# OptRand: Optimistically Responsive Reconfigurable Distributed Randomness

Adithya Bhat*
abhatk@purdue.edu
Purdue University

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

Aniket Kate
aniket@purdue.edu
Purdue University / Supra

Kartik Nayak
kartik@cs.duke.edu
Duke University

***Abstract*—Public random beacons publish random numbers at regular intervals, which anyone can obtain and verify. The design of public distributed random beacons has been an exciting research direction with significant implications for blockchains, voting, and beyond. Distributed random beacons, in addition to being bias-resistant and unpredictable, also need to have low communication overhead and latency, high resilience to faults, and ease of reconfigurability. Existing synchronous random beacon protocols sacrifice one or more of these properties.**

**In this work, we design an efficient unpredictable synchronous random beacon protocol, OptRand, with quadratic (in the number $n$ of system nodes) communication complexity per beacon output. First, we innovate by employing a novel combination of bilinear pairing based publicly verifiable secret-sharing and non-interactive zero-knowledge proofs to build a linear (in $n$) sized publicly verifiable random sharing. Second, we develop a state machine replication protocol with linear-sized inputs that is also optimistically responsive, i.e., it can progress responsively at actual network speed during optimistic conditions, despite the synchrony assumption, and thus incur low latency. In addition, we present an efficient reconfiguration mechanism for OptRand that allows nodes to leave and join the system. Our experiments show our protocols perform significantly better compared to state-of-the-art protocols under optimistic conditions and on par with state-of-the-art protocols in the normal case. We are also the first to implement a reconfiguration mechanism for distributed beacons and demonstrate that our protocol continues to be live during reconfigurations.**

## I. Introduction

The use of public random numbers is fundamental to many secure privacy-preserving systems where protocol parties have tamper-proof access to these random values (or common coins). Voting, lotteries, blockchains, and financial services depend on public randomness, and generating public randomness [47] has been an active area of research for the last four decades [55], [26], [49], [50], [14], [35], [34], [21], [33], [46], [18], [10]. Among these efforts, NIST's randomness beacons project [22] and Drand organization's beacon [28] have emerged in the last few years as real-world systems towards catering to this need for randomness beacons.

Informally, these systems offer random beacons, which are regular outputs of bias-resistant and unpredictable public random numbers. Bias-resistance ensures that the adversary cannot affect any future beacon value, say, for instance, affect the beacon to win a lottery, and unpredictability ensures that the adversary cannot predict any future beacon value, say, for example, bet on a favorable number in a lottery.

Distributing trust across multiple nodes such that only a minority of those can be compromised allows us to mitigate single-point-of-failures. Here, a distributed coin-tossing protocol combines randomness from multiple nodes to generate random beacons, which has been explored both theoretically [29], [15], [14], [32] and on the systems front [55], [50], [28], [10], [49], [35].

A common approach to designing random beacons involves a set of $n$ nodes each sharing a random value such that a random value is computed by combining a subset of these individual values. To share the value, protocols typically require every node (called a dealer or leader) to use a verifiable secret sharing (VSS) scheme where the node commits to a value with the guarantee that if it is successful, it can be reconstructed by honest nodes even if the dealer is malicious (Byzantine). By utilizing at least $t+1$ such dealers, where $t$ is the maximum number of compromised nodes, we are guaranteed that the reconstructed value is uniformly random since the contribution from an honest node is uniformly random.

There are several ideal properties that protocols in the literature have aimed to optimize. In an $n$-node system tolerating $t$ Byzantine faults, a secure coin-tossing protocol should aim for (i) bias-resistance, (ii) unpredictability, (iii) optimal resilience, (iv) low latency, (iv) high scalability, and (v) friendliness towards reconfiguration (allowing efficient addition and removal of nodes). In this work, we focus on the synchronous network setting where messages sent by the sender will arrive at the receiver within a known bounded delay $\Delta$. Synchronous protocols have the advantage of tolerating up to a minority corruption. While a myriad of random beacon protocols [14], [55], [50], [10], [49] have been proposed in this setting, existing solutions fall short in one or more of these directions. For example, Cachin et al. [14] is efficient in terms of communication complexity with $O(\kappa n^2)$, but their distributed (cryptographic) setup makes its expensive in term of reconfiguration especially as $n$ increases. Randrunner and similar approaches instead employ the use of verifiable delay functions (VDFs) frequently to generate beacons and possibly tolerate dishonest majority of faults. However, VDFs are computationally expensive and these protocols cannot offer low

*Contributed equally and listed alphabetically

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24832
www.ndss-symposium.org

TABLE I: **Comparison of related works on Random Beacon protocols.**

| Protocol | Net. | Res. | Comm. Compl. | | Unpred. | Reusable Setup | Resp. | Crypto Assumption* | Setup Assumption† |
|---|---|---|---|---|---|---|---|---|---|
| | | | Best | Worst | | | | | |
| Cachin et al./Drand [14], [28] | sync | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | 1 | ✗ | ✗ | Uniq. Sig, CDH | SRS |
| Dfinity [35], [2] | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^3)^{\ddagger}$ | $O(\kappa)$ | ✗ | ✗ | Uniq. Sig, CDH | SRS |
| HERB [21] | sync. | 33% | $O(\kappa n^3)$ | $O(\kappa n^3)$ | 1 | ✗ | ✗ | DDH | SRS |
| RandHerd [55] | sync. | 33% | $O(\kappa c^2 \log n)^{\S}$ | $O(\kappa n^4)$ | $O(\kappa)$ | ✗ | ✗ | DL | SRS |
| RandChain [34] | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | $O(\kappa)$ | ✗ | ✗ | VDF, PoW | SRS |
| RandHound [55] | sync. | 33% | $O(\kappa c^2 n)^{\P}$ | $O(\kappa c^2 n^2)^{\P}$ | 1 | ✓ | ✗ | DL | SRS |
| RandShare [55] | async. | 33% | $O(\kappa n^3)$ | $O(\kappa n^4)$ | 1 | ✓ | ✓ | DL | SRS |
| RandRunner [49] | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | $O(\kappa)$ | ✓ | ✗ | tVDF, DL | SRS |
| HydRand [50] | sync. | 33% | $O(\kappa n^2)$ | $O(\kappa n^3)$ | $\min(\kappa, t+1)$ | ✓ | ✗ | DDH | CRS |
| SPURT [23] | psync. | 33% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | 1 | ✓ | ✓ | DBS | CRS |
| GULL [17] | sync. | 50%$^{\|}$ | $O(\kappa n^3)$ | $O(\kappa n^3)$ | 1 | ✗ | ✗ | DDH | CRS |
| STROBE [7] | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | 1 | ✗ | ✓ | RSA, DL | SRS |
| GRandPiper [10] | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | $\min(\kappa, t+1)$ | ✓ | ✗ | q-SDH, SXDH | SRS |
| BRandPiper [10] | sync. | 50% | $O(\kappa n^2)^{\|}$ | $O(\kappa n^3)$ | 1 | ✓ | ✗ | q-SDH, SXDH | SRS |
| **OptRand** | sync. | 50% | $O(\kappa n^2)$ | $O(\kappa n^2)$ | 1 | ✓ | ✓** | q-SDH, SXDH | SRS |

**Net.** refers to the network assumption. **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 a rushing adversary can predict. **Reusable Setup** refers to a setup that can be reused when a node is replaced in the system. **Resp.** refers to responsiveness, i.e., if commit latency is a function of the network speed $\delta$. *All of these protocols assume Random oracles. †All of these protocols assume Public Key Infrastructure (PKI). ‡probabilistically $O(\kappa n^3)$ when $\Theta(n)$ consecutive leaders are bad. §$c$ is the average (constant) size of the groups of server nodes. ¶$c$ is is a client specified parameter to obtain client-specific randomness. ‖in conditions where the actual number of faults $f = O(1) \leq t$. **optimistically responsive during optimistic conditions.

latency [26], [49]. HydRand [50], with its use of publicly verifiable secret sharing (PVSS) [16] can be made reconfiguration-friendly while simultaneously achieving quadratic communication complexity in the best case; however, it works in a synchronous network and its resilience to faults, $t < n/3$, is sub-optimal. Finally, Randpiper set of protocols [10], while offering quadratic communication complexity in the best case and reconfiguration-friendliness, is communication-inefficient in the presence of faults and outputs a beacon value every $11\Delta$ time.

A natural question is whether we can achieve all of the above mentioned desirable properties simultaneously. Our work, OptRand, answers this question affirmatively. In particular, OptRand is a bias-resistant and unpredictable random beacon, with an $O(\kappa n^2)$ communication complexity and tolerating one-half Byzantine faults in a synchronous network. In fact, under optimistic conditions when the number of faults are $< n/4$ and a "leader" is honest, the protocol is responsive, i.e., it advances at the speed of the network, thus achieving low latency. Compared to the state-of-the-art in the synchronous setting, RandPiper [10], this protocol has better communication complexity (always $O(\kappa n^2)$) and optimistically-responsive latency.

### *A. Our Approach, Key Ideas and Results*

Our protocol is a novel combination of aggregatable PVSS and a state machine replication (SMR) to achieve an optimistically responsive random beacon protocol. We elaborate on the key features of OptRand and our approach to achieving them.

**Towards an always quadratic communication random beacon protocol.** An approach to obtaining random values is to combine random secrets from more than $t$ nodes, at least one of whom is honest. Thus, every node commits to its secrets using a publicly verifiable secret sharing (PVSS) scheme [16], which, naïvely, requires sharing $O(\kappa n)$-sized information per node with every other node, making the communication complexity at least cubic. Thus, the crux of the challenge is to route this information through a "leader" such that the (i) communication complexity is quadratic while (ii) still allowing nodes to verify the correctness of the shares (iii) even when the number of faults is $t < n/2$. We achieve this using the aggregation property of PVSS [16] employing bilinear pairings: in particular, unlike non-interactive zero-knowledge ($\Sigma$ protocol) proofs, pairing-based aggregate verification can be an efficient local process.

While the approach of routing the PVSS instance through the leader has been considered recently [23] to tolerate $t < n/3$ faults, the higher resilience of $t < n/2$ forces us to solve the problem differently. For instance, SPURT [23] can tolerate $t$ honest nodes from never receiving a share. However, when $n = 2t + 1$, this is not-acceptable. In particular, to deal with Byzantine leaders efficiently, similar to [10], we take a two-pronged approach: (i) we employ pipelined state machine replication (SMR) that piggybacks consecutive consensus instances, and (ii) we buffer secrets shared using PVSS for each node, and reconstruct the last shared secret/beacon for the Byzantine leader before removing it from subsequent proposals.

**Towards optimistically responsive random beacons.** While using the above discussed cryptographic approach and substituting it in RandPiper can offer us always quadratic communication complexity, RandPiper [10] still requires $11\Delta$ latency in each epoch (a duration coordinated by a distinct leader) for beacons – even under optimistic conditions, their beacon protocol cannot progress at the network speed.

The key challenges to obtain responsiveness are (i) efficient propagation of large messages, and (ii) efficient synchronization of all the nodes when some nodes move to the next

epoch. RandPiper [10] uses erasure coding and cryptographic accumulators along with waiting for $\Omega(\Delta)$ time to check for possible misbehavior from the current leader to efficiently propagate large messages. In OptRand, we design a new technique to efficiently propagate large messages without checking for misbehavior from the current leader; hence, do not require $\Omega(\Delta)$ wait to ensure propagation.

In synchronous protocols, synchronization refers to all the nodes starting the protocol within $\Delta$ of each other. When committing responsively at speeds independent of $\Delta$, the nodes can easily go out-of-sync. Typically, such synchronization between all the nodes is performed by multicasting synchronization proofs to all other nodes [24], [1]; in the absence of threshold signatures, these proofs tend to be $O(n)$-sized, making the communication cubic again. In OptRand, we instead broadcast reconstructed secrets opened in a verifiable manner in an epoch to synchronize all the nodes. The size of the reconstructed secret is $O(\kappa)$ bits and thus, the communication complexity stays quadratic.

In particular, OptRand combines this ability to move responsively to the next epoch with responsive propagation of large messages to obtain an optimistically responsive random beacon. The resulting random beacon protocol can output beacon values responsively whenever more than $3n/4$ nodes and the leader of the epoch are honest, and otherwise, emits the next beacon value every $11\Delta$ time.

**Reconfiguration mechanism.** Additionally, we also present a reconfiguration mechanism that allows a new node to enter the system with a latency of $t+1$ epochs, which is $2t+2$ epochs (explained later) faster than RandPiper [10]. The added benefit of responsiveness means the $t+1$ epochs can be responsive leading to even faster reconfiguration. A key improvement is in the synchronization mechanism [24], [1] to allow the new node to synchronize with all the existing honest nodes; while prior work required $2t+2$ epochs to perform this, we use a more efficient synchronization mechanism at the end of every epoch to synchronize the new node.

**Implementation and Evaluation.** We implement and evaluate the performance of our protocols and compare it with state-of-the-art synchronous random beacon protocols. In our evaluation, we observe that our protocol generates beacons at significantly higher rate than other protocols under optimistic conditions. Under non-optimistic cases, our protocol offers comparable performance. Additionally, we also implement and evaluate our reconfiguration protocol. We find that our system can seamlessly generate beacons even when removing and adding new nodes with good performance.

In summary,

- ✓ We present an efficient random beacon protocol assuming broadcast channels in Section IV.
- ✓ We present an optimistically responsive random beacon protocol with $O(\kappa n^2)$ communication in Section V. The resulting protocol is reconfiguration-friendly and can be used as an optimistically responsive BFT SMR protocol.
- ✓ We present our reconfiguration scheme in Section VI.
- ✓ We evaluate our OptRand protocol in Section VII.

**Related Work.** Table I compares the related beacon protocols comprehensively. Although several protocols with always quadratic communication complexity exist [14], [34], [49], [23], [7], [10], they lack responsiveness or reconfiguration-friendliness. Most protocols [14], [35], [21], [34], [49], [17], [7] are not reconfiguration-friendly and assume a threshold setup that needs to be re-generated every time any node in the system changes. To the best of our knowledge, our study shows that responsive and reconfiguration-friendly synchronous random beacons with optimal communication complexity and fault-tolerance were not explored previously and OptRand is the first protocol to achieve them.

## II. Related Work

### A. Related Works in the Random Beacon Literature

There has been a long line of work on distributed public randomness starting from Blum's two-node coin-tossing protocol [11]. Due to its practical application, the problem has been studied under various system models [55], [26], [17], [49], [50], [14], [35], [34], [21], [23]. We review the most recent and closely related works below. Compared to all of these protocols, OptRand has optimal resilience, perfect unpredictability, incurs $O(\kappa n^2)$ communication per beacon output and has a reusable setup. Moreover, OptRand is optimistically responsive i.e., it can make progress at the speed of actual network delay $\delta$ during optimistic conditions despite synchrony assumption.

The protocols by Cachin et al. [14], Drand [28] and Dfinity [35] require DKG [54] to setup threshold keys among participating nodes. STROBE [7] uses a similar threshold-RSA based setup to generate beacons. Although these protocols have optimal resilience, perfect unpredictability, and quadratic communication complexity per beacon output, these protocols do not have reusable setup i.e., replacing a single node in the system involves re-running the setup all over again which blows up communication.

HERB [21] and GULL [17] use partial homomorphic ElGamal encryption scheme to generate random numbers. HERB [21] tolerates only $t < n/3$ failures despite synchrony assumption and uses bulletin boards to post random shares. Mt. Random [17] uses PVSS and threshold ElGamal, protocols from Cachin et al. [14], VRG and assumes bulletin boards to realize their random beacons. Instantiating bulletin boards using Byzantine Consensus primitives trivially incurs $O(\kappa n^3)$. Moreover, both protocols use a variant of threshold setup and thus lack a reusable setup.

RandShare [55] assumes an asynchronous network and requires executing $n$ concurrent instances of Byzantine Agreement with a worst case communication of $O(\kappa n^4)$. RandHerd [55] improves on RandShare by sampling the system into smaller groups of size $c$ resulting in a communication complexity of $O(\kappa c^2 \log n)$ in the common case. RandHound [55] further improves on RandHerd by building tree-based hierarchy among the nodes and executes leader-based Byzantine Consensus among sub-trees. The resulting construction has a communication of $O(\kappa c^2 \log n)$ when all leaders are honest. With a sequence of Byzantine leaders, the communication worsens to $O(\kappa n^3)$.

RandChain [34] has optimal resilience of $t < n/2$, and incurs $O(\kappa n^2)$ per beacon output. However, they use computationally expensive sequential Proof-of-Work, and VDFs

along with Nakamoto consensus for consistency and has high computation cost.

RandRunner [49] 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. While RandRunner has qudratic communication per round, it has worst case unpredictability of $t+1$ rounds.

Most relevant to our protocol are Hydrand [50], RandPiper [10] and SPURT [23]. HydRand [50] tolerates only $t < n/3$ faults despite assuming synchrony. While Hydrand has low computation overhead and a reusable setup due to its use of PVSS scheme, it incurs $O(\kappa n^3)$ communication in the worst case and an unpredictability of $t+1$ rounds in the worst case.

RandPiper [10] improved upon Hydrand by designing a communication efficient BFT SMR protocol. Using the SMR protocol, they obtain a random beacon protocol with optimal resilience, quadratic communication and reusable setup but with worst case unpredictability of $t+1$ rounds. To provide perfect unpredictability, they propose BRandPiper [10] using VSS scheme to share $n$ secret in each round. The resulting construction has $O(\kappa f n^2)$ communication where $f < t$ is the actual number of faults. However, when $f = \Theta(n)$, the communication is $O(\kappa n^3)$. Moreover, their construction incur large latency to generate random beacons.

Recently, Das et al. proposed SPURT [23] in the partial synchronous model with perfect unpredictability, reusable setup and responsiveness and an always $O(\kappa n^2)$ communication. They use aggregatable PVSS scheme to combine $t+1$ PVSS vector in each round and provide perfect unpredictability with $O(\kappa n^2)$ communication.

### B. Related Works in the BFT SMR Literature

There has been a long line of work in improving communication complexity of consensus protocols [37], [29], [1], [57], [4], [43] and round complexity of consensus protocols [25], [1], [8], [29], [31], [37], [47]. We review the most recent and closely related works below. Compared to all of these protocols, our protocol incurs $O(\kappa n^2)$ communication per consensus decision while avoiding the use of threshold signatures. Moreover, our protocol is optimistically responsive with a responsive commit latency of $4\delta$ and synchronous commit latency of $4\Delta+3\delta$ in common case (or $7\Delta$ in the worst case). Our protocol follows rotating leader paradigm and can change leaders in optimistically responsive manner.

With respect to the communication complexity, the state-of-the-art synchronous BFT SMR protocols [1], [3], [52], [5], [6] incur quadratic communication per consensus decision while using threshold signatures. Without threshold signatures, they incur cubic communication per consensus decision. To the best of our knowledge, the only optimally resilient protocol to achieve $O(\kappa n^2)$ communication without threshold signature is BFT SMR protocol of RandPiper [10]. However, their protocol is not responsive even under optimistic conditions and commits a decision every $11\Delta$ time.

With respect to optimistic responsiveness, protocols due to Thunderella [45] and Sync HotStuff [3] are presented in a back-and-forth slow-path–fast-path paradigm. If started in the wrong path, these protocol cannot commit responsively. Recent work such as PiLi [19], OptSync [52] and Hybrid-BFT [42] achieve simultaneity between responsive and synchronous modes. However, they incur cubic communication without the use of threshold signatures. Ours is the first work that achieves simultaneity under synchrony assumption with $O(\kappa n^2)$ communication while avoiding threshold signatures.

*OptSync.* OptSync [52] presents an optimistically responsive protocol with optimal $2\delta$ latency during responsive commit and $2\Delta$ synchronous latency. However, their protocol follow stable leader paradigm and incur synchronous delay of $2\Delta$ while changing leaders. They also provide a separate protocol that support changing leaders in optimistically responsive manner in $O(\delta)$ time. Compared to their protocol, our protocol can change leaders responsively only when the new leader has highest ranked certificate; otherwise our protocol incurs $2\Delta$ wait.

*Hybrid-BFT.* Hybrid-BFT [42] presents an optimistically responsive protocol with both responsive and synchronous commit paths existing simultaneously. They also follow rotating leader paradigm and has responsive commit latency of $2\delta$ and synchronous commit latency of $2\Delta+2\delta$. Similar to our work, their protocol can also change leaders in responsive manner only when the new leader has highest ranked certificate; otherwise the protocol waits for $2\Delta$ time.

## III. System Model and Definitions

We consider a system $\mathcal{P} := \{p_1, \ldots, p_n\}$ consisting of $n$ nodes with reliable, authenticated point-to-point links, where up to $t < n/2$ nodes can be Byzantine faulty and can behave arbitrarily. We assume static corruption. A node that is not corrupted is considered to be honest and executes the protocol as specified.

Communication links between nodes are synchronous. If an honest node $p_i$ sends a message $x$ to another node $p_j$ at time $\tau$, $p_j$ receives the message by time $\tau+\delta$. The delay parameter $\delta$ is upper bounded by $\Delta$. The upper bound $\Delta$ is known, but $\delta$ is unknown to the system. $\delta$ can be regarded as an actual delay in the real-world network. We assume all honest nodes have clocks moving at the same speed. They also start executing the protocol within $\Delta$ time from each other. This can be easily achieved by using the clock synchronization protocol [1] once at the beginning of the protocol.

We employ digital signatures and public-key infrastructure (PKI) to prevent spoofing and validate messages. Message $x$ sent by a node $p_i$ is digitally signed by $p_i$'s private key and is denoted by $\langle x \rangle_i$. We use $H(x)$ to denote the invocation of the random oracle hash $H$ on input $x$.

### A. Definitions

**Pairings.** We assume a Type-III pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are cyclic groups of prime order $q$. Let $\mathbb{Z}_q$ be its scalar field. We will use the multiplicative notation

of groups for group operations in this paper. Let $g_1 \in \mathbb{G}_1$ and $g_2, g_2' \in \mathbb{G}_2$ be independent generators[1].

**State Machine Replication—SMR.** We consider a SMR protocol defined as follows:

**Definition III.1** (Byzantine Fault-tolerant SMR [51])**.** *A Byzantine fault-tolerant SMR 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:*

1) ***Safety****. Honest nodes do not commit different values at the same log position.*
2) ***Liveness****. Each client request is eventually committed by all honest nodes.*

**Random beacon.** We consider the following definition of a secure random beacon:

**Definition III.2** (Secure random beacon [23])**.** *A random beacon protocol is secure if for any PPT adversary $\mathcal{A}$ corrupting at most $t$ nodes in an epoch, $\mathcal{A}$ has a negligible advantage in the following security game played against a challenger $\mathcal{C}$.*

1. *$\mathcal{C}$ sends the setup parameters of the system.*
2. *$\mathcal{A}$ compromises up to $t$ nodes and notifies $\mathcal{C}$ of these nodes.*
3. *$\mathcal{C}$ creates the remaining public parameters (such as public keys) and sends them to $\mathcal{A}$.*
4. *$\mathcal{A}$ sends the remaining public parameters (such as public keys).*
5. *$\mathcal{C}$ and $\mathcal{A}$ execute the protocol per epoch:*
   - *$\mathcal{C}$ sends messages on behalf of the honest nodes to $\mathcal{A}$*
   - *$\mathcal{A}$ decides on the delivery of messages, and sends (or does not send) its messages.*
   - *The above steps are run interactively until an epoch ends and an honest node outputs the protocol transcript* $\mathsf{transcript}$.
6. *$\mathcal{C}$ samples a random bit $b \in \{0,1\}$ and sends either the beacon output based on* $\mathsf{transcript}$ *or a random $\mathbb{G}_T$ element.*
7. *$\mathcal{A}$ outputs a guess bit $b'$*

*The advantage of $\mathcal{A}$ is defined as* $|\mathsf{Prob}\left[b = b'\right] - 1/2|$.

While we define all notations as we introduce them, we also include a notations summary in the appendix in Table II.

### B. Employed Primitives

**1. Linear erasure and error correcting codes.** We use standard $(n, b)$ Reed-Solomon (RS) codes [48]. This code encodes $b$ data symbols into code words of $n$ symbols using the $\mathsf{ENC}$ function and can decode the $b$ elements of code words to recover the original data using the $\mathsf{DEC}$ function defined as follows:

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

In our protocol, we instantiate the RS codes with $n$ equal the number of all nodes, and $b$ equal to $\lfloor n/4 \rfloor + 1$.

**2. Cryptographic accumulators.** An accumulator scheme constructs a value called the accumulator to prove membership of elements using the $\mathsf{Eval}$ function, and produces a witness for each value in the accumulator using the $\mathsf{CreateWit}$ function. Given the accumulation value and a witness, any node can verify if a value is indeed in the set using the $\mathsf{Verify}$ function. An example accumulator is Merkle trees, where the root is the accumulator, and the paths are witnesses to leaves. In this paper, we employ *collision-free bilinear accumulators* from Nguyen [44] which generates constant-sized witness and accumulators. The bilinear accumulators of Nguyen [44] requires $q$-SDH assumption. Merkle trees [41] can be used instead, at the expense of $O(\log n)$ multiplicative communication complexity.

Formally, given a parameter $\kappa$, and a set $D$ of $n$ values $d_1, \ldots, d_n$, an accumulator has the following interface:

- $\mathsf{Gen}(1^\kappa, n)$: takes a parameter $\kappa$ and an accumulation threshold $n$ (an upper bound on the number of values that can be accumulated securely), returns an accumulator key $ak$. The accumulator key $ak$ is part of the trusted setup and therefore is public to all nodes.
- $\mathsf{Eval}(ak, \mathcal{D})$: takes an accumulator key $ak$ and a set $\mathcal{D}$ of values to be accumulated, returns an accumulation value $z$ for the value set $\mathcal{D}$.
- $\mathsf{CreateWit}(ak, z, d_i, \mathcal{D})$: takes an accumulator key $ak$, 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 $d_i \in \mathcal{D}$.
- $\mathsf{Verify}(ak, z, w_i, d_i)$: takes an accumulator key $ak$, 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.

The bilinear accumulator satisfies the following property:

**Lemma 1** (Collision-free accumulator [44])**.** *The bilinear accumulator is collision-free. That is, for any set of size $n$ and a probabilistic polynomial-time adversary $\mathcal{A}$, the following function is negligible in $\kappa$:*

$$\mathsf{Pr}\left[\begin{array}{c} ak \leftarrow \mathsf{Gen}(1^\kappa, n), \\ (\{d_1, \ldots, d_n\}, d', w') \leftarrow \\ \mathcal{A}(1^\kappa, n, ak), \\ z \leftarrow \mathsf{Eval}(ak, \{d_1, \ldots, d_n\}) \end{array} \middle| \begin{array}{c} (d' \notin \{d_1, ..., d_n\}) \wedge \\ (Verify(ak, z, w', d') = 1) \end{array}\right]$$

**3. Discrete log proof of knowledge.** Let $g, u \in \mathbb{G}$ be public values with $\mathbb{Z}_q$ as the scalar field. A prover $P$ who knows the value $x \in \mathbb{Z}_q$ such that $u = g^x$, and wants to prove this non-interactively in zero-knowledge runs the algorithm $\pi \leftarrow \mathsf{NIZKPK}(x, g, u)$ to generate a non-interactive zero-knowledge (NIZK) proof $\pi$. The proof $\pi$ can be verified using $\{0,1\} \leftarrow \mathsf{Verify}(\pi, g, u)$ by anyone.

We will also use the same notation $\pi \leftarrow \mathsf{NIZKPK}(x, g_1, u_1, g_2, u_2)$ to prove knowledge of $s \in \mathbb{Z}_q$ for public values $g_1, u_1 \in \mathbb{G}_1$ and $g_2, u_2 \in \mathbb{G}_2$ and $\{0,1\} \leftarrow \mathsf{Verify}(\pi, g_1, u_1, g_2, u_2)$. The same proof technique

[1] By independent generators, we mean that the adversary controlling $t$ nodes does not know a value $x \in \mathbb{Z}_q$ such that $e(g_1, 1_{\mathbb{G}_2})^x = e(1_{\mathbb{G}_1}, g_2)$ and similarly a value $y \in \mathbb{Z}_q$ such that $g_2^y = g_2'$.

from [20] is used to prove discrete log equality of logarithms, i.e., prove that $\log_{g_1} u_1 = \log_{g_2} u_2$.

**4. SCRAPE PVSS [16].** OptRand relies on a modified version of the pairing based PVSS scheme introduced in SCRAPE ([16, Section 4]). We refer interested readers to the full version [9, Appendix] for a summary of the original SCRAPE pairing-based PVSS scheme.

*Setup*: Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be an efficient pairing group with independent generators[1] $g_1 \in \mathbb{G}_1, g_2, g_2' \in \mathbb{G}_2$ which can be derived assuming a common reference string assumption [40]. Every node $p_i \in \mathcal{P}$ has a secret key $sk_i \leftarrow \mathbb{Z}_q$ and public keys $pk_i = g_1^{sk_i}$. We denote the public keys of all the nodes as follows $\mathbf{pk} := \{pk_1, pk_2, \ldots, pk_n\}$. The setup for keys is realized using a PKI assumption.

The protocol consists of the following algorithms:

*Sharing phase*: Here, the dealer $L \in \mathcal{P}$ chooses a random value $s \in \mathbb{Z}_q$ and creates a polynomial $p(x) \in \mathbb{Z}_q[x]$ of degree $t$ with $p(0) = s$. Without loss of generality, we employ indices from $1, \ldots, n$ such that the shares $s_i$ for node $p_i$ is $p(i)$.

*Commitment*: In SCRAPE, the dealer *commits* to polynomial $p(x)$ by producing a commitment vector: $\mathbf{v} = \{g_2^{s_1}, g_2^{s_2}, \ldots, g_2^{s_n}\}$. SCRAPE [16, Section 2] observes that if $C = \{p(1), \ldots, p(n)\}$ where $p(x)$ is a degree $t$ polynomial, then there exists a dual code $C^{\perp} = \{\mu_1 f(1), \ldots, \mu_n f(n)\}$, where $\mu_i = \prod_{i=1, i\neq j}^{n} \frac{1}{i-j}$ and $f(x)$ is a random polynomial of degree up to $n - t$, such that their dot-product is zero.

This check easily ensures that the commitment vector $\mathbf{v}$ is also a *commitment* to a $t$ degree polynomial by checking the following:

$$\prod_{i=1}^{n} v_i^{c_i^{\perp}} = 1_{\mathbb{G}_2}, \ \{c_1^{\perp}, c_2^{\perp}, \ldots, c_n^{\perp}\} \in C^{\perp} \tag{1}$$

*Encryption*: Along with the commitments, the dealer creates encryptions $\mathbf{c} = \{pk_1^{s_1}, pk_2^{s_2}, \ldots, pk_n^{s_n}\}$ for all the nodes. Nodes check the correctness of the encryptions by checking $e(pk_i, v_i) = e(c_i, g_2)$ both of which are $e(g_1, g_2)^{sk_i s_i}$.

*Decryption*: Every node computes $d_i \leftarrow c_i^{sk_i^{-1}} = g_1^{s_i}$. Nodes verify decryptions $d_j$ from others by checking $e(d_j, g_2) = e(g_1, v_j)$ both of which are $e(g_1, g_2)^{s_j}$.

*Reconstruction*: In this phase, the nodes reconstruct $B \leftarrow g_1^s = g_1^{p(0)}$ and compute $S \leftarrow e(g_1^s, g_2')$ as the secret $S$. The IND1-secrecy definition requires that an adversary is not able to tell whether the sharing is for the secret $S$ even if $S$ was provided to the adversary. Thus, it is important the secret is defined as $S \in \mathbb{G}_T$ and not $g_2^s$, as the latter directly reveals whether the sharing is for $S$.

## IV. Warm-up: Random Beacons Using a Broadcast Channel

In this section, we will describe a warm-up random beacon protocol using a pairing-based publicly verifiable secret-sharing scheme in the broadcast-channel model with the following security properties:

**Definition IV.1** (Warm up beacon)**.** *Let $L \in \mathcal{P}$ be a dealer. The warm-up beacon guarantees the following:*

1) ***Weak agreement.*** *Let $\mathcal{B}$ be the space of all beacon values. Then all honest nodes output the same value in $\mathcal{B} \cup \perp$.*

2) ***Value validity.*** *If an honest node outputs $v \neq \perp$, then $v$ is uniformly random in $\mathcal{B}$.*

3) ***Validity.*** *If $L$ is honest, then the value $v$ output by all the honest nodes satisfies $v \neq \perp$.*

In Fig. 1 we describe our broadcast channel based protocol satisfying Definition IV.1. Weak agreement allows honest nodes to output $\perp$ if the leader is Byzantine. Value validity guarantees that if an honest node outputs $v \neq \perp$, then it must uniformly random even if the leader is Byzantine. The scheme in Fig. 1, which is a combination of techniques from Gurkan et al. [32], Das et al. [23], and SCRAPE [16], allows us to realize OptRand in Section IV.

We consider Fig. 1 to be a warm-up as it misses two factors: first, the honest nodes can output $\perp$ when $L$ is Byzantine; secondly, we also need to implement the broadcast channel using a responsive SMR with the constraint that the communication complexity cannot exceed $O(\kappa n^2)$ in an epoch. We will overcome these limitations using rotating leaders, buffering of shares, and a novel optimistically responsive SMR with the desired properties in the next section.

### A. Our Protocol

Our protocol (Fig. 1) satisfies Definition IV.1 and consists of a designated leader $L \in \mathcal{P}$, who acts as the coordinator. When the leader is honest, all the honest nodes obtain a secret share, which when used for reconstruction outputs a unique and unpredictable element $S \in \mathbb{G}_T$. When the leader is Byzantine, either all the honest nodes commit shares for an element $g_1^s$ from which we build a random unpredictable element $S \in \mathbb{G}_T$, or all the honest nodes abort, i.e., output $\perp$.

**Decomposition proofs.** When using a leader as the coordinator, if we use SCRAPE naively, we need a mechanism to prevent the possibility that a leader can simply cancel the contributions of honest nodes by proposing a PVSS vector where every element is raised to $-1$. We mitigate this by adding a simple cryptographic proof of knowledge of the shared secret $\widetilde{\pi}$ called *decomposition proof*. Here, a node proves that it knows the secret (or $p(0)$) using the discrete log proof of knowledge [20] algorithm NIZKPK discussed in Section III-B. This proves to everyone non-interactively that the proposer knows the secret being shared without revealing it.

Our protocol forces the leader to propose the combined commitments and encryptions and produce at least $t+1$ proofs of knowledge $\widetilde{\pi}$ values. These proofs are $O(1)$ sized making the whole proposal $O(n)$ sized. Since $n-t > t$, we know that there must always be $t+1$ valid shares. So an honest leader will always have sufficient proofs to propose. Therefore, if a leader does not propose, it must be Byzantine. Since any valid proposal from a leader must include $t+1$ proofs of knowledge, at least one of them is a sharing for a random number, we can intuitively guarantee *Value validity* (Property 2).

**Sharing phase.** The sharing phase consists of three steps: (i) Commitment, (ii) Aggregation, and (iii) Commit steps. We detail the steps below.

Let $L \in \mathcal{P}$ be the leader. Every node $p_i \in \mathcal{P}$ does the following:

**Sharing Phase**

*Commitment*

1. Generate a random degree $t$ polynomial $p_i(x)$ and compute the shares for every $j \in [n]$ as follows:

$$\begin{gathered} s_j \leftarrow p_i(j), v_j \leftarrow g_2^{s_j}, c_j \leftarrow pk_i^{s_i} \\ \mathbf{v}_i \leftarrow \{v_1, \ldots, v_n\}, \mathbf{c}_i \leftarrow \{c_1, \ldots, c_n\} \end{gathered} \tag{2}$$

2. Compute a signed knowledge proof $\widetilde{\pi}_i$ using

$$\widetilde{\pi}_i \leftarrow \langle g_2^{p_i(0)}, \mathsf{NIZKPK}(p_i(0), g_2, g_2^{p_i(0)}) \rangle_i \tag{3}$$

3. Send $(\mathbf{v}_i, \mathbf{c}_i, \widetilde{\pi}_i)$ to the leader $L$.

*Aggregation*

The leader does the following:

1. Collect $t+1$ $S_j := (\mathbf{v}_j, \mathbf{c}_j, \widetilde{\pi}_j)$ from $p_j$ satisfying the following:
   a) $\prod_{i=1}^{n} v_i^{c_i^{\perp}} = 1_{\mathbb{G}_2},\ \{c_1^{\perp}, c_2^{\perp}, \ldots, c_n^{\perp}\} \in C^{\perp}$
   b) $e(pk_i, g_2) = e(c_i, v_i)$
   c) $\mathsf{Verify}(y_b, g_2, y_a)$ with $\widetilde{\pi}_j := (y_a, y_b)$
   d) $y_a = \mathtt{Interpolate}(\mathbf{v}_j)$ with $\widetilde{\pi}_i := (y_a, y_b)$
   e) Verify signed $\widetilde{\pi}_j$ with verification key of node $p_j$
2. Aggregate valid shares as follows:

$$\begin{gathered} \mathbf{v} \leftarrow \mathbf{v}_1 + \cdots + \mathbf{v}_{t+1}, \\ \mathbf{c} \leftarrow \mathbf{c}_1 + \cdots + \mathbf{c}_{t+1}, \\ \boldsymbol{\pi} \leftarrow \{\pi_1, \ldots, \pi_{t+1}\} \end{gathered} \tag{4}$$

3. Post $(\mathbf{v}, \mathbf{c}, \boldsymbol{\pi})$ on the broadcast channel.

*Commit*

1) For the aggregated sharing $(\mathbf{v}, \mathbf{c}, \boldsymbol{\pi})$, check
   a) $\prod_{i=1}^{n} v_i^{c_i^{\perp}} = 1_{\mathbb{G}_2},\ \{c_1^{\perp}, c_2^{\perp}, \ldots, c_n^{\perp}\} \in C^{\perp}$
   b) $e(pk_i, v_i) = e(c_i, g_2)$
   c) $\mathsf{Verify}(y_{bi}, g_2, y_{ai})$ with $\widetilde{\pi}_i := (y_{ai}, y_{bi}) \forall \widetilde{\pi}_i \in \boldsymbol{\pi}$
   d) $\forall \widetilde{\pi}_i := (y_{ai}, y_{bi}) \in \boldsymbol{\pi}$, $\prod_i y_{ai} = \mathtt{Interpolate}(\mathbf{v})$
   e) Verify signed $\widetilde{\pi}_i \in \boldsymbol{\pi}$ with verification key of node $p_i$
2) If a valid $(\mathbf{v}, \mathbf{c}, \boldsymbol{\pi})$ is posted in time, commit $(\mathbf{v}, \mathbf{c})$. Otherwise, commit $\perp$.

**Reconstruction Phase**

1. Send $d_i \leftarrow c_i^{sk_i^{-1}}$ to all the nodes.
2. Collect $t+1$ shares $d_j$ which satisfy $e(d_j, g_2) = e(g_1, v_i)$ from node $p_j \in \mathcal{P}$ in the set $\mathbf{S}$.
3. Using Lagrange interpolation compute $B \leftarrow \mathtt{Interpolate}(\mathbf{S})$.
4. Compute the beacon value $S \leftarrow e(S, g_2')$.

Fig. 1: **A warm up beacon protocol** using a modified pairing bases SCRAPE PVSS [16] and broadcast-channels.

*1. Commitment.* In this step, each node $p_i$ computes the publicly-verifiable shares for every nodes using Step 1 in the sharing phase in Fig. 1. The node also signs and sends the decomposition proof $\widetilde{\pi}_i$. Finally, all the nodes send their commitments and encryptions to the leader $L$.

*2. Aggregation.* The leader receives valid $t+1$ PVSS vectors from nodes in set $I$. In this step, the leader combines it to produce the final PVSS vector for a random polynomial $P$. This consists of the combined commitments, combined encryption and aggregated proofs (Eq. (4)) for $j \in I$ and for all the nodes $p_i \in \mathcal{P}$.

After combining the PVSS vectors, the leader broadcasts $(\mathbf{v}, \mathbf{c}, \widetilde{\pi})$. Note that this post has a size of $O(n)$.

*3. Commit.* In the commit step, all the nodes observe the sharing (or its lack thereof within sufficient time) on the broadcast channel and decide to commit, or to abort by checking (i) the $\mathbf{v}$ is a valid $(n, t)$ sharing, (ii) the shares for all the nodes are correct, (iii) all the constituent decomposition proofs are valid, and finally (iv) the interpolated values for $\mathbf{v}$ and the product of all the first elements in $\boldsymbol{\pi}$ are the same. If the leader was honest, then all the honest nodes commit. If the leader was Byzantine, the sharing will still include at least one honest node's contribution and if committed by all the honest nodes, is secure.

**Reconstruction phase.** The nodes decrypt their shares by cancelling the secret key $sk_i$ from the exponent in $c_i$. The decrypted share $d_i$ is then sent to all the nodes who can non-interactively verify the validity. On collecting $t+1$ valid decryption shares, the nodes reconstruct $B \leftarrow g_1^s$ using to obtain the final secret $S \leftarrow e(B, g_2')$.

**Verifiable beacon.** The pairing-based PVSS scheme allows for any node to verify the correctness of the reconstructed unpredictable secret. In particular, on reconstructing $B \leftarrow g_1^s$, any node can confirm that this is the correct beacon value against a sharing, by checking $e(B, g_2) = e(g_1, g_2^s)$, where $g_2^s$ can be generated using Lagrange interpolation on $\mathbf{v}$. This property is critical to obtain *optimistic responsiveness* in the next section.

**Security analysis.** We defer the security analysis to Section B and Section C.

Note that the protocol described in Fig. 1, can result in nodes aborting the beacon generation when the leader is Byzantine. This can make the protocol violate guaranteed output delivery. In the next section, we overcome the problem while maintaining quadratic communication complexity using pipelined SMRs [57], [3] and pre-processing.

## V. Optimistically Responsive Random Beacon

In this section, we present OptRand, an optimistically responsive random beacon protocol. Our protocol is a novel combination of a state machine replication (SMR) protocol and a random beacon protocol to achieve an optimistically responsive random beacon. Our protocol uses the generated random beacons to achieve responsiveness. In particular, we use aggregated secrets to synchronize between honest nodes and achieve responsiveness.

The underlying SMR protocol includes an optimistic path that can make progress at the network speed i.e., in $O(\delta)$ time during optimistic condition when the leader and $> 3n/4$ nodes behave honestly. A quorum of $\lfloor 3n/4 \rfloor + 1$ nodes are required for an optimistically responsive protocol [45]. Under standard conditions, i.e., when only $> n/2$ nodes behave honestly, the SMR protocol makes progress in $\Omega(\Delta)$ time. We follow the optimistic responsive paradigm of OptSync [52], i.e., our protocol does not require explicit back-and-forth switching between slow synchronous mode and fast optimistic mode employed in [45], [3]. Similar to the optimistically responsive view-change protocol in OptSync, our protocol changes leaders in an optimistically responsive manner.

**Epochs.** Our protocol progresses through a series of numbered *epochs* with epoch $r$ coordinated by a distinct leader $L_r$

rotated in a round-robin manner every epoch. During optimistic conditions, the system progresses through epochs responsively, i.e., in $O(\delta)$ time; otherwise each epoch lasts for $O(\Delta)$ time.

**Blocks and block format.** We represent a proposal in an epoch in the form of a *block*. Each block references its predecessor to form a blockchain with the exception of the genesis block which has no predecessor. We call a block's position in the blockchain 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 $h$ and $H(B_{h-1})$ is the hash digest of $B_{h-1}$. The predecessor for the genesis block is $\perp$. In our protocol, the payload $b_h$ is set to the aggregated PVSS commitment and encryption. 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 is a valid PVSS vector, i.e., the *verification* algorithm outputs a 1 (discussed in Commit step in Fig. 1). A block $B_h$ *extends* a block $B_l$ ($h \geq l$) if $B_l$ is an ancestor of $B_h$.

**Certified blocks, and locked blocks.** A *block certificate* represents a set of signatures on a block in an epoch by a quorum of nodes. We use two types of signed vote messages: a responsive vote resp-vote and a synchronous vote sync-vote. Accordingly, we consider two *types* of block certificates. A *responsive certificate* $\mathcal{C}_r^{3/4}(B_h)$ for a block $B_h$ consists of $\lfloor 3n/4 \rfloor + 1$ distinct resp-vote on $B_h$ in epoch $r$. Similarly, a *synchronous certificate* $\mathcal{C}_r^{1/2}(B_h)$ consists of $t + 1$ distinct sync-vote on $B_h$ in epoch $r$. Whenever the distinction is not important, we will represent the certificates by $\mathcal{C}_r(B_h)$.

Certified blocks are ranked by epochs, i.e., blocks certified in a higher epoch have a higher rank. We do not rank between responsive and synchronous certificate from the same epoch. During the protocol execution, each node keeps track of all certified blocks and keeps updating the highest ranked block certificate to its knowledge. Nodes will lock on highest ranked block certificate and do not vote for blocks that do not extend highest ranked block certificates 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 $r$ sends propose, resp-cert, and sync-cert messages (explained later) to all other nodes. In order to facilitate efficient equivocation checks, the leader sends the payload along with signed hash of the payload. When an equivocation is detected, broadcasting the signed hash suffices to prove equivocation by $L_r$.

**Background: Dissecting BFT SMR protocol of RandPiper [10].** A key component of RandPiper is a communication efficient BFT SMR protocol that incurs $O(\kappa n^2)$ communication per decision to decide on $O(n)$-sized input without using threshold signatures. The efficient communication was achieved by making use of erasure coding schemes, cryptographic accumulators and broadcast of equivocating hashes (if any). In their protocol, they use $(n, t + 1)$ RS codes to encode large messages. When a node receives a valid proposal from the leader, they use RS codes to encode the proposal into $n$ code words ($s_1$, ..., $s_n$) and compute corresponding cryptographic witnesses $(w_1, \ldots, w_n)$, and send each code word and witness pair ($s_i$, $w_i$) to node $p_i$ $\forall p_i \in \mathcal{P}$. A node votes for the proposed block only if it does not detect any equivocation for $2\Delta$ time. The $2\Delta$ wait before voting ensures (i) no honest node received an equivocating proposal and conflicting $(s_i', w_i')$ before receiving $(s_i, w_i)$ (ii) all honest nodes receive at least $t+1$ code words for the proposed block sufficient to reconstruct the proposal.

To ensure safety of a committed block, in general, SMR protocols ensure that all honest nodes receive and lock a certificate for the proposed block. A certificate consisting of $t + 1$ signatures for the proposed block is linear in size in the absence of threshold signatures. Thus, an all-to-all broadcast of the certificate trivially incurs cubic communication. The BFT SMR protocol of RandPiper solves the issue using following technique. First, nodes send their vote only to the leader. The leader is expected to collect $t + 1$ votes, form a single certificate and send it to all nodes. Second, in order to ensure the certificate is propagated among all honest nodes, instead of broadcasting it to all nodes, they use RS codes to encode the certificate, send the code word and witnesses and wait for $2\Delta$ to check for an equivocation before making a commit.

**Achieving optimistic responsiveness.** The techniques employed by the BFT SMR protocol enables communication efficient consensus on $O(n)$-sized input. However, their technique requires waiting for $\Omega(\Delta)$ time to detect equivocation before making a decision.

In this paper, we propose a new technique that allows us to responsively make decision and change leaders without relying on equivocation detection. We modify RandPiper in the following manner: First, we use $(n, \lfloor n/4 \rfloor + 1)$ RS codes to encode large messages (in the Deliver primitive in Fig. 2). This allows decoding with $\lfloor n/4 \rfloor + 1$ code words at the expense of doubled code word size. Second, a node sends a responsive vote to the leader as soon as it receives a valid block proposal. The node also sends the RS coded code words and witnesses to all other nodes. The leader collects $\lfloor 3n/4 \rfloor + 1$ votes to form a responsive certificate and sends the responsive certificate to all nodes. The nodes broadcast an ack message in response to the responsive certificate and commit on receiving $> 3n/4$ distinct ack messages. In addition, they also send RS coded code words and witnesses for the responsive certificate. The existence of $> 3n/4$ ack messages ensures that all honest nodes can decode the proposed blocks and the responsive certificate. In particular, at least $\lfloor n/4 \rfloor + 1$ honest nodes must have received the block proposal and the responsive certificate for the committed block and they have forwarded their code words to all nodes. Thus, all honest node must receive $\lfloor n/4 \rfloor + 1$ code words sufficient to decode the proposed blocks and the responsive certificate.

**Responsively changing epochs.** The above technique allows an honest node to responsively commit a decision. In order to responsively change epochs, a synchronization primitive is required to signal all honest nodes to move to a higher epoch. Prior works [3], [52], [5] perform an all-to-all broadcast of certificates to synchronize between epochs which incurs cubic communication without threshold signatures. In this protocol, we broadcast aggregated secret opened in an epoch to synchronize all the nodes. The size of aggregated secret is $O(1)$ bits and all-to-all broadcast of $O(1)$-sized aggregated secret does not blow up communication.

In cases when optimistic conditions are not met, the underlying consensus mechanism works similar to the BFT

Deliver(mtype, $m, z_r, r$):

1) Partition input $m$ into $\lfloor n/4 \rfloor + 1$ data symbols. Encode the $\lfloor n/4 \rfloor + 1$ data symbols into $n$ code words $(s_1, \ldots, s_n)$ using the ENC function. Compute witness $w_j \ \forall s_j \in (s_1, \ldots, s_n)$ using CreateWit function. Send $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_r, r \rangle_{p_i}$ to node $j \ \forall j \in [n]$.
2) If $j^{th}$ node receives the first valid code word $\langle \text{codeword}, \text{mtype}, s_j, w_j, z_r, r \rangle$ for the accumulator $z_r$, forward the code word to all the nodes.
3) Upon receiving $\lfloor n/4 \rfloor + 1$ valid code words for a common accumulator $z_r$, decode $m$ using the DEC function.

Fig. 2: **Deliver function**

SMR in RandPiper except we use $(n, \lfloor n/4 \rfloor + 1)$ RS codes.

### A. Protocol Details

**Deliver function.** We first present a Deliver function (refer Fig. 2) that is used by an honest node to propagate long messages received from the epoch leader. The Deliver function enables efficient broadcast of long messages using erasure coding techniques and cryptographic accumulators. The input parameters to the function are a keyword mtype, long message $m$, accumulation value $z_r$ corresponding to message $m$ and epoch $r$ in which Deliver function is invoked. The input keyword mtype corresponds to message *type* containing long message $m$ sent by leader $L_r$. In order to facilitate efficient leader equivocation, the input keyword mtype, hash of long message $m$, accumulation value $z_r$, and epoch $r$ are signed by leader $L_r$.

This function is similar to that in RandPiper [10] except that we use $(n, \lfloor n/4 \rfloor + 1)$ RS codes instead of $(n, t+1)$ RS codes used in [10]. As a result, the size of code word is doubled and the communication is increased by a factor of 2. However, this does not linearly blow up the communication complexity and the communication complexity still remains $O(\kappa n^2)$.

Our beacon protocol is described in Fig. 3. Nodes maintain a chain of blocks to add blocks proposed by leaders, a queue $\mathcal{Q}()$ to store a recently committed PVSS vector proposed by an epoch leader and set $\mathcal{P}_r$ to keep track of removed nodes. The queue $\mathcal{Q}()$ holds one PVSS vector per node. Before the start of the beacon protocol execution, a setup phase is executed where we establish PVSS parameters (namely $g_1 \in \mathbb{G}_1$, $g_2, h_2 \in \mathbb{G}_2$), and public keys $pk_i$ for every node $p_i \in \mathcal{P}$. We also buffer one secret share for aggregated PVSS tuples for every node $p_i$, i.e., fill $\mathcal{Q}(p_i)$ for $p_i \in \mathcal{P}$. This ensure the beacons are generated from the first epoch. The nodes in $\mathcal{P} \backslash \mathcal{P}_r$ are selected as leaders in a round-robin manner.

After the setup phase, the nodes execute following steps in each epoch $r$.

**Epoch advancement.** Each node keeps track of epoch duration $\text{epoch-timer}_r$ for epoch $r$. A node $p_i$ enters epoch $r$ (i) when its $\text{epoch-timer}_{r-1}$ expires, or (ii) when it receives a round $r-1$ aggregated secret $R_{r-1}$ and a round $r-1$ block certificate $\mathcal{C}_{r-1}(B_l)$. Upon entering epoch $r$, node $p_i$ generates PVSS vector $(\mathbf{v}_i, \mathbf{c}_i, \widetilde{\pi}_{K,i})$ (defined in *Commitment phase* of Fig. 1) and sends the PVSS tuple and its highest ranked certificate to the leader $L_r$. In addition, it aborts all timers below epoch $r$ and sets $\text{epoch-timer}_r$ to $11\Delta$ and starts counting down.

**Propose.** Upon entering epoch $r$, if Leader $L_r$ has $\mathcal{C}_{r-1}(B_l)$, it proposes as soon as it receives $t+1$ PVSS tuples; otherwise, it waits for $2\Delta$ time to ensure it can receive the highest ranked certificate from all honest nodes. Upon receiving $t+1$ PVSS tuples from $I \subset [n]$, it aggregates the PVSS tuples to obtain aggregated PVSS committments $\mathbf{v}$, aggregated encrypted secret shares $\mathbf{c}$ and NIZK proofs $\{\widetilde{\pi}_{K,i}\}_{i \in I}$ denoted as $\widetilde{\pi}_K$. The leader $L_r$ constructs a block $B_h$ by extending on the highest ranked certificate $\mathcal{C}_{r'}(B_l)$ known to $L_r$ with payload $b_h$ set to $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ and sends proposal $p_r := \langle \text{propose}, B_h, \mathcal{C}_{r'}(B_l), z_{pa}, r \rangle_{L_r}$ to node $p_j$, $\forall p_j \in \mathcal{P}$. Here, $z_{pa}$ is the accumulation value for the pair $(B_h, \mathcal{C}_{r'}(B_l))$. While conceptually, the leader is sending $\langle \text{propose}, B_h, \mathcal{C}_{r'}(B_l), z_{pa}, r \rangle_{L_r}$, to facilitate equivocation checks it instead sends $\langle \text{propose}, H(B_h, \mathcal{C}_{r'}(B_l)), z_{pa}, r \rangle_{L_r}$ with $(B_h, \mathcal{C}_{r'}(B_l))$ sent separately. The size of the signed message is $O(1)$ and hence can be broadcast during equivocation or while delivering proposal $p_r$ without incurring cubic communication overhead.

**Vote.** If node $p_i$ receives a proposal $p_r := \langle \text{propose}, B_h, \mathcal{C}_{r'}(B_l), z_{pa}, r \rangle_{L_r}$ it first checks PVSS verification for $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ is valid (refer Commit step in Fig. 1). We call such a proposal valid. If node $p_i$ receives a valid proposal and the proposed block $B_h$ extends the highest ranked certificate known to the node such that its $\text{epoch-timer}_r \geq 7\Delta$, then it invokes $\text{Deliver}(\text{propose}, p_r, z_{pa}, r)$ and sends a responsive vote $\langle \text{resp-vote}, H(B_h), r \rangle_{p_i}$ immediately to $L_r$. In addition, the node sets its $\text{vote-timer}_r$ to $2\Delta$ and starts counting down. When $\text{vote-timer}_r$ reaches 0 and detects no epoch $r$ equivocation, the node sends a synchronous vote $\langle \text{sync-vote}, H(B_h), r \rangle_{p_i}$ to $L_r$. If block $B_h$ does not extend the highest ranked certificate known to the node or node $p_i$ does not receive a proposal $p_r$ when its $\text{epoch-timer}_r < 7\Delta$, the node simply ignores the proposal and does not vote for $B_h$.

**Resp cert.** When the leader $L_r$ receives $\lfloor 3n/4 \rfloor + 1$ distinct resp-vote messages for the proposed block $B_h$ in epoch $r$, denoted by $\mathcal{C}_r^{3/4}(B_h)$, $L_r$ broadcasts $\langle \text{resp-cert}, \mathcal{C}_r^{3/4}(B_h), z_{ra}, r \rangle_{L_r}$ to all nodes where $z_{ra}$ is the accumulation value of $\mathcal{C}_r^{3/4}(B_h)$. Similar to the proposal, the hash of the certificate $\mathcal{C}_r^{3/4}(B_h)$ is signed to allow for efficient equivocation checks. Since our protocol requires the certificate to be *delivered* to all nodes in case of a commit, we require two different certificates for the same block shared by a leader to be considered an equivocation.

**Sync cert.** When leader $L_r$ receives $t+1$ distinct sync-vote messages for the proposed block $B_h$ in epoch $r$, denoted by $\mathcal{C}_r^{1/2}(B_h)$, $L_r$ broadcasts $\langle \text{sync-cert}, \mathcal{C}_r^{1/2}(B_h), z_{sa}, r \rangle_{L_r}$ to all nodes where $z_{sa}$ is the accumulation value of $\mathcal{C}_r^{1/2}(B_h)$. Again, the hash of the certificate $\mathcal{C}_r^{1/2}(B_h)$ is signed to allow for efficient equivocation checks.

**Ack.** When a node $p_i$ receives a responsive certificate $rc := \langle \text{resp-cert}, \mathcal{C}_r^{3/4}(B_h), z_{ra}, r \rangle_{L_r}$ while in epoch $r$, it invokes $\text{Deliver}(\text{resp-cert}, rc, z_{ra}, r)$ to deliver $rc$ and broadcasts $\langle \text{ack}, H(B_h), z_{ra}, r \rangle_{p_i}$ to all nodes. If $\text{epoch-timer}_r \leq 3\Delta$, node $p_i$ sets $\text{commit-timer}_r$ to $2\Delta$ and starts counting down.

**Commit.** The protocol includes two commit rules that commits

Let $r$ be the current epoch, $L_r$ be the leader of epoch $r$ and $\mathcal{P}_r$ be the set of removed nodes. For each epoch $r$, node $p_i \in \mathcal{P}$ performs following operations:

1) **Epoch advancement.** Node $p_i$ advances to epoch $r$ using following rules:
   a) When $\mathsf{epoch\text{-}timer}_{r-1}$ reaches 0, enter epoch $r$.
   b) On receiving aggregated secret $R_{r-1}$, broadcast $R_{r-1}$. Wait until $\mathcal{C}_{r-1}(B_l)$ is received and enter epoch $r$.
   Upon entering epoch $r$, send PVSS tuple $(\mathbf{v}_i, \mathbf{c}_i, \widetilde{\pi}_{K,i})$ and highest ranked certificate $\mathcal{C}_{r'}(B_l)$ to $L_r$. Set $\mathsf{epoch\text{-}timer}_r$ to $11\Delta$ and start counting down.
2) **Propose.** Wait for $t+1$ PVSS tuples and either $\mathcal{C}_{r-1}(B_l)$ or $2\Delta$ time after entering epoch $r$. Upon receiving $t+1$ valid PVSS tuples, $L_r$ aggregates them to obtain $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ (refer Aggregation Step in Fig. 1). Set $b_h := (\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ and send $\langle \mathsf{propose}, B_h, \mathcal{C}_{r'}(B_l), z_{pa}, r\rangle_{L_r}$ to node $p_j$ $\forall p_j \in \mathcal{P}$ where $B_h$ extends $B_l$ and $\mathcal{C}_{r'}(B_l)$ is the highest ranked certificate known to $L_r$.
3) **Vote.** If $\mathsf{epoch\text{-}timer}_r \geq 7\Delta$ and node $p_i$ receives the first proposal $p_r := \langle \mathsf{propose}, B_h, \mathcal{C}_{r'}(B_l), z_{pa}, r\rangle_{L_r}$, check the validity of the aggregated PVSS tuple (refer Commit Step in Fig. 1). If valid and $B_h$ extends a highest ranked certificate, invoke $\mathsf{Deliver}(\mathsf{propose}, p_r, z_{pa}, r)$ and send $\langle \mathsf{resp\text{-}vote}, H(B_h), r\rangle_{p_i}$ to $L_r$. Set $\mathsf{vote\text{-}timer}_r$ to $2\Delta$ and start counting down. When $\mathsf{vote\text{-}timer}_r$ reaches 0, send $\langle \mathsf{sync\text{-}vote}, H(B_h), r\rangle_{p_i}$ to $L_r$.
4) **Resp cert.** On receiving $\lfloor 3n/4 \rfloor + 1$ $\mathsf{resp\text{-}vote}$ for $B_h$, $L_r$ broadcasts $\langle \mathsf{resp\text{-}cert}, \mathcal{C}_r^{3/4}(B_h), z_{ra}, r\rangle_{L_r}$.
5) **Sync cert.** On receiving $t+1$ $\mathsf{sync\text{-}vote}$ for $B_h$, $L_r$ broadcasts $\langle \mathsf{sync\text{-}cert}, \mathcal{C}_r^{1/2}(B_h), z_{sa}, r\rangle_{L_r}$.
6) **Ack.** Upon receiving the first responsive certificate $rc := \langle \mathsf{resp\text{-}cert}, \mathcal{C}_r^{3/4}(B_h), z_{ra}, r\rangle_{L_r}$, invoke $\mathsf{Deliver}(\mathsf{resp\text{-}cert}, rc, z_{ra}, r)$ and broadcast $\langle \mathsf{ack}, H(B_h), z_{ra}, r\rangle_{p_i}$.
7) **Commit.** Node $p_i$ commits using one of the following rules:
   a) *Responsive.* If $\mathsf{epoch\text{-}timer}_r \geq 2\Delta$ and node $p_i$ receives $\langle \mathsf{ack}, H(B_h), z_{ra}, r\rangle$ from $\lfloor 3n/4 \rfloor + 1$ distinct nodes and detects no equivocation, commit $B_h$ and all its ancestors.
   b) *Synchronous.* If $\mathsf{epoch\text{-}timer}_r \geq 3\Delta$ and node $p_i$ receives the first certificate (either responsive or synchronous), set $\mathsf{commit\text{-}timer}_r$ to $2\Delta$ and start counting down. If the received certificate is synchronous i.e., $sc := \langle \mathsf{sync\text{-}cert}, \mathcal{C}_r^{1/2}(B_h), z_{sa}, r\rangle_{L_r}$, invoke $\mathsf{Deliver}(\mathsf{sync\text{-}cert}, sc, z_{sa}, r)$. When $\mathsf{commit\text{-}timer}_r$ reaches 0, if no epoch-$r$ equivocation has been detected, commit $B_h$ and all its ancestors.
8) **Update, reconstruct and output.** When node $p_i$ commits or when epoch $r$ ends, perform following operations:
   a) Commit block $B_\ell$ proposed in epoch $r-t$ if the highest ranked chain extends $B_\ell$ (if $B_\ell$ has not been committed).
   b) If block $B_\ell$ proposed by $L_{r-t}$ has been committed by epoch $r$, update $\mathcal{Q}(L_{r-t})$ with $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ shared in $b_\ell$. Otherwise, remove $L_{r-t}$ from future proposals, i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{r-t}\}$.
   c) Obtain $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ corresponding to block committed in $\mathsf{Dequeue}(\mathcal{Q}(L_r))$. Broadcast decrypted share $d_i$. On receiving share $d_j$ from another node $p_j$, ensure that $\mathsf{ShVrfy}(pk_j, c_j, d_j) = 1$. On receiving $t+1$ valid shares in $\mathbf{S}$, reconstruct $B$ and $R_r \leftarrow \mathsf{Recon}(\mathbf{S})$. Broadcast $(B, R_r)$. On receiving $(B, R_r)$ from others, accept $R_r$ if $R_r = e(B, h_2)$ and $e(B, g_2) = e(g_1, g_2^s)$.
   d) Compute and output $\mathcal{O}_r \leftarrow H(R_r)$.
9) **(Non-blocking) Equivocation.** Broadcast equivocating hashes signed by $L_r$ and stop performing epoch $r$ operations, except Step 8. If $\mathsf{epoch\text{-}timer}_r > 2\Delta$, reset $\mathsf{epoch\text{-}timer}_r$ to $2\Delta$ and start counting down.

Fig. 3: Optimistically responsive random beacon protocol with $O(\kappa n^2)$ bits communication per epoch.

proposals made in the same epoch and an additional commit rule that commits proposals made $t+1$ epochs earlier. A replica commits using the rule that is triggered first. In responsive commit, a replica commits block $B_h$ and all its ancestors immediately when it receives at least $\lfloor 3n/4 \rfloor + 1$ $\mathsf{ack}$ messages for a responsive certificate $\mathcal{C}_r^{3/4}(B_h)$ with a common accumulation value $z_{ra}$ such that its $\mathsf{epoch\text{-}timer}_r$ is large enough ($2\Delta$). Note that a responsive commit happens at the actual speed of the network ($\delta$).

In synchronous commit, when node $p_i$ receives a valid epoch $r$ certificate when its $\mathsf{epoch\text{-}timer}_r$ is large enough ($3\Delta$), it sets $\mathsf{commit\text{-}timer}_r$ to $2\Delta$ and starts counting down. If the received certificate is synchronous i.e., $sc := \langle \mathsf{sync\text{-}cert}, \mathcal{C}_r^{1/2}(B_h), z_{sa}, r\rangle_{L_r}$, it invokes $\mathsf{Deliver}(\mathsf{sync\text{-}cert}, sc, z_{sa}, r)$ and sets $\mathsf{commit\text{-}timer}_r$ to $2\Delta$. When $\mathsf{commit\text{-}timer}_r$ reaches 0, if no epoch-$r$ equivocation has been detected, node $p_i$ commits $B_h$ and all its ancestors. Invoking $\mathsf{Deliver}()$ on the $\mathsf{sync\text{-}cert}$ ensures that all honest nodes have received $\mathcal{C}_r(B_h)$ before quitting epoch $r$.

In addition to above commit rules, we include an additional commit rule. We consider a block $B_\ell$ proposed in epoch $r-t$ proposed by $L_{r-t}$ committed if the highest ranked chain at the end of epoch $r$ extends $B_\ell$ even though none of the blocks that extends $B_\ell$ proposed after epoch $r-t$ have been committed using either of the above commit rules. This commit rule helps in committing safe blocks possibly uncommitted due to responsively moving to higher epoch.

We note that if an honest node commits a block $B_h$ in epoch $r$ using one of the commit rules, it is not necessary that all honest nodes commit $B_h$ in epoch $r$ using the same rule, or commit $B_h$ at all. Depending on how Byzantine nodes behave, only some honest nodes may receive $> 3n/4$ $\mathsf{ack}$ messages and commit using responsive commit rule while some other honest nodes may commit using synchronous commit rule. It is also possible that only some honest node commits $B_h$ while no commit rules are triggered for rest of the honest nodes. For example, an honest node commits a block $B_h$ responsively but all other nodes detect equivocation in the epoch. In such a case, we ensure that all honest nodes receive and lock on a

certificate for $B_h$, i.e., $\mathcal{C}_r(B_h)$, to ensure safety of a commit. Eventually after $t+1$ epochs, all honest nodes will commit $B_h$ using our third commit rule.

**Equivocation.** At any time in epoch $r$, if a node $p_i$ detects an equivocation, it broadcasts equivocating hashes signed by leader $L_r$. Node $p_i$ also stops performing epoch $r$ operations except update, reconstruct and output steps described below. In addition, if $\mathsf{epoch\text{-}timer}_r > 2\Delta$, node $p_i$ resets $\mathsf{epoch\text{-}timer}_r$ to $2\Delta$ to assist in terminating a faulty epoch faster.

**Update.** The update step ensures that the leaders failing to commit a block in $t+1$ epochs are removed the active set of nodes, i.e., if the leader $L_{r-t}$ of epoch $r-t$ fails to add a new block by the end of epoch $r$, $L_{r-t}$ is removed from future proposals, i.e., $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{L_{r-t}\}$. On the other hand, if block $B_\ell$ proposed by $L_{r-t-1}$ has been committed by epoch $r$, update $\mathcal{Q}(L_{r-t})$ with $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ shared in $b_\ell$. This step ensures that our protocol produces a random beacon in each epoch.

**Reconstruct and output.** Node $p_i$ starts to reconstruct aggregated secret $R_r$ when node $p_i$ commits or when its $\mathsf{epoch\text{-}timer}_r$ expires. It obtains $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ corresponding to block committed in $\mathsf{Dequeue}(\mathcal{Q}(L_r))$ and decrypts the share by computing $d_i = c_i^{sk_i^{-1}}$. It then broadