, (ii) also holds for epoch e.
Proof for (iii). The randomness R_e depends on choosing a committed value to be reconstructed. The output O_e depends on R_e and $\{O_{e-1}, \ldots, O_{e-t}\}$. By the induction hypothesis for (iii), all honest nodes agree on $O_{e-1}, \ldots, O_{e-f}$. Moreover, by the induction hypothesis for (iv), they also choose the same leader L_e for epoch e. Thus, if we can prove that all honest nodes agree upon the value from L_e that is reconstructed in epoch e, then agreement on O_e is trivial. From the proof of (i) and (ii), we know that there is a block b_h that is enqueued in the queue $Q(L_e)$ for L_e, which all honest nodes agree on, and therefore obtain the same R_e for epoch e. Thus, (iii) holds true for epoch e.
Proof for (iv). The leader derivation depends on the candidate set $\mathcal{L}_e$, outputs of the last t iterations $\{O_{e-1}, \ldots, O_{e-t}\}$, and the reconstructed randomness R_e in epoch e. By the induction hypothesis for (iv), and proof of (iii), the output of the last f iterations and that of epoch e is agreed upon. From the proof of (i), (ii), and the induction hypothesis for (iv), all honest nodes have the same $\mathcal{L}_{Last}$ and $\mathcal{P}_r$. From (iii), all honest nodes derive the same leader for epoch $e+1$. □

Next, we show that GRandPiper protocol is a secure PVSS protocol, satisfying the requirements specified in Definition 3.5.

Theorem 5.2 (Secure PVSS). *Assuming a secure PVSS scheme $\mathcal{PVSS}$, the GRandPiper protocol is a secure publicly verifiable secret sharing protocol with the dealer as the leader of an epoch, and the rest of the nodes as the verifiers.*

Proof Sketch. We already know that our SMR is secure against a t-bounded adversary assuming a secure digital signature scheme, q-SDH and a random oracle H. Given a secure suite of algorithms in $\mathcal{PVSS}$, on a high level we do not reveal any new information.

To formally prove it, consider the view V_i of any honest node p_i. It observes $V_i := (\mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\mathsf{pk}i, \mathcal{PVSS}.\pi)$. In the underlying PVSS scheme $\mathcal{PVSS}$, the view of a node is also V_i. An adversary $\mathcal{A}$ that can successfully violate the secrecy property from GRandPiper can do so by:

(1) Breaking the underlying PVSS scheme $\mathcal{PVSS}$. Since $\mathcal{PVSS}$ satisfies Definition 3.5, this can occur with $\mathsf{negl}(\kappa)$ probability.

(2) Guessing the secret. The probability of an adversary winning this way is $\mathsf{negl}(\kappa)$.

Correctness. Let L_e be an honest leader for epoch e. Then its proposed block that shares R_e is always committed (from Fact 4.6). Thus when the secret is reconstructed in the beacon protocol (Figure 4) all the honest nodes will output R_e with a high probability of $1 - \mathsf{negl}(\kappa)$ (from the underlying $\mathcal{PVSS}$ algorithm).
Commitment. If L_e is Byzantine, then either all the honest nodes commit to one of the blocks B_h proposed, or $\perp$ by epoch $e+t+1$. Therefore, the *commitment* property is satisfied by our protocol. From the underlying scheme $\mathcal{PVSS}$, there is a negligible probability $\mathsf{negl}(\kappa)$ for two correct nodes p_i and p_j to output different $s_i^* \neq s_j^* \neq \perp$.
(Public) Verifiability. This property holds true with high probability from the underlying PVSS scheme $\mathcal{PVSS}$ as the views are identical. The probability is over the choice of randomness for the verifier. □

Concrete instantiations. Consider instantiating GRandPiper using SCRAPE [18]. We can show a reduction from an adversary breaking the IND1-secrecy [31] property in GRandPiper into an adversary that can break the secrecy property from SCRAPE (which in turn shows a reduction to DDH or DBS assumptions [18, Sec.3, Sec.4]). In the simulation, since the adversary is static, pick random public keys for the $n-t$ honest nodes, and use $\mathcal{A}$ to run an instance of GRandPiper using the input secrets. When $\mathcal{A}$ wins, we can directly break the IND1-Secrecy property.
Remark. There are no known adaptively secure PVSS protocols. It is not the case that there are attacks on existing PVSS schemes when assuming an adaptive adversary, it is just that the existing proof techniques fail to show security against adaptive adversaries.

Lemma 5.3 (Rushing Adversary Advantage). *For any epoch $e \geq 1$, a rushing adversary can reconstruct output O_e at most 2Δ time before the honest nodes.*

Proof. An honest node sends its secret shares in epoch e when its epoch-timer$_{e-1}$ expires. Let node p_i be the earliest honest node whose epoch-timer$_{e-1}$ expires and node p_i sends its secret share at time τ. A rushing adversary may instantaneously receive the share and reconstruct the output O_e at time τ.

Due to the Δ delay among the honest nodes entering epoch e, the other honest nodes may send their secret shares only at time $\tau + \Delta$ which arrives at all the honest nodes by time $\tau + 2\Delta$. In the worst case, the honest nodes can reconstruct only at time $\tau + 2\Delta$. Thus, a rushing adversary can reconstruct output O_e at most 2Δ time before honest nodes. □

Lemma 5.4 (Guaranteed Beacon Output). *For any epoch $e \geq 1$, all the honest nodes output a new beacon output O_e.*

Proof. By Theorem 5.1 part (iv), all the honest nodes have consistent leaders. Let node p_i be the leader of epoch e. The honest nodes output a new beacon output in each epoch e if $Q(p_i) \neq \perp$. Suppose for the sake of contradiction $Q(p_i) = \perp$ in epoch e. Observe that nodes update $Q(p_i)$ with secret proposed in epoch e' (with $e' < e-t$) when p_i was an epoch leader in epoch e' by epoch e and node p_i did not propose any secrets in epoch e'. However, if p_i did not propose in epoch e', p_i would have been removed from the candidate leader set for epoch e and would not be epoch leader for epoch e and honest nodes would not use $Q(p_i)$ in epoch e. A contradiction.

Thus, all the honest nodes send secret shares for secret shared in $Q(p_i)$ and all the honest nodes will receive $t + 1$ valid shares to reconstruct a common randomness R_e and output O_e. □

Now, we show that GRandPiper is a secure random beacon protocol by showing that the individual properties are satisfied.

Lemma 5.5 (Bias-Resistance). *For any epoch $e \geq 1$, the probability that a t bounded adversary $\mathcal{A}$ can fix any c bits of the GRandPiper beacon output O_e is $\mathsf{negl}(c) + \mathsf{negl}(\kappa)$.*

Proof Sketch. The output in any epoch e is O_e which is the hash $H(R_e, O_{e-1}, \dots, O_{e-t})$. Assume that a static adversary $\mathcal{A}$ wants to bias c bits of O_e. Now there is at least one honest leader in epoch e' where $e - t \leq e' \leq e$. WLOG, assume that the leader at epoch $e' = e - t$ is honest. Then the output of epoch e' is known only in epoch e' within 2Δ time of entering the epoch e' (from Lemma 5.4). Therefore, a rushing adversary $\mathcal{A}$ can know the $O_{e'}$ at max 2Δ before an honest node enters epoch e' (from Lemma 5.3). But the adversary has to choose all $R_{e''}$ before epoch e', where $e - t < e'' \leq e$, so that it can bias O_e. But all blocks containing $R_{e''}$ are committed before the epoch e', since $R_{e''}$ comes from the blocks previously proposed by the leaders before epoch e' at the start (or during the setup). Thus all blocks containing $R_{e''}$ are proposed before observing $R_{e'}$, which is guaranteed to be secret for a honest node against $\mathcal{A}$ (from the *secrecy* property of Theorem 5.2) except with negligible probability $\mathsf{negl}(\kappa)$. Thus, an adversary $\mathcal{A}$ can do no better than $\mathsf{negl}(c) + \mathsf{negl}(\kappa)$ to fix c bits. □

Lemma 5.6 (GRandPiper unpredictability). *Assuming a secure PVSS scheme $\mathcal{PVSS}$ and SMR protocol, the GRandPiper random beacon protocol is an $O(\min(\kappa, t))$-absolute unpredictable random beacon protocol against a static adversary.*

Proof Sketch. Since the leaders are chosen using the beacon outputs, the probability that the adversary's nodes are chosen in an epoch e is $t/n < 1/2$. The probability that c consecutive leaders are Byzantine is therefore $\binom{t}{c}/(n-t)^c < 2^{-c}$ for $3 < c < t$ and is exponentially decreasing in c. The expected value of c is $\lceil \log 2 \rceil = 2$. If $c = t + 1$, the probability is already $\mathsf{negl}(\kappa)$ (from the probability of breaking secrecy of secrets shared by honest nodes). Thus, for a given security parameter κ, a static adversary cannot predict the output with better than $\mathsf{negl}(\kappa)$ probability in $\min(\kappa, t) + 1$ epochs. □

Theorem 5.7 (GRandPiper secure random beacon). *GRandPiper protocol is a $O(\min(\kappa, t))$-secure random beacon protocol assuming a static adversary.*

Proof. The proof follows from Lemma 5.5 for bias-resistance, Lemma 5.4 for guaranteed output delivery, and Lemma 5.6 for unpredictability. □

5.3 RandPiper – BRandPiper Protocol

In this section, we present a random beacon protocol using $O(\kappa f n^2)$ bits of communication complexity where $f \leq t$ is the actual number of faults and with 1-absolute unpredictability. Thus, in the optimistic case when $f = O(1)$, our communication complexity is quadratic.
In order to achieve 1-absolute unpredictability, we need to ensure that we reconstruct inputs from $> t$ nodes in every epoch. If we use PVSS schemes, we need to add $O(t)$ shares in every epoch, so that we can consume $> t$ combined shares in every round. A PVSS sharing for one secret is of size $O(n)$, and therefore performing $O(nt)$ sharings trivially results in a communication complexity of $O(n^3)$. Therefore, we will use VSS schemes (refer Appendix A for detailed VSS interface) in an attempt to improve the communication complexity for a 1-absolute unpredictable random beacon protocol.

5.3.1 *Improved VSS.*

We will first describe an improved VSS scheme that achieves better communication complexity to share n secrets in the optimistic case which will then be used in our random beacon protocol.

Efficient VSS (eVSS). eVSS [32] (refer Figure 5) presents the state-of-the-art VSS scheme for synchronous network setting. The protocol is described assuming the presence of a bulletin board (or broadcast channels) [8, 18, 20, 32] where there exists a public bulletin board, in which messages posted by any node are available instantly, and the bulletin board provides a consistent view to all the nodes. We can realize such message delivery guarantees by invoking Byzantine Broadcast (BB) protocols.

In this protocol, a dealer L creates a commitment $\mathcal{VSS}.\mathsf{C}$ to a random polynomial whose constant term is the secret, and posts the commitment on the bulletin board (Step 1), while *privately* sending individual shares $\mathcal{VSS}.\mathsf{s}_j$ along with witnesses $\mathcal{VSS}.\pi_j$ to every node $p_j \in \mathcal{P}$ (Step 2). Nodes post complaints on the bulletin board in the form of blame message if they do not receive valid shares (Step 3) in a timely manner. The dealer then opens the secret shares on the bulletin board corresponding to the nodes that blamed (Step 4). If there are $> t$ complaints, the nodes abort (Step 5). Otherwise, the honest nodes commit their shares (Step 5), with the guarantee that all honest nodes will be able to reconstruct the shared secret.

Note that $f \leq t$ Byzantine nodes can always blame regardless of the dealer being honest or not. This forces an honest dealer to post $O(fn)$ shares on the bulletin board when secret sharing $O(n)$ secrets. In general, the amount of information posted on the bulletin board is $O(\kappa n + \kappa f + \kappa f n)$ corresponding to $O(n)$ commitments, f blame messages and $O(fn)$ opened secret shares. A naïve approach of using BB protocols (extension protocols [36] for larger inputs) to instantiate the bulletin board involves following steps:

(1) *Commitment and sharing.* Dealer L invokes BB to broadcast n commitments Step 1, while privately sharing individual shares Step 2.
(2) *Blame.* Nodes invoke n parallel instances of BB to broadcast blame messages Step 3.
(3) *Open shares.* Dealer L invokes an instance of BB with secret shares corresponding to the blames received.

We note that state-of-the-art honest majority BB protocols, without threshold signatures, incur $O(\kappa n^3)$ bits communication cost to achieve consensus on a single decision [3, 23, 33]. Thus, invoking n parallel instances of BB trivially incurs $O(\kappa n^4)$ communication cost. In addition, running BB on inputs of size $O(fn)$ incurs $O(\kappa f n^3)$ without threshold signatures and extension techniques. Thus, the total communication complexity is $O(\kappa n^4)$ bits.

Improved eVSS (iVSS). In order to reduce the large communication overhead, we first present an improved VSS scheme, that reduces (i) the number of posts to the bulletin board, and (ii) the amount of information posted on the bulletin board.

Let $\mathcal{VSS}$ be the VSS scheme being used. Let $\mathcal{VSS}$.pp be the public VSS parameters. Let L be a dealer with secret s. Assuming the existence of a bulletin board, each node $p_i \in \mathcal{P}$ does the following:

1. **Post commitment.** If p_i is L, then generate shares for every node by running $(\mathcal{VSS}.\vec{S}, \mathcal{VSS}.\vec{W}, \mathcal{VSS}.\mathsf{C}) \leftarrow \mathcal{VSS}.\mathsf{ShGen}(s)$, and post the commitment $\mathcal{VSS}.\mathsf{C}$ to the secret s on the bulletin board.
2. **Send shares.** If p_i is L, then send shares $\mathcal{VSS}.\mathsf{s}_j \in \mathcal{VSS}.\vec{S}$ and witness $\mathcal{VSS}.\pi_j \in \mathcal{VSS}.\vec{W}$ over the confidential channel to all nodes $p_j \in \mathcal{P}$.
3. **Send blames.** Post complaints $\langle \mathsf{blame}, L\rangle_i$ on the bulletin board, if no valid share is received privately or if $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}_i, \mathcal{VSS}.\pi_i, \mathcal{VSS}.\mathsf{C}) = 0$.
4. **Open shares.** For all blames $\langle \mathsf{blame}, L\rangle_i$, if p_i is L, post their shares $\mathcal{VSS}.\mathsf{s}_j$ and witnesses $\mathcal{VSS}.\pi_j$ on the bulletin board.
5. **Decide.** If the published share and witness satisfies $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}_k, \mathcal{VSS}.\pi_k, \mathcal{VSS}.\mathsf{C}) = 1$ for every blame, and there are only up to $f \leq t$ blames on the bulletin board, then commit $\mathcal{VSS}.\mathsf{s}_j$. Otherwise, abort, i.e., output $\perp$.

Figure 5: eVSS [32] protocol description. This scheme is to secret share one secret.

Let $\mathcal{VSS}$ be the VSS scheme being used. Let $\mathcal{VSS}$.pp be the public VSS parameters. Let L be a dealer with n secrets $S := \{s_1, \ldots, s_n\}$ it wishes to secret share with nodes $\mathcal{P}$. Assuming the existence of a bulletin board, each node $p_i \in \mathcal{P}$ does the following:

(1) **Post commitment.** If p_i is L, run $(\mathcal{VSS}.\vec{S}_i, \mathcal{VSS}.\vec{W}_i, \mathcal{VSS}.\mathsf{C}_i) \leftarrow \mathcal{VSS}.\mathsf{ShGen}(s_i)$ for all $s_i \in S$. Build the commitment vector $\mathcal{VSS}.\vec{C} := \{\mathcal{VSS}.\mathsf{C}_1, \ldots, \mathcal{VSS}.\mathsf{C}_n\}$ which contain commitments $\mathcal{VSS}.\mathsf{C}_i$ for s_i. Post $\mathcal{VSS}.\vec{C}$ on the bulletin board.

(2) **Send shares.** If p_i is L, collect shares and witnesses $(\mathcal{VSS}.\mathsf{s}_j, \mathcal{VSS}.\pi_j)$ for every node $p_j \in \mathcal{P}$, and secret $s_i \in S$, and build $\mathcal{VSS}.\vec{S}_j, \mathcal{VSS}.\vec{W}_j$. Send $(\mathcal{VSS}.\vec{S}_j, \mathcal{VSS}.\vec{W}_j)$ to node $p_j \in \mathcal{P}$.
    If $p_i \in \mathcal{P}$ is not the dealer L, then wait to obtain $(\mathcal{VSS}.\vec{S}_i, \mathcal{VSS}.\vec{W}_i)$ from the dealer L, and ensure that $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\vec{C}_j) = 1$ holds for $\mathcal{VSS}.\mathsf{s}_j \in \mathcal{VSS}.\vec{S}_i$, and $\mathcal{VSS}.\pi_j \in \mathcal{VSS}.\vec{W}_i$.

(3) **Send blames.** If invalid/no shares are received from the dealer L, then send $\langle \mathsf{blame}, L\rangle_i$ to *all* the nodes. Collect similar blames from other nodes.

(4) **Private open.** Send all the collected blames to the dealer L. If p_i is the leader, then for every blame $\langle \mathsf{blame}, L\rangle_k$ received from node p_j, send $(\mathcal{VSS}.\vec{S}_k, \mathcal{VSS}.\vec{W}_k)$ to node p_j.
    If p_i is not L, then ensure that $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\vec{C}_j) = 1$ for every $\langle \mathsf{blame}, L\rangle_j$, $\mathcal{VSS}.\mathsf{s}_j \in \mathcal{VSS}.\vec{S}_j$, and $\mathcal{VSS}.\pi_j \in \mathcal{VSS}.\vec{W}_j$.

(5) **Ack and decide.** If p_i received $\leq t$ blames and the leader responded with valid shares $(\mathcal{VSS}.\vec{S}_j, \mathcal{VSS}.\vec{W}_j)$ for every $\langle \mathsf{blame}, L\rangle_j$ it forwarded, then send an ack message to the dealer L.
    If p_i is L, then post ack certificate (denoted by $\mathcal{AC}(\mathcal{VSS}.\vec{C})$) on the bulletin board.
    If there is an ack certificate $\mathcal{AC}(\mathcal{VSS}.\vec{C})$ on the bulletin board, commit $\mathcal{VSS}.\vec{C}$, and send $(\mathcal{VSS}.\vec{S}_j, \mathcal{VSS}.\vec{W}_j)$, if received from L.

(6) **Reconstruction.** Each node $p_i \in \mathcal{P}$ does the following:
  (a) If there is a share $\mathcal{VSS}.\mathsf{s}_i, \mathcal{VSS}.\pi_i$, send the share and witness to all the nodes.
      On receiving a share and witness $(\mathcal{VSS}.\mathsf{s}_j, \mathcal{VSS}.\pi_j)$ from p_j, ensure that $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\mathsf{C}) = 1$.
  (b) On receiving $t+1$ valid shares in $\mathcal{VSS}.\vec{S}$, reconstruct the secret s using $s \leftarrow \mathcal{VSS}.\mathsf{Recon}(\mathcal{VSS}.\vec{S})$. Send s to all the nodes.
  (c) On receiving an opened secret s, ensure that $\mathcal{VSS}.\mathsf{ComVrfy}(\mathcal{VSS}.\mathsf{C}, s) = 1$ and output s.

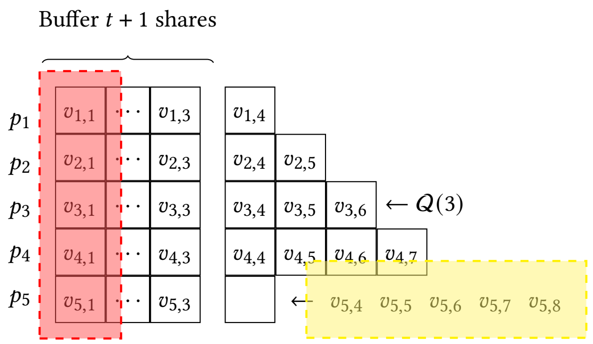
Figure 6: iVSS - Improved eVSS protocol description

In iVSS (refer Figure 6), the dealer posts commitments on the bulletin board, privately sends the secret shares and corresponding witnesses similar to eVSS. However, unlike eVSS, nodes send the blame messages to all nodes. In addition, nodes forward the received blame messages to the dealer to request for missing shares. The dealer *privately* sends missing shares to the nodes that forwarded the blame message instead of posting on the bulletin board. If an honest node receives missing shares for all blame messages it forwarded, it sends an ack to the dealer. The dealer collects $t+1$ ack messages and posts the ack certificate on the bulletin board. An honest node commits the proposed commitment if it observes an ack certificate on the bulletin board.

The honest nodes then forward the missing shares if the dealer sent the missing shares. A key correctness argument for our scheme is the following: if an honest node $p_i \in \mathcal{P}$ does not receive commitments and secret shares, it must have sent blame messages to all honest nodes. If some honest node $p_j \in \mathcal{P}$ sends an ack message, it must have received missing shares corresponding to the blame messages it received and forwarded (which includes share for p_i). Thus, all the honest nodes shares to reconstruct the proposed secrets.

We note that both eVSS and iVSS schemes guarantee secrecy (see Definition 3.4) only when the dealer is honest. If t Byzantine nodes send a blame message, then an honest but curious node can violate secrecy, however this was also possible in the bulletin board based protocol and can be easily solved by assuming an additional honest node, i.e., $n > 2t+1$.

5.3.2 *Random Beacon for BRandPiper.* In this section, we instantiate bulletin boards using our SMR protocol (Section 4) and present a random beacon protocol, we call BRandPiper, using the iVSS scheme. If we use our SMR protocol with rotating leaders, we can commit blocks of size $O(\kappa n)$ within $t+1$ epochs while incurring $O(\kappa n^2)$ bits of communication per epoch. To obtain 1-absolute unpredictability, we need to reconstruct at least $t+1$ secrets from *distinct* nodes in each epoch. For simplicity, we reconstruct one secret from *all* nodes that have not been removed. Using the round-robin leader selection rule, every node can share secrets at least once every n epochs. If in every epoch, the leader proposes commitments to n secrets using SMR protocol, we can use these secrets for the next n epochs in the reconstruction. Our beacon output step can take advantage of the homomorphic properties of the underlying VSS scheme $\mathcal{VSS}$ to combine secret shares for multiple secrets from different nodes into an $O(\kappa)$-sized share which can be efficiently broadcast to all nodes. Honest nodes collect $t+1$ homomorphic shares to reconstruct the common randomness R_e. Such reconstructed randomness is guaranteed to be unbiasable since an adversary cannot know the secrets of honest nodes until reconstructed, and an adversary cannot prevent reconstruction. For the same reason, our BRandPiper protocol ensures 1-absolute unpredictability, even for a rushing adaptive adversary.



Buffer $t+1$ shares
p_1
p_2
p_3
p_4
p_5
$v_{1,1}$ $\cdots$ $v_{1,3}$ $v_{1,4}$
$v_{2,1}$ $\cdots$ $v_{2,3}$ $v_{2,4}$ $v_{2,5}$
$v_{3,1}$ $\cdots$ $v_{3,3}$ $v_{3,4}$ $v_{3,5}$ $v_{3,6}$ $\leftarrow Q(3)$
$v_{4,1}$ $\cdots$ $v_{4,3}$ $v_{4,4}$ $v_{4,5}$ $v_{4,6}$ $v_{4,7}$
$v_{5,1}$ $\cdots$ $v_{5,3}$ $\leftarrow$ $v_{5,4}$ $v_{5,5}$ $v_{5,6}$ $v_{5,7}$ $v_{5,8}$

Figure 7: An example illustration of BRandPiper for $n = 5$ and $t = 2$ in epoch $e = 5$. The region marked in red are the shares that will be homomorphically combined in every epoch for reconstruction. In general, in every epoch, shares from the left-most column will be used for reconstruction. The region marked in yellow is the addition of n new shares by the leader $L_e = p_5$.

Example. Consider an example scenario as shown in Figure 7. In epoch 5, p_5 proposes and adds n $\mathcal{VSS}$ shares to the system which will be committed within $t+1 = 3$ epochs. If p_5 is Byzantine, by the end of epoch 8, all the nodes will remove p_5 from future proposals, thus guaranteeing outputs for every epoch. Until epoch 8 observe that we have shares for p_5.

Protocol Details. **Leader selection.** We employ a round robin leader selection policy. If an epoch leader p_i fails to commit within $t+1$ epochs, it is added to the set of removed nodes $\mathcal{P}_r$ and is prevented from being a future leader. The remaining nodes $\mathcal{P} \setminus \mathcal{P}_r$ propose in a round robin manner.

Setup. During the *setup* phase, all the nodes are provided with VSS parameters $\mathcal{VSS}.\mathsf{pp}$ required for using the VSS scheme $\mathcal{VSS}$. Each node maintains n queues $Q(p_i)$, for $p_i \in \mathcal{P}$. Each queue $Q(p_i)$ holds tuples with each tuple containing a secret share, its witness and commitment proposed by node p_i when node p_i was an epoch leader. During the setup phase, each queue $Q(p_i)$ is filled with $m = n + t$ tuples containing secret shares, witnesses and commitments for m secrets. This ensures that all honest nodes have common secret shares in $Q(p_n)$ and can perform $\texttt{Dequeue}(Q(p_n))$ up to epoch $n+t$ even if p_n does not propose in epoch n. This is because honest nodes perform $\texttt{Dequeue}(Q(p_n))$ (explained later) in each epoch unless node p_n has been removed. If node p_n does not propose in epoch n, it is removed only in epoch $n+t$.

Block validation protocol. BRandPiper uses a block validation protocol to generate valid blocks for use in the SMR. A *valid* block in BRandPiper is a vector of VSS commitments $\mathcal{VSS}.\vec{C}$ along with acks from $t+1$ nodes. The block validation protocol is essentially an instance of iVSS where the leader ends up with $t+1$ votes for a VSS commitment vector $\mathcal{VSS}.\vec{C}$. The commitment and ack certificate is then input to the SMR protocol to ensure that all honest nodes agree on a single commitment vector. During the SMR protocol, the honest nodes vote only if a *valid* block is produced via the block validation protocol. The block validation protocol guarantees that if a block is certified, then all the honest nodes have sharings for all the secrets committed in $\mathcal{VSS}.\vec{C}$. When these commitments are committed via SMR, all the honest nodes use the secret shares in the commitments in different epochs to generate the common randomness.

The block validation protocol (refer Figure 8) is executed in parallel with SMR protocol. The leader L_e of epoch e executes the block validation protocol while in epoch $e-1$ to generate an ack certificate for commitments to be proposed in epoch e. The protocol consists of following steps:

<u>Distribute.</u> Leader L_e creates n commitments $\mathcal{VSS}.\vec{C}$ corresponding to n secrets $\{s_1,\ldots,s_n\}$ it wishes to share using $\mathcal{VSS}.\mathsf{ShGen}$ algorithm for secrets $\{s_i | \forall 1 \le i \le n\}$, along with shares $\mathcal{VSS}.\vec{S}_j := \{\mathcal{VSS}.s_{1,j},\ldots,\mathcal{VSS}.s_{n,j}\}$ and witnesses $\mathcal{VSS}.\vec{W}_j := \{\mathcal{VSS}.\pi_{1,j}, \ldots, \mathcal{VSS}.\pi_{n,j}\}$, for all $p_j \in \mathcal{P}$. We define a block containing n commitments $\mathcal{VSS}.\vec{C}$ as $SB := \langle \texttt{Commitment}, \mathcal{VSS}.\vec{C}, e, z_{se}\rangle_{L_e}$. The leader L_e sends $\mathcal{VSS}.\vec{S}_j$, $\mathcal{VSS}.\vec{W}_j$, and SB to node p_j, for all $p_j \in \mathcal{P}$. Similar to the SMR protocol, the leader signs the tuple $\langle \texttt{Commitment}, H(\mathcal{VSS}.\vec{C})), e, z_{se}\rangle$ and sends $\mathcal{VSS}.\vec{C}$ separately to facilitate efficient equivocation checks. It is important to note that commitment $\mathcal{VSS}.\vec{C}$, shares $\mathcal{VSS}.\vec{S}_j$, and witness $\mathcal{VSS}.\vec{W}_j$ are $O(n)$-sized and the shares $\mathcal{VSS}.s_j$ are only sent to node p_j. Sending only the required shares to designated nodes reduces communication complexity.

<u>Blame/Forward.</u> If a node p_i receives a valid secret share $\mathcal{VSS}.\vec{S}_i$, witness $\mathcal{VSS}.\vec{W}_i$, and sharing block $SB := \langle \texttt{Commitment}, \mathcal{VSS}.\vec{C}, e, z_{se}\rangle_{L_e}$ within 3Δ time in epoch $e-1$, it invokes $\texttt{Deliver}(\texttt{Commitment}, SB, z_{se}, e)$. A valid share satisfies $\mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.s_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\vec{C}_j) = 1, \forall j \in [n]$. Otherwise, node p_i broadcasts $\langle \text{blame}, e\rangle_i$ to all nodes.

<u>Request open.</u> Node p_i waits for 6Δ time in epoch $e-1$ to collect any blames sent by other nodes. If no blames or equivocation by L_e has been detected within that time, p_i sends $\langle \text{ack}, H(SB), e\rangle_i$ to L_e. If up to t blames are received, p_i forwards the blames to L_e.

<u>Private open.</u> If L_e receives any blames from node p_i, it sends valid $\mathcal{VSS}.\vec{S}_j$, witness $\mathcal{VSS}.\vec{W}_j$ for every blame $\langle \text{blame}, e\rangle_j$ received from node p_i.

<u>Ack.</u> If node p_i forwarded any blames and received valid secret shares $\mathcal{VSS}.\vec{S}_j$ and witness $\mathcal{VSS}.\vec{W}_j$ for every blame $\langle \text{blame}, e\rangle_j$ it forwarded and detects no equivocation, node p_i sends $\langle \text{ack}, H(SB), e\rangle_i$ to L_e. In addition, node p_i forwards secret shares $\mathcal{VSS}.\vec{S}_j$ and witness $\mathcal{VSS}.\vec{W}_j$ for every blame $\langle \text{blame}, e\rangle_j$ it received. Thus, if an honest node sends an ack for the sharing block SB, then all honest

This protocol is executed in parallel with BFT SMR protocol in Figure 1 using the round-robin leader selection. Let L_e be the leader of epoch e and the current epoch be $e-1$. Node p_i performs following operations while in epoch $e-1$:

(1) **Distribute.** L_e waits for Δ time after entering epoch $e-1$ and then does the following:
- Let $\{s_1, \ldots, s_n\}$ be n random numbers chosen uniformly from the input space of $\mathcal{VSS}$.
- Build $SB := \langle \texttt{Commitment}, \mathcal{VSS}.\vec{C}, e, z_{se}\rangle_{L_e}$, the sharing block which consists of commitments $\mathcal{VSS}.\vec{C} := \{\mathcal{VSS}.\mathrm{C}_1, \ldots, \mathcal{VSS}.\mathrm{C}_n\}$ to the n random numbers generated by running $(\mathcal{VSS}.\vec{S}_i, \mathcal{VSS}.\vec{W}_i, \mathcal{VSS}.\mathrm{C}_i) \leftarrow \mathcal{VSS}.\mathsf{ShGen}(s_i)$ for $i \in \{1, \ldots, n\}$, where $\mathcal{VSS}.\vec{S}_i := \{\mathcal{VSS}.\mathrm{s}_{i,1}, \ldots, \mathcal{VSS}.\mathrm{s}_{i,n}\}$, and $\mathcal{VSS}.\vec{W}_i := \{\mathcal{VSS}.\pi_{i,1}, \ldots, \mathcal{VSS}.\pi_{i,n}\}$.
- Build the share vector $\mathcal{VSS}.\vec{S}_j := \{\mathcal{VSS}.\mathrm{s}_1 \leftarrow \mathcal{VSS}.\vec{S}_{1,j}, \ldots, \mathcal{VSS}.\mathrm{s}_n \leftarrow \mathcal{VSS}.\vec{S}_{n,j}\}$ and the witness vector $\mathcal{VSS}.\vec{W}_j := \{\mathcal{VSS}.\pi_1 \leftarrow \mathcal{VSS}.\vec{W}_{1,j}, \ldots, \mathcal{VSS}.\pi_n \leftarrow \mathcal{VSS}.\vec{W}_{n,j}\}$ for node p_j using j^{th} share and witness from $\mathcal{VSS}.\vec{S}_i$ and $\mathcal{VSS}.\vec{W}_i$ for random number s_i.
- Send $\mathcal{VSS}.\vec{S}_j$, $\mathcal{VSS}.\vec{W}_j$, and SB to every node $p_j \in \mathcal{P}$.

(2) **Blame/Forward.** If epoch-timer$_{e-1} \geq 8\Delta$ and node p_i receives valid share vector $\mathcal{VSS}.\vec{S}_i$, witness vector $\mathcal{VSS}.\vec{W}_i$ and commitment $SB := \langle \texttt{Commitment}, \mathcal{VSS}.\vec{C}, e, z_{se}\rangle_{L_e}$, then invoke $\mathsf{Deliver}(\texttt{Commitment}, SB, z_{se}, e)$. If no shares has been received within 3Δ time while in epoch $e-1$, broadcast a blame $\langle \text{blame}, e\rangle_i$ to all nodes.

(3) **Request open.** Wait until epoch-timer$_{e-1} \geq 5\Delta$. Collect all blames received so far. If up to t blames are received so far, forward the blames to L_e. If no blames or equivocation by L_e has been detected, send $\langle \text{ack}, H(SB), e\rangle_i$ to L_e.

(4) **Private open.** L_e sends valid share $\mathcal{VSS}.\vec{S}_j$ and witness $\mathcal{VSS}.\vec{W}_j$ to node p_i, for every blame $\langle \text{blame}, e\rangle_j$ received from node p_i.

(5) **Ack.** Upon receiving valid share $\mathcal{VSS}.\vec{S}_j$ and witness $\mathcal{VSS}.\vec{W}_j$ for every $\langle \text{blame}, e\rangle_j$ it forwarded and detects no equivocation, send $\langle \text{ack}, H(SB), e\rangle_i$ to L_e. Forward share $\mathcal{VSS}.\vec{S}_j$ and witness $\mathcal{VSS}.\vec{W}_j$ to node p_j for every $\langle \text{blame}, e\rangle_j$ it received.

(6) **(Non-blocking) Equivocation.** Broadcast equivocating hashes signed by L_e and stop performing any operations.

Figure 8: Block validation protocol.

nodes have their respective secret shares corresponding to sharing block SB (more details in Lemma 5.9).

Equivocation. At any time in epoch $e-1$, if a node p_i detects an equivocation, it broadcasts equivocating hashes signed by L_e and stops participating in epoch $e-1$ block validation protocol.

Beacon protocol. We now present the beacon protocol (refer Figure 9) in BRandPiper. It consists of the following rules for an epoch e. Here, an epoch corresponds to an epoch in SMR protocol.

Generate Blocks. The leader L_e of an epoch e chooses n secrets uniformly at random and invokes the block validation protocol while in epoch $e-1$ to obtain an ack certificate (denoted by $\mathcal{AC}_e(SB)$), to generate a valid block SB corresponding to the n secrets. In epoch e, the leader proposes block B_h with $b_h := (H(SB), \mathcal{AC}_e(SB))$ where $\mathcal{AC}_e(SB)$ is an ack certificate for commitment SB using the SMR protocol obtained from the block validation protocol. We redefine valid blocks for the SMR protocol with an additional constraint to contain an ack certificate created in epoch $e-1$[3] and all honest nodes vote in the SMR protocol as long as the proposed block meets this additional constraint. As mentioned before, an ack certificate for a sharing block SB implies all honest nodes have secret shares required to reconstruct the secrets corresponding to commitments in SB. Thus, it is safe for honest nodes to vote in the SMR protocol although they sent blame during the block validation phase.

Update. At the end of epoch e, node p_i updates $Q(L_{e-t})$ as follows. If L_{e-t} proposed a valid block B_l in epoch $e-t$ and B_l has been committed by epoch e, node p_i replaces the contents of $Q(L_{e-t})$ with n tuples with each tuple containing secret shares, witnesses and commitments shared in epoch $e-t$. If no epoch $e-t$ block was committed, it removes L_{e-t} from future proposals, i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{e-t}\}$. It is important to note that the SMR protocol guarantees all honest nodes commit proposed blocks in $t+1$ epochs. Thus, all honest nodes either update $Q(L_{e-t})$ or remove L_{e-t} in epoch e.

[3]For the first epoch, an ack certificate can be created during the setup phase.

Reconstruct. At the end of epoch e, all nodes perform $\mathsf{Dequeue}(Q(p_j))$, $\forall p_j \notin \mathcal{P}_r$, to fetch n secret shares (one from each node) and corresponding witnesses. The nodes compute the homomorphic sum of shares and witnesses and broadcast it to all other nodes.

Output. From the above discussion, it is clear that all honest nodes send homomorphic sum of shares for common commitments and all honest nodes will receive at least $t+1$ valid homomorphic shares. When a node p_i receives $t+1$ homomorphic shares, it reconstructs the randomness R_e using $\mathcal{VSS}.\mathsf{Recon}$ primitive and computes $O_e \leftarrow H(R_e)$.

5.4 BRandPiper Security Analysis

Theorem 5.8 (Security of iVSS). *The verifiable secret sharing scheme proposed in Figure 6 is a secure verifiable secret sharing scheme assuming a bulletin board.*

Proof Sketch. Consider any secure VSS scheme $\mathcal{VSS}$. The view V_i of an honest node is $V_i := (\mathcal{VSS}.\mathrm{C}, \mathcal{VSS}.\mathrm{s}i, \mathcal{VSS}.\pi i)$ to every node p_i in both $\mathcal{VSS}$ and the iVSS protocol. Any t-bounded adversary with access to t views in both the protocols, has an equal probability of extracting the secret. The case where the adversary forges the digital signatures to obtain $t+1$ acks, which happens with negligible probability, is an extra case to consider for the commitment and correctness properties.

Formally, assume a secure VSS scheme satisfying Definition 3.4.

Secrecy: If the dealer L is honest, then no honest node will blame and the maximum number of blames is at most t. Thus, only up to t blames will be opened privately by the leader. Therefore, the view V_T of an adversary corrupting $T \subset [n]$ nodes with $|T| \leq t$ has the same view in both the protocols.

Correctness: If the dealer L is honest, then all honest nodes have their shares for the secret s, and similar to eVSS, will output the same secret s except with $\mathsf{negl}(\kappa)$ probability, where the probability is over forging digital signatures.

Let $\mathcal{VSS}$ be the VSS scheme used, e be the current epoch and L_e be the leader of epoch e. Node $p_i \in \mathcal{P}$ augments SMR protocol in Figure 1 as follows:

- **Setup.** Set $e = 1$. All nodes agree upon and fill $Q(p_i)$ with $m = n + t$ tuples $\forall p_i \in \mathcal{P}$. Set $\mathcal{P}_r \leftarrow \emptyset$. Run $\mathcal{VSS}$.Setup and agree on the public parameters $\mathcal{VSS}$.pp. Set $L_e \leftarrow p_1$.
- **Blocks.** While in epoch $e-1$, leader L_e starts the block validation protocol (refer Figure 8) with $\{s_1, \ldots, s_n\}$, where the secrets are chosen randomly $s_i \leftarrow\$ \{0,1\}^\kappa$ for $1 \le i \le n$.
  In epoch e, L_e proposes block B_h with $b_h := (H(SB), \mathcal{AC}_e(SB))$ where $\mathcal{AC}_e(SB)$ is an ack certificate for commitment SB.
- **Update.** When epoch-timer$_e$ expires, if L_{e-t} proposed a valid block B_l in epoch $e-t$ and B_l has been committed by epoch e, update $Q(L_{e-t})$ with n tuples with each tuple containing secret shares, witnesses and commitments shared in epoch $e-t$. Otherwise, remove L_{e-t} from future proposals i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{e-t}\}$.
- **Reconstruct.** When epoch-timer$_e$ expires, do the following:
  (1) Get $(\mathcal{VSS}.\vec{S}, \mathcal{VSS}.\vec{W}, \mathcal{VSS}.\vec{C}) := \{\mathsf{Dequeue}(Q(p_j)) \mid p_j \notin \mathcal{P}_r\}$.
  (2) Build homomorphic sum share SV_i, witness $\mathcal{VSS}.\pi_i$, and commitment $\mathcal{VSS}.\mathsf{C}_e$ using all shares from $\mathcal{VSS}.\vec{C}$. Send SV_i and $\mathcal{VSS}.\pi_i$ to all the nodes.
  (3) Upon receiving share SV_j and witness $\mathcal{VSS}.\pi_j$ for $\mathcal{VSS}.\mathsf{C}_e$, ensure that $\mathcal{VSS}.\mathsf{ShVrfy}(SV_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\mathsf{C}_e) = 1$.
  (4) Upon receiving $(t+1)$ valid homomorphic sum shares in SV, obtain $R_e \leftarrow \mathcal{VSS}.\mathsf{Recon}(SV)$.
- **Output.** Compute and output $O_e \leftarrow H(R_e)$.

Figure 9: RandPiper – BRandPiper beacon protocol.

Commitment: If an ack certificate is formed, irrespective of the leader being honest or Byzantine, at least one honest node has not observed $\ge t+1$ blames, and has received valid shares for every blame. This honest node, say p_i has all the shares for every honest node that does not have a share. Therefore, all honest nodes together have $t+1$ shares, which guarantees reconstruction to the unique secret s that was committed except with $\mathsf{negl}(\kappa)$ probability. If no ack certificate is formed, then all the honest nodes, agree on $\perp$, thus satisfying the *Commitment* requirement with high probability of $1 - \mathsf{negl}(\kappa)$, where the probability is over forging digital signatures and the adversary generating incorrect witnesses. □

Lemma 5.9. *If an honest node sends an* ack *for a sharing block SB in epoch e, then (i) all honest nodes receive the sharing block SB in epoch e, (ii) all honest nodes receive their respective secret shares corresponding to sharing block SB within Δ time of entering epoch $e+1$.*

Proof. Suppose an honest node p_i sends an ack for sharing block $SB := \langle \mathtt{Commitment}, \mathcal{VSS}.\vec{C}, e, z_{se}\rangle_{L_e}$ at time τ in epoch e. Node p_i must have received up to t blame messages. This implies at least one honest node p_j received a valid share $\mathcal{VSS}.\mathsf{s}_i$ and sharing block SB within 3Δ time in epoch e and invoked Deliver($\mathtt{Commitment}, SB, z_{se}, e$). Let τ' be the time when node p_j invoked Deliver($\mathtt{Commitment}, SB, z_{se}, e$). The earliest node p_i sends an ack for SB is when it waits until epoch-timer$_e \ge 5\Delta$ (i.e., 6Δ in epoch e) and does not detect any equivocation by L_e or any blame messages. Due to Δ delay between honest nodes entering into epoch e, this time corresponds to $\tau' + 2\Delta$ in the worst case. This implies no honest node received an epoch e equivocation by time $\tau' + \Delta$. By Fact 4.1, all honest nodes receives the sharing block SB. This proves part (i) of the Lemma.

For part (ii), node p_i can send ack on two occasions: (a) when it does not detect any equivocation or blame until its epoch-timer$_e \ge 5\Delta$, and (b) when leader L_{e+1} sent valid secret shares for every blame message it forwarded and does not detect any equivocation by time τ.

In case (a), node p_i did not detect any equivocation or blame messages until its epoch-timer$_e > 5\Delta$ at time τ. Observe that all honest nodes must have received valid shares corresponding to the sharing block SB within 3Δ time in epoch e; otherwise node p_i must have received blame message by time τ (since honest nodes may enter epoch e with Δ time difference and send blame message if no valid secret shares received within 3Δ time in epoch e). In addition, no honest node received an equivocating sharing block SB' within 3Δ time in epoch e; otherwise, node p_i must have received a share for SB' (via Deliver) by time τ. Thus, all honest nodes receive their respective secret shares corresponding to sharing block SB in epoch e (i.e., within Δ time of entering epoch $e+1$).

In case (b), node p_i receives valid secret shares from leader L_{e+1} for every blame (up to t blame) messages it forwarded and detected no equivocation by time τ. Observe that node p_i received $f \le t$ blame messages and received valid shares for every blame message it forwarded. This implies at least $n - t - f$ honest nodes have received valid shares for sharing block SB from leader L_{e+1} within 3Δ in epoch e; otherwise, node p_i would have received more than f blame message by the time its epoch-timer$_e = 5\Delta$. Since, node p_i forwards f received secret shares corresponding to f received blame message in epoch e and honest nodes enter epoch $e+1$ within Δ time, all honest nodes receive their respective secret shares corresponding to sharing block SB within Δ time of entering epoch $e+1$.

□

Theorem 5.10 (Consistent Beacon). *For any epoch e, all honest nodes reconstruct the same randomness R_e and output the same beacon O_e.*

Proof. Honest nodes output the same randomness R_e and output the same beacon O_e in epoch e if all honest nodes receive $t+1$ valid homomorphic shares for the same set of secrets. This condition is satisfied if all honest nodes (i) have consistent $Q(p_i), \forall p_i \in \mathcal{P}$ and consistent $\mathcal{P}_r$ in each epoch, (ii) $\{\mathsf{Dequeue}(Q(p_i)) \ne \perp, \forall p_i \in \mathcal{P} \setminus \mathcal{P}_r\}$ in each epoch, and (iii) share valid homomorphic shares corresponding to dequeued secret shares.

For part(i), we show all honest nodes have consistent $Q(p_i)$, $\forall p_i \in \mathcal{P}$ and consistent $\mathcal{P}_r$ in every epoch.

We prove part (i) by induction on epochs. Consider the base case for epochs 1 to t. During setup phase, each node is assigned $m = n+t$ tuples (with each tuple containing secret shares, witnesses and commitments) for each $Q(p_i)$, $\forall p_i \in \mathcal{P}$ (i.e., $m * n$ secrets in total). Since, removing a Byzantine node requires $t+1$ epochs, all honest nodes have $\mathcal{P}_r = \emptyset$ for epochs 1 to t. In addition, no honest node update $Q(p_i)$ during epochs 1 to t. Thus, for epochs 1 to t, all honest nodes have consistent $Q(p_i)$, $\forall p_i \in \mathcal{P}$ and $\mathcal{P}_r$.

We assume part(i) holds until epoch $e-1$.

Consider an epoch $e > t$. In epoch e, all honest nodes update only $Q(L_{e-t})$. If L_{e-t} proposed a valid block B_l (with $b_l = (H(SB), \text{ack-cert}(SB))$ for some commitment SB and B_l is committed by epoch e, all honest nodes update $Q(L_{e-t})$ with n tuples containing secret shares, witnesses and commitments in SB (by Lemma 5.9, all honest nodes receive commitments and secret shares in SB before epoch e). Otherwise, all honest nodes update $\mathcal{P}_r$ to exclude L_{e-t} i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{e-t}\}$. Thus, all honest nodes should have consistent $Q(L_{e-t})$ by epoch e. Since honest nodes do not update $Q(p_i \neq L_{e-t})$ and do not add p_i into $\mathcal{P}_r$ in epoch e, by induction hypothesis, all honest nodes should have consistent $Q(p_i)$ $\forall p_i \in \mathcal{P}$ and consistent $\mathcal{P}_r$ in epoch e. This proves part(i). Since, all honest nodes have a consistent $Q(p_i)$ $\forall p_i \in \mathcal{P}$ and consistent $\mathcal{P}_r$, all honest nodes perform $\{\text{Dequeue}(Q(p_i)) \forall p_i \in \mathcal{P} \setminus \mathcal{P}_r\}$ for common secrets.

Next, we show $\{\text{Dequeue}(Q(p_i)) \neq \perp \forall p_i \in \mathcal{P} \setminus \mathcal{P}_r\}$ in epoch e. Suppose for the sake of contradiction, $\text{Dequeue}(Q(p_i)) = \perp$ and $p_i \notin \mathcal{P}_r$ in epoch e. Observe that, honest nodes update $Q(p_i)$ or include p_i in $\mathcal{P}_r$ $t+1$ epochs after node p_i becomes an epoch leader. Let e' be the last epoch in which node p_i last proposed with $e' \leq e - t$. However, if node p_i did not propose in e', all honest nodes would have removed p_i by epoch $e' + t \leq e$ and $p_i \in \mathcal{P}_r$ in epoch e. A contradiction.

Finally, we show all honest nodes send valid homomorphic shares for the dequeued secret shares. Observe that honest nodes only dequeue secret shares corresponding to a committed block that contains a valid ack certificate. By Lemma 5.9 part(ii), all honest nodes receive valid secret shares before honest nodes update their queues. Thus, all nodes will dequeue common secret shares and will receive at least $t+1$ valid homomorphic shares for a common secrets and reconstruct the same randomness R_e and output the same beacon O_e. □

Lemma 5.11 (Liveness)**.** *If the leader L_e of an epoch e is honest, then (i) an* ack *certificate for its sharing block SB will form in epoch $e-1$, and (ii) all honest nodes commit $(H(SB), \mathcal{AC}_e(SB))$ in epoch e.*

Proof. Consider an honest leader L_e for an epoch e. Let τ be the time when leader L_e enters epoch $e-1$. Leader L_e waits for Δ time after entering epoch $e-1$ and must have sent valid shares $\mathcal{VSS}.s_i$ and sharing block SB containing commitments to node p_i $\forall p_i \in \mathcal{P}$ at time $\tau + \Delta$.

Since honest nodes enter epoch $e-1$ within Δ time, all honest nodes must have entered epoch $e-1$ by time $\tau + \Delta$. Leader L_e could have entered epoch $e-1$ Δ time before some honest nodes or Leader L_e could have entered epoch $e-1$ Δ time after some honest nodes. In any case, all honest nodes must have received valid secret shares and sharing block SB within 3Δ after entering epoch $e-1$. Thus, no honest nodes send blame in epoch $e-1$ and will receive no blame messages from honest nodes within 6Δ time in epoch $e-1$ (i.e., until epoch-timer$_{e-1} > 5\Delta$).

Consider an honest node p_i. If node p_i receives no blame messages from Byzantine nodes, it will send an ack for sharing block SB to L_e. On the other hand, if node p_i receives up to t blame messages from Byzantine nodes, it forwards blame messages to L_e. Honest Leader L_e sends the shares corresponding to the blame messages to node p_i which node p_i receives within 8Δ in epoch $e-1$. Moreover, there is no equivocation from leader L_e. Thus, node p_i sends an ack for sharing block SB to L_e.

Thus, all honest nodes send ack for sharing block SB and leader L_e receives $t+1$ ack message for sharing block SB within 10Δ (L_e may start epoch $e-1$ Δ time before node p_i) in epoch $e-1$. This proves part (i) of the Lemma.

Since leader L_e proposes a valid proposal $(H(SB), \mathcal{AC}_e(SB))$ in epoch e, part(ii) follows immediately from Fact 4.6. □

Lemma 5.12 (Guaranteed Beacon Output)**.** *For any epoch $e \geq 1$, all the honest nodes output a new beacon output O_e.*

Proof. Due to the round-robin leader selection, the honest nodes propose in at least $n-t$ epochs out of n epochs. By Lemma 5.11, all honest nodes commit n new secret shares in every honest epoch and updates their queues after $t+1$ epochs. Thus, $\text{Dequeue}(Q(p_i)) \neq \perp \forall p_i \in \mathcal{P} \setminus \mathcal{P}_r$. where p_i is an honest node. From the proof of Theorem 5.10, all honest nodes have consistent queues and $\mathcal{P}_r$ in each epoch. At the end of each epoch, all honest nodes dequeue common secret shares and send homomorphic sums to all other nodes. Thus, honest nodes will have $t+1$ valid homomorphic sums and will output new beacon outputs in every epochs. □

Lemma 5.13 (Communication Complexity)**.** *Let $f \leq t$ be the number of actual Byzantine faults, κ be the size of accumulator and w be the size of witness. The amortized communication complexity of the protocol is $O(\kappa f n^2 + (\kappa + w)n^2)$ bits per epoch.*

Proof. In the Block validation protocol, distributing $O(\kappa n)$-sized commitment has a cost of $O(\kappa n^2)$ communication. Sending corresponding $O(\kappa n)$-sized secret shares and $O(wn)$-sized witness incur $O((\kappa + w)n^2)$ communication. Next, f Byzantine nodes can always blame even when the epoch leader is honest. Thus, an epoch leader needs to send $O(\kappa f n)$-sized secret shares while privately opening the secret shares. The nodes also forward privately opened secret secrets to nodes that blamed. This step has a cost of $O(\kappa f n^2)$ in an honest epoch. When the leader is Byzantine, it can create a scenario when up to t nodes send blame and hence, this step has $O(\kappa t n^2)$ cost. Out of n consecutive epochs, there can be at most f Byzantine epochs and $n-f$ honest epochs. Hence, this step has amortized complexity of $O(\kappa f n^2)$.

By Lemma 4.8, SMR protocol has a cost $O((\kappa + w)n^2)$ bits for input of size $O(\kappa n)$. The homomorphic sum of secret shares is κ and homomorphic sum of witness is w. Thus, all-to-all broadcast of homomorphic sums incurs $O((\kappa + w)n^2)$. Thus, the amortized communication complexity is $O(\kappa f n^2 + (\kappa + w)n^2)$ bits per epoch. □

Now, we prove the beacon properties for BRandPiper protocol. We first show that BRandPiper is a secure VSS protocol as per Definition 3.4. The proof is similar to the proof in Theorem 5.8, but we present it here for completeness.

THEOREM 5.14 (SECURE VSS). *Assuming a secure VSS scheme $\mathcal{VSS}$, the BRandPiper protocol is a secure verifiable secret sharing protocol with the dealer as the leader of an epoch, and the rest of the nodes as the verifiers.*

PROOF SKETCH. The view of an adversary $\mathcal{A}$ in BRandPiper is the same as the view of an adversary running one instance of iVSS assuming a bulletin board. Therefore, an adversary that can break the secrecy property in BRandPiper protocol can also break the secrecy in iVSS, which in turn can break the secrecy property from $\mathcal{VSS}$ (Theorem 5.8). The commitment property has an additional failure probability arising from the case where the adversary can forge $t + 1$ signatures which occurs with $\mathsf{negl}(\kappa)$ probability.

Formally, we prove the security of VSS by proving the individual properties:
Secrecy: For an honest leader L_e of epoch e, no honest node will blame, and therefore an adversary $\mathcal{A}$ will only learn the t shares of its own corruption, and not learn any new share by blaming. Therefore the probability of $\mathcal{A}$ of violating the secrecy property is $\mathsf{negl}(\kappa)$ from the underlying $\mathcal{VSS}$ scheme, since the views are identical to that of iVSS.
Correctness: For an honest leader L_e of epoch e, from Lemma 5.11, all the honest nodes commit the SB with shares for the secret. During the reconstruction for the beacon, every honest node $p_i \in \mathcal{P}$ use the same share for $SV_{L_e,i}$ with a high probability of $1-\mathsf{negl}(\kappa)$. A Byzantine node $p_j \in \mathcal{P}$ cannot provide a valid witness $\mathcal{VSS}.\pi_{L_e,j}$ for an incorrect share with probability better than $\mathsf{negl}(\kappa)$, thereby ensuring that the correctness property is maintained.
Commitment: If an honest node commits a valid block SB from a byzantine leader L_e in some epoch e, then all honest nodes commit SB, from the SMR property in Theorem 4.5. Therefore during reconstruction, a Byzantine node $p_j \in \mathcal{P}$ cannot provide incorrect shares as it cannot generate a valid witness $\mathcal{VSS}.\pi_{L_e,j}$ (except with $\mathsf{negl}(\kappa)$ probability). If a Byzantine leader does not propose any block, then all honest nodes agree on $\perp$, thereby ensuring the commitment property. □

Concrete Instantiations. Consider instantiating $\mathcal{VSS}$ using the Pedersen commitment based VSS [32] using the polynomial commitment scheme. This scheme is identical to the Pedersen VSS [38] which is an information-theoretic VSS scheme except that the polynomial commitment based on q-SDH is used. The polynomial commitment scheme however is identical to the Pedersen commitment and is unconditionally hiding. Since our SMR is adaptively secure, and our VSS scheme is unconditionally hiding, BRandPiper is also adaptively secure. For the binding part, as shown in [32], one can show a reduction to an adversary violating the binding property to an adversary violating the q-SDH assumption.

Next, we show that BRandPiper satisfies the bias-resistance requirement from Definition 3.3.

Lemma 5.15 (Bias-resistance). *For any epoch $e \geq 1$, the probability that a t-bounded adversary $\mathcal{A}$ can fix any c bits of the BRandPiper beacon output O_e is* $\mathsf{negl}(c) + \mathsf{negl}(\kappa)$.

PROOF SKETCH. The output in any epoch e is $O_e \leftarrow H(R_e)$, where R_e is the homomorphic sum of secrets from at least $t + 1$ honest nodes. From the *secrecy* guarantee in Theorem 5.14, we know that no adversary $\mathcal{A}$ can predict the value of these honest nodes until reconstruction with probability better than $\mathsf{negl}(\kappa)$. At the same time, no adversary $\mathcal{A}$ can change the committed value for any p_i during reconstruction due to the *commitment* guarantee from Theorem 5.14 with probability better than $\mathsf{negl}(\kappa)$. Therefore, a t-bounded adversary cannot do better than guessing whose probability is $\mathsf{negl}(c) + \mathsf{negl}(\kappa)$ to fix c bits in the output O_e for any epoch $e \geq 1$. □

Lemma 5.16 (Rushing Adversary Advantage). *For any epoch e, a rushing adversary can reconstruct output O_e at most 2Δ time before honest nodes.*

The proof remains identical to Lemma 5.3.

Next, we show that BRandPiper satisfies the 1-absolute unpredictability requirement from Definition 3.3.

Lemma 5.17 (BRandPiper 1-absolute unpredictability). *The BRandPiper random beacon protocol is an 1-absolute unpredictable random beacon.*

PROOF SKETCH. Since our SMR protocol is adaptively secure and our protocol is as secure as $\mathcal{VSS}$, we can instantiate $\mathcal{VSS}$ with Pedersen VSS which is information theoretically secure but at the cost of communication complexity to prove adaptive security of BRandPiper. By instantiating $\mathcal{VSS}$ with eVSS [32], we do not know how to show adaptive security using existing proof techniques. However, no adaptive attacks against eVSS are known either.

Let τ be some time at which all honest nodes are in an epoch $e \geq 1$. We show that an adversary $\mathcal{A}$ cannot predict O_{e+1}. Due to the secrecy property in Theorem 5.14 and the fact that the beacon output O_{e+1} is derived from the reconstruction of R_{e+1}, which is a homomorphic sum of inputs from at least $n - t > t$ nodes, any adversary $\mathcal{A}$ cannot predict O_{e+1}. The values from the honest nodes are guaranteed to be truly random (by definition). Therefore, the output O_{e+1} is unpredictable for an adversary $\mathcal{A}$.

An adversary $\mathcal{A}$ can get a 1 epoch advantage since there can exists times τ where some honest nodes are in epoch e and others are in epoch $e - 1$. At this point, a rushing adversary knows the output O_e before the honest nodes. □

THEOREM 5.18 (BRANDPIPER SECURE RANDOM BEACON). *BRandPiper protocol is a 1-secure random beacon.*

PROOF. The proofs follow trivially from Lemma 5.15, Lemma 5.17, and Lemma 5.12. □

6 CLOCK SYNCHRONIZATION FOR NEW NODES

In this section, we present a clock synchronization protocol to synchronize some additional nodes when majority of honest nodes are synchronized. Such a synchronization is useful during reconfiguration when a new node joins the system. Prior known clock synchronization protocol [3] can be used to synchronize all nodes with a communication cost of $O(\kappa n^3)$ without threshold signatures.

Let clock synchronization protocol start in epoch e. Node $p_i \in \mathcal{P}$ performs the following:

1. **Share secrets.** Leaders $\{L_e, \dots, L_{e+t}\}$ use block validation (refer Figure 8) and the BFT protocol to commit secrets $\{s_e, \dots, s_{e+t}\}$ respectively. e.g., Leader L_e uses the block validation protocol while in epoch $e - 1$ to share a single secret s_e chosen uniformly at random and collect an ack certificate $\mathcal{AC}(SB)$ on the commitment SB for secret s_e. In epoch e, Leader L_e proposes block $B_k := (H(SB), \mathcal{AC}(SB))$.
2. **Reconstruct.** When epoch-timer$_{e+2t}$ expires, perform the following:
   (a) Build homomorphic sum share SV_i, witness $\mathcal{VSS}.\pi_i$, and commitment $\mathcal{VSS}.\mathsf{C}_e$ using secret shares for $\{s_e, \dots, s_{e+t}\}$. Send SV_i and $\mathcal{VSS}.\pi_i$ to all the nodes.
   (b) Upon receiving share SV_j and witness $\mathcal{VSS}.\pi_j$ for $\mathcal{VSS}.\mathsf{C}_e$, ensure that $\mathcal{VSS}.\mathsf{ShVrfy}(SV_j, \mathcal{VSS}.\pi_j, \mathcal{VSS}.\mathsf{C}_e) = 1$.
   (c) Upon receiving $t + 1$ valid homomorphic sum shares in SV, obtain $R_e \leftarrow \mathcal{VSS}.\mathsf{Recon}(SV)$.
3. **Synchronize.** The first time node p_i receives a valid homomorphic secret R_e either through reconstruction or on receiving sync message from other nodes, it
   - resets its epoch-timer$_{e+2t+1}$ to the beginning of epoch $e + 2t + 1$.
   - broadcasts a sync message containing R_e to all other nodes

Figure 10: Clock synchronization protocol

This holds true even when synchronizing a single node in the system where a majority of nodes are already synchronized.

Our clock synchronization protocol to add new nodes (refer Figure 10) makes use of VSS secret sharing scheme presented in Section 5.3.2. Our approach requires a total communication complexity of $O(\kappa n^3)$; however, this can be split over $O(n)$ iterations with $O(\kappa n^2)$ communication. This will be useful in our beacon protocol to maintain quadratic communication complexity in each round. Our protocol uses the fact that $O(t)$ secret shares can be homomorphically combined to a single secret share of size $O(1)$ and $t + 1$ homomorphic secret shares can be be opened to get a $O(\kappa)$ sized secret. The opened secret can be broadcast among all nodes to synchronize the clocks of all honest nodes including the new joining node within Δ time from each other.

The first honest node to reset epoch-timer for some epoch e will broadcast sync message containing R_e which makes all other honest parties reset their epoch-timer$_e$ within Δ time. Observe that since the size of homomorphic R_e is $O(\kappa)$ bits, an all-to-all broadcast has a cost of $O(\kappa n^2)$ bits.

7 RECONFIGURATION

In this section, we present reconfiguration mechanisms for our beacon protocols to restore the resilience of the protocol after removing some Byzantine nodes. We make following modification to the protocols. Each node maintains a variable n_t that records the number of additional nodes that can been added to the system. Variable n_t is incremented each time a Byzantine node is added to set $\mathcal{P}_r$ and is decreased by one when a new node joins the system. The value of n_t can be at most t.

The generic reconfiguration protocol is presented in Figure 11. The reconfiguration protocol applies to both beacon protocols. Later, we make appropriate modifications for each beacon protocols.

Lemma 7.1 (Liveness). *If $n_t > 0$ at some epoch e^* and there are new nodes intending to join the system in epochs $\geq e^*$, then eventually a new node will be added to the system.*

Proof. Suppose $n_t > 0$ and a new node p_i intends to join the system. Suppose for the sake of contradiction, no new node including p_i is added to the system. However, since node p_i intends to join the system, it must have sent inquire requests to all nodes in the system and at least $t + 1$ honest nodes will respond to the inquire request since $n_t > 0$ at the end of some epoch $e' \geq e^*$.

Let node p_i send join request along with an inquire certificate and nodes receive the request in epoch $e \geq e'$. The first honest leader $L_{e''}$ of epoch $e'' \geq e$ will include the join request in its block proposal if no new node has been added to the system since epoch e' and there does not exist any block proposal with a join request in the last $t + 1$ epochs in its highest ranked chain and by Fact 4.6, the block proposal with join request will be committed. A contradiction.

If some node has already been added to the system since epoch e', this trivially satisfies the statement and we obtain a contradiction. If there exists a block proposal B_h with a join request for some node p_k in last t epochs in the highest ranked chain for $L_{e''}$, B_h will be committed since honest node $L_{e''}$ extends it. The lemma holds and we obtain a contradiction.

□

7.1 Reconfiguration for GRandPiper

Node p_k generates a PVSS sharing $(\mathcal{PVSS}.\vec{S}, \mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi) \leftarrow \mathcal{PVSS}.\mathsf{ShGen}(R)$ of a random value chosen from the input space of $\mathcal{PVSS}$ for nodes $\mathcal{P} \cup \{p_k\} \setminus \mathcal{P}_r$. During the join phase in the reconfiguration protocol (refer Figure 11), it sends a join request to all nodes $\mathcal{P} \setminus \mathcal{P}_r$ with entity $\mathcal{M}$ set to the above PVSS sharings. In addition, all nodes update $Q(p_k)$ with the PVSS sharings provided by node p_k once the join request from node p_k gets committed.

We note that the adaptive adversary can corrupt any node as long as $t + 1$ honest nodes have full queue $Q(p_j) \neq \bot \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$, i.e., if the adversary already corrupted t nodes some of which are removed, the adversary can corrupt old honest nodes only when node p_k has full queue (i.e., $Q(p_j) \neq \bot \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$). This happens when all nodes in $\mathcal{P} \setminus \mathcal{P}_r$ becomes a leader at least once after node p_k becomes a leader. Due to random leader election, the expected number of epochs required for all nodes to be selected as leaders is $n \sum_{i=0}^{n} \frac{1}{i} = \Theta(n \log n)$.

Remark. We note that our GRandPiper beacon protocol can employ a rotating leader election for our BFT SMR protocol with randomized leader election during reconstruction phase. With this

A new node p_k that intends to join the system uses following procedure to join the system.

(1) **Inquire.** Node p_k inquires all nodes in the system to send the set of active nodes, i.e., $\mathcal{P} \setminus \mathcal{P}_r$. Upon receiving the inquire request, an honest node p_i responds to the request only if $n_t > 0$. Node p_i sends $\mathcal{P} \setminus \mathcal{P}_r$ at the end of some epoch e' in which the inquire request was received. Node p_k waits for at least $t + 1$ consistent responses from the same epoch e' and forms an inquire certificate. An inquire certificate is valid if it contains $t + 1$ inquire responses that belong to the same epoch e' and contains the same set of active nodes.

(2) **Join.** Node p_k sends a join request to all nodes $\mathcal{P} \setminus \mathcal{P}_r$ along with the inquire certificate and an additional entity $\mathcal{M}$ specific to the beacon protocols.

(3) **Propose.** Upon receiving the join request, the leader L_e of current epoch e adds the join request containing inquire certificate and entity $\mathcal{M}$ in its block proposal B_k if (i) L_e does not observe a block proposal with a join request in last $t + 1$ epochs in its highest ranked chain and (ii) no new node has been added since epoch e'.

(4) **Update.** If the block B_k with the join request from node p_k proposed in epoch e gets committed by epoch $e + t$, update $n_t \leftarrow n_t - 1$ in epoch $e + t$ and send $\mathcal{P} \setminus \mathcal{P}_r$ to node p_k. Henceforth, node p_k becomes a *passive* node and receives all protocol messages from active nodes.

(5) **Synchronize.** Nodes execute clock synchronization protocol (refer Figure 10) from epoch $e + t + 2$ to synchronize node p_k. All nodes including node p_k are synchronized in epoch $e + 3t + 3$. At epoch $e + 3t + 3$ update $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_k\}$. Henceforth, node p_k becomes an *active* node and participates in the protocol. Additionally, node p_k participates in the reconstruction protocol only if it has required secret shares.

If node p_k fails to join the system, it restarts reconfiguration process again after some time.

Figure 11: Reconfiguration protocol

change, the adaptive resilience of the beacon protocol is restored in $n + t + 1$ epochs compared to expected $n \log n$ epochs.

Theorem 7.2. *GRandPiper protocol maintains safety and liveness after reconfiguration.*

Proof. Let node p_i be the new joining node. GRandPiper protocol is safe and live before reconfiguration. Since we assume the adversary can corrupt a new node as long as $t + 1$ honest nodes have full queue, i.e., $Q(p_j) \neq \perp \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$, there will always be $t + 1$ honest nodes with correct secret shares. Hence, the protocol maintains safety and liveness after reconfiguration. □

7.2 Reconfiguration for BRandPiper

Node p_k that intends to join the system uses the reconfiguration protocol (refer Figure 11) to join the system. During the join phase, node p_k does not need to send any additional commitment i.e., sets $\mathcal{M} := \perp$. Once node p_k becomes the active node, it is then allowed to become a leader using round-robin leader election and shares VSS commitments to n secrets when it becomes the leader. All active nodes uses secret shares for node p_k only when they have committed the commitment shared by node p_k.

Like the reconfiguration for GRandPiper protocol, the adaptive adversary can corrupt any node as long as $t + 1$ honest nodes have full queue $Q(p_j) \neq \perp \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$. I.e., if the adversary already corrupted t nodes some of which are removed, the adversary can corrupt old honest nodes only when node p_k has full queue (i.e., $Q(p_j) \neq \perp \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$). This happens when all nodes in $\mathcal{P} \setminus \mathcal{P}_r$ becomes a leader at least once after node p_k becomes a leader. Due to the round-robin leader election, node p_k will have full queue after $n + t + 1$ epochs after it has become an active node.

Theorem 7.3. *BRandPiper protocol maintains safety and liveness after reconfiguration.*

The proof remains identical to Theorem 7.2.

8 PERFORMANCE EVALUATION

In this section, we evaluate the performance of RandPiper, and compare it with the performance of related works.

Implementation. We implement BRandPiper protocol in Rust, due to its strong support for correctness in concurrency, and compile-time memory safety guarantees. Our implementation is lock-free and uses message passing to ensure efficiency. We provide the setup parameters for every node in the config files. Our code is event driven, and reacts to various timeouts and messages from the network. We use ED25519 for digital signatures. We use the BLS-12-381 [11] curve and use implementations of SCRAPE [18] over this curve as the PVSS scheme, Pedersen-based eVSS and Polycommit [32] over this curve as the VSS and bilinear accumulator scheme.

Optimizations. We perform some system-level optimizations. In particular, we do the following: (1) We generate random shares before the propose step (this can be done using extra cores, an external node supplying sharings) and use it during the propose step, so that the share generation does not block the critical path. (2) We take advantage of the tokio library [48] and futures in Rust to run concurrently without spawning threads. (3) We implement both the accumulator libraries: the bilinear accumulator as well as the Merkle tree accumulator. We observe that the computational performance of the Merkle tree accumulator is much better in practice in general.

Setup. All our experiments were conducted on t2.micro AWS instances located at *ohio* data center, which have 1 GB RAM, 8 GB hard disk, 1 vCPU running at up to 3.3 GHz. The advertised bandwidth for these instances is 60-80 MBits/s.

Baselines. We compare the performance of our implementation with two baselines which are the current state-of-the-art public implementations: drand [25] and HydRand [41, 43]. We choose drand because it is a practically deployed system implementing Cachin *et*

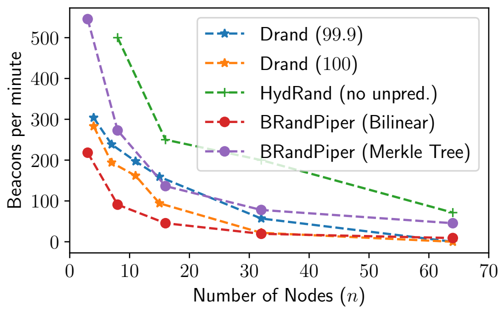


Figure 12: Overview of the beacons produced per minute for the various protocols. Among the compared implementations, HydRand is only bias-resistant but does not offer 1-absolute unpredictability

al. [17] and evaluating our performance against it justifies our practicality. We choose HydRand as our second baseline because, it is theoretically related to our work: HydRand requires 2/3 honest majority as compared to the optimal 1/2 honest majority necessary for us. Note that the basic HydRand protocol and implementation [41] only offers bias-resistance but no unpredictability: an adversary may correctly predict a random beacon in $t + 1$ epochs in advance.
Micro-Benchmarks. We measure the efficiency of the primitives used in our protocol. Concretely, we measure the run times for (1) accumulator share generation, verification and reconstruction for the bilinear accumulator as well as the Merkle tree accumulator, (2) PVSS share generation, verification, and reconstruction, (3) eVSS share generation, verification, and reconstruction, and (4) the size of the various messages used in the protocol. The details are provided in Appendix B. We observe that the Merkle tree accumulator for our small scales has smaller message sizes and very efficient run times. We also observe that eVSS operations in general perform much better than its PVSS counterparts.

In this work, we build communication-efficient random beacon protocols with comparable or better performance than the state of the art solutions. Thus, the key metric we compare is the number of beacon values that can be produced in a minute. In addition, compared to Drand, we have the advantage of reconfigurability and weaker network assumptions, and compared to HydRand, we can tolerate more faults. The methodology is to run protocols at appropriate values of Δ, which in turn depends on the computation and communication costs.

BRandPiper. Our beacon produces a value every 11Δ. We measure the smallest value of Δ for which our beacon produces outputs without any of the n correct nodes blaming/reporting malicious behaviour in the logs. Using this, we measure the metric: number of beacons produced per minute for BRandPiper protocol, and present them in Figure 12. We run our beacon for 100 rounds, thereby giving a strong confidence that the Δ used is viable, provided the network conditions stay the same.

Drand. For Drand, we measure the parameter *time discrepancy*, which is a value output by the drand daemon by every node that reports the time (in ms) between obtaining the beacon value and the time at which the epoch started. This time accounts for the synchronization losses, network delays as well as the computations. The beacon continues to produce values every *period* seconds. However, the time discrepancy parameter defines the lowest period we can set to ensure continuous beacon output. We give the benefit of the doubt to drand here as there is no guarantee that setting such low values for Δ does not overwhelm the system. We allow the beacon to run for 100 epochs, and measure the 99.9^{th} percentile of the time discrepancies observed in the logs of all the nodes over all the epochs, and present their growth in Figure 12. For 100 percentile, we observe that we are always better than drand. In our experiments, when $n = 65$, we observed that the DKG initialization in drand results in all the nodes aborting, even after setting large values for period.

HydRand. We use the public implementation [41]. It consists of three rounds: propose, acknowledge and vote, timeouts for which can be configured. We find the smallest such configuration that allows the system to work and report the numbers in Figure 12. HydRand on its own offers $t + 1$-absolute unpredictability where $t < n/3$ along with the unbiasability property. However, drand and BRandPiper are both 1-absolute unpredictable, and in that sense, HydRand is not unpredictable.

From Figure 12, we can clearly see that the Merkle tree based BRandPiper implementation is quantitatively as practical as the state of the art practical random beacon protocol: *drand.* Drand uses a leader to co-ordinate the DKG and reconfiguration protocols. There is no description on how to recover if the leader was Byzantine. Additionally, in drand, the synchronization for the reconfigured instance is via the co-ordinator (the leader). It is not clear how the protocol will recover if the leader sends start messages differently to different nodes in the system. Therefore, qualitatively, we use much clearer and formal network assumptions and allow efficient and secure reconfiguration, including synchronization for the incoming nodes, without pausing the protocol, unlike drand, and therefore can conclude that BRandPiper protocol is not just theoretically interesting, but practical as well.

REFERENCES

[1] [n.d.]. blockchain oracle service, enabling data-rich smart contracts. https://provable.xyz/
[2] [n.d.]. Generate Random Numbers for Smart Contracts using Chainlink VRF. https://docs.chain.link/docs/chainlink-vrf
[3] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. 2019. Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. *Financial Cryptography and Data Security (FC)* (2019).
[4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2018. Dfinity Consensus, Explored. *IACR Cryptol. ePrint Arch.* 2018 (2018), 1153.
[5] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. [n.d.]. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy (SP)*. 654–667.
[6] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 337–346.
[7] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2020. Optimal good-case latency for byzantine broadcast and state machine replication. *arXiv preprint arXiv:2003.13155* (2020).
[8] Michael Backes, Aniket Kate, and Arpita Patra. 2011. Computational verifiable secret sharing revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 590–609.
[9] Shehar Bano, Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, et al. 2020. State machine replication in the Libra Blockchain. *Avalaible at: https://developers. libra. org/docs/state-machine-replication-paper (Consulted on December 19, 2020)* (2020).

[10] Niko Barić and Birgit Pfitzmann. 1997. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*. Springer, 480–494.

[11] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. 2002. Constructing Elliptic Curves with Prescribed Embedding Degrees. Cryptology ePrint Archive, Report 2002/088. https://eprint.iacr.org/2002/088.

[12] Bheisler. [n.d.]. bheisler/criterion.rs. https://github.com/bheisler/criterion.rs

[13] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.

[14] Dan Boneh and Xavier Boyen. 2008. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *J. Cryptol.* 21, 2 (2008), 149–177.

[15] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation.

[16] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

[17] 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.

[18] 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.

[19] TH Hubert Chan, Rafael Pass, and Elaine Shi. 2018. PiLi: An Extremely Simple Synchronous Blockchain. (2018).

[20] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. 2019. Homomorphic Encryption Random Beacon. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1320.

[21] Information Technology Laboratory Computer Security Division. [n.d.]. Interoperable Randomness Beacons: CSRC. https://csrc.nist.gov/projects/interoperable-randomness-beacons

[22] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.

[23] Danny Dolev and H. Raymond Strong. 1983. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666.

[24] J Drake. [n.d.]. Minimal VDF randomness beacon. Ethereum Research Post (2018).

[25] Drand. [n.d.]. Drand - A Distributed Randomness Beacon Daemon. https://github.com/drand/drand

[26] Pesech Feldman and Silvio Micali. 1997. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM J. Comput.* 26, 4 (1997), 873–933.

[27] 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.

[28] Mads Haahr. [n.d.]. True Random Number Service. https://www.random.org/

[29] Runchao Han, Jiangshan Yu, and Haoyu Lin. [n.d.]. RANDCHAIN: Decentralised Randomness Beacon from Sequential Proof-of-Work. ([n. d.]).

[30] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548* (2018).

[31] 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.

[32] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*. Springer, 177–194.

[33] Jonathan Katz and Chiu-Yuen Koo. 2006. On expected constant-round protocols for Byzantine agreement. In *Annual International Cryptology Conference*. Springer, 445–462.

[34] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. [n.d.]. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). 2111–2128.

[35] Atsuki Momose and Ling Ren. 2020. Optimal Communication Complexity of Byzantine Consensus under Honest Majority. *arXiv preprint arXiv:2007.13175* (2020).

[36] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. 2020. Improved Extension Protocols for Byzantine Broadcast and Agreement. *arXiv preprint arXiv:2002.11321* (2020).

[37] Lan Nguyen. 2005. Accumulators from bilinear pairings and applications. In *Cryptographers' track at the RSA conference*. Springer, 275–292.

[38] Torben Pryds Pedersen. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*. Springer, 129–140.

[39] Michael O Rabin. 1983. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 403–409.

[40] 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.

[41] Philipp Schindler. [n.d.]. HydRand. https://github.com/PhilippSchindler/hydrand

[42] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar Weippl. 2020. *RandRunner: Distributed Randomness from Trapdoor VDFs with Strong Uniqueness.* Technical Report. Cryptology ePrint Archive, Report 2020/942, https://eprint. iacr. org/2020/942.

[43] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. 2020. Hydrand: Practical continuous distributed randomness. In *2020 IEEE Symposium on Security and Privacy (SP). IEEE.*

[44] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[45] Serde-Rs. [n.d.]. serde-rs/serde. https://github.com/serde-rs/serde

[46] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. 2020. On the Optimality of Optimistic Responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 839–857.

[47] 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.

[48] Tokio-Rs. [n.d.]. tokio-rs/tokio. https://github.com/tokio-rs/tokio

[49] 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*. 347–356.

A EXTENDED PRELIMINARIES

In this section, we provide additional preliminaries used in our paper. The VSS interface is presented in Table 2 and the PVSS interface is presented in Table 3.

B MICRO BENCHMARKS

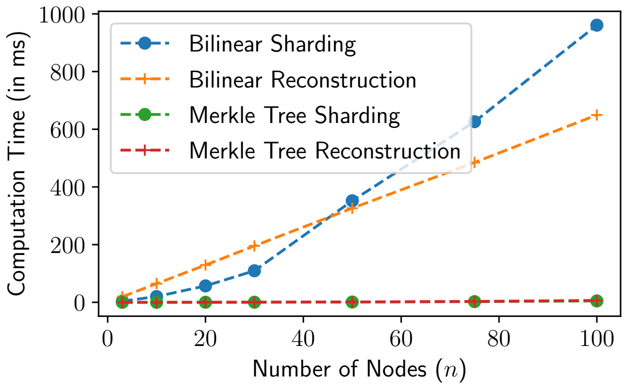


Figure 13: Performance comparison of the bilinear accumulator and the Merkle tree accumulator. We measure the performance to create and reconstruct shares for the propose message, one of the largest messages in our protocol.

Methodology. We use the criterion [12] benchmarking suite for performing benchmarks. It warms the CPU for 3 seconds with plain runs, and then finally benchmarks our code, computes a distribution over the runs, and returns the likely cost (time) of the benchmark after removing the outliers.

Accumulators. For this benchmark, we plot the time to create shares for messages typically used in our protocol for the value of n in the x-axis and the time in y-axis. We do not use parallelism here, so when we say verify n shares we mean verifying n witnesses serially.

Figure 13 shows the variation in the performance of the bilinear accumulator and the Merkle tree accumulator. We observe that the performance of the Merkle tree accumulator is significantly

Table 2: VSS scheme algorithm interface

Interface	Description
$\mathcal{VSS}.\mathsf{pp} \leftarrow \mathcal{VSS}.\mathsf{Setup}(\kappa, \mathsf{aux})$	Generate the scheme parameters $\mathcal{VSS}.\mathsf{pp}$. $\mathcal{VSS}.\mathsf{pp}$ is an implicit input to all other algorithms.
$(\mathcal{VSS}.\vec{S}, \mathcal{VSS}.\vec{W}, \mathcal{VSS}.\mathsf{C}) \leftarrow \mathcal{VSS}.\mathsf{ShGen}(s)$	Typically executed by the dealer L with secret s to generate secret shares $\mathcal{VSS}.\vec{S} := \{\mathcal{VSS}.\mathsf{s}_1, \ldots, \mathcal{VSS}.\mathsf{s}_n\}$, witnesses $\mathcal{VSS}.\vec{W} := \{\mathcal{VSS}.\pi_1, \ldots, \mathcal{VSS}.\pi_n\}$, and a constant size commitment $\mathcal{VSS}.\mathsf{C}$ to a degree $t+1$ polynomial.
$\{0,1\} \leftarrow \mathcal{VSS}.\mathsf{ShVrfy}(\mathcal{VSS}.\mathsf{s}, \mathcal{VSS}.\pi, \mathcal{VSS}.\mathsf{C})$	Verify if the share $\mathcal{VSS}.\mathsf{s}$ and witness $\mathcal{VSS}.\pi$ form a correct share for $\mathcal{VSS}.\mathsf{C}$. 0 indicates a failure and 1 indicates a success.
$s \leftarrow \mathcal{VSS}.\mathsf{Recon}(\mathcal{VSS}.\vec{S})$	Reconstruct the shared secret s from the collection of shares $\mathcal{VSS}.\vec{S} \subseteq \{\mathcal{VSS}.\mathsf{s}_1, \ldots, \mathcal{VSS}.\mathsf{s}_n\}^{t+1}$.
$\{0,1\} \leftarrow \mathcal{VSS}.\mathsf{ComVrfy}(\mathcal{VSS}.\mathsf{C}, s)$	Check if s is the correct opening for the commitment $\mathcal{VSS}.\mathsf{C}$. 0 indicates a failure, and 1 indicates a success.

Table 3: PVSS scheme algorithm interface

Interface	Description
$\mathcal{PVSS}.\mathsf{pp} \leftarrow \mathcal{PVSS}.\mathsf{Setup}(\kappa, \mathsf{aux})$	Generate the scheme parameters $\mathcal{PVSS}.\mathsf{pp}$. $\mathcal{PVSS}.\mathsf{pp}$ is an implicit input to all other algorithms.
$(\mathcal{PVSS}.\mathsf{pk}, \mathcal{PVSS}.\mathsf{sk}) \leftarrow \mathcal{PVSS}.\mathsf{KGen}(\kappa)$	Generate PVSS key-pair $(\mathcal{PVSS}.\mathsf{pk}, \mathcal{PVSS}.\mathsf{sk})$ used for share encryption and decryption
$c \leftarrow \mathcal{PVSS}.\mathsf{Enc}(\mathcal{PVSS}.\mathsf{pk}, m)$ $m \leftarrow \mathcal{PVSS}.\mathsf{Dec}(\mathcal{PVSS}.\mathsf{sk}, c)$	The encryption and decryption algorithms used to send shares to all, and obtain private share respectively. The invariant $\Pr[\mathcal{PVSS}.\mathsf{Dec}(\mathcal{PVSS}.\mathsf{Enc}(m)) = m] = 1$ must always hold true for all m in the message domain of $\mathcal{PVSS}.\mathsf{Enc}$. The decryption algorithm may also optionally output a proof of correct decryption.
$(\mathcal{PVSS}.\vec{S}, \mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi) \leftarrow \mathcal{PVSS}.\mathsf{ShGen}(s)$	Typically, executed by the dealer L with secret s to generate secret shares $\mathcal{PVSS}.\vec{S} := \{\mathcal{PVSS}.\mathsf{s}_1, \ldots, \mathcal{PVSS}.\mathsf{s}_n\}$ and encryptions of shares $\mathcal{PVSS}.\vec{E} := \{\mathcal{PVSS}.\mathsf{Enc}(\mathcal{PVSS}.\mathsf{s}_1), \ldots, \mathcal{PVSS}.\mathsf{Enc}(\mathcal{PVSS}.\mathsf{s}_n)\}$ for all nodes $\mathcal{P}$, and a cryptographic proof $\mathcal{PVSS}.\pi$ committing to s which guarantees any node with $> t$ shares reconstruct a unique s.
$\{0,1\} \leftarrow \mathcal{PVSS}.\mathsf{ShVrfy}(\mathcal{PVSS}.\vec{E}, \mathcal{PVSS}.\pi)$	Verify if the sharing is correct. A successful verification guarantees that all the encrypted shares are correct and that any $t+1$ nodes will reconstruct a unique s. 0 indicates a failure and 1 indicates a success.
$s \leftarrow \mathcal{PVSS}.\mathsf{Recon}(\mathcal{PVSS}.\vec{S})$	Reconstruct the shared secret s from the collection of shares $\mathcal{PVSS}.\vec{S} \subseteq \{\mathcal{PVSS}.\mathsf{s}_1, \ldots, \mathcal{PVSS}.\mathsf{s}_n\}^{t+1}$

better than that of the bilinear accumulator, and therefore in cloud settings it is efficient to choose a Merkle tree implementation. For high performance machines, or very tightly constrained networks, the bilinear accumulator provides a trade-off between the message size and computation.

Secret sharing. Figure 14 gives an overview of the performance of the various secret sharing schemes. We observe that the VSS scales much better than PVSS scheme for the same number of nodes. An interesting quirk we observe is that for eVSS, the costliest operation is the share generation as it involves generating shares for every node, witnesses and commitments. However, this step is not on the critical path; it can be performed one epoch before a proposal. The verification is the next costliest operation which again can proceed at its own pace without affecting the BFT for BRandPiper and only affects a delay in producing the beacon value.

Message sizes. We plot the message sizes of the various steps used in both variant of our implementation. The y-axis in Figure 15 and Figure 16 represents the size of the message after serialization.

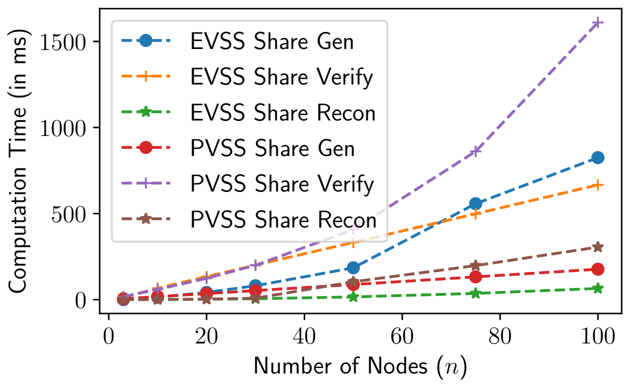


Figure 14: An overview of the performance of VSS and PVSS schemes. In general, we observe that the VSS scheme (eVSS) performs better than the PVSS schemes.

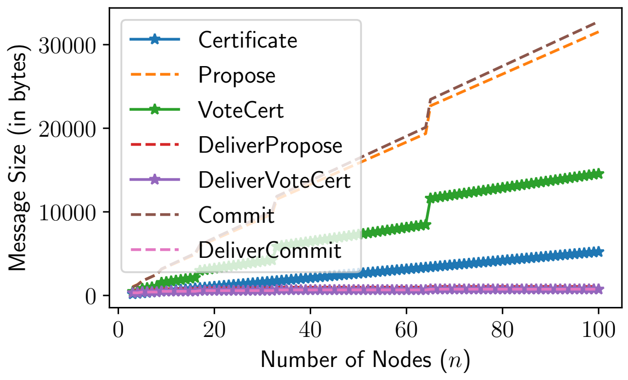


Figure 15: An overview of the message sizes of the various messages in our Merkle tree based implementation of BRandPiper. The sharp rise observed for the deliver messages is due to the $\log$ **factor.**

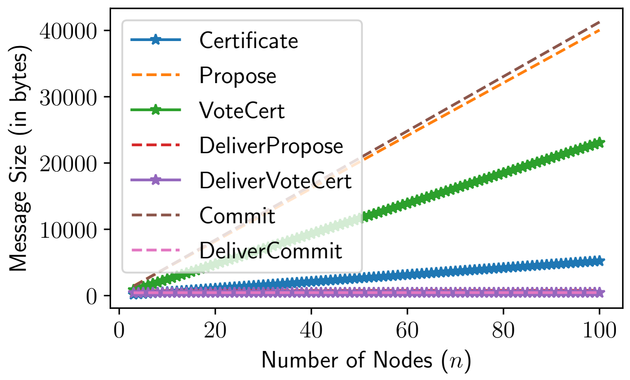


Figure 16: An overview of the message sizes of the various messages in our *bilinear accumulator* based implementation of BRandPiper.

We use Rust's in built serde [45] mechanism to derive an efficient binary encoding/decoding scheme for our data. For instance, The *Ack* message is 56 bytes,a *Vote* message is 56 bytes, and *Reconstruct* is 189 bytes *after serialization* in both our implementations.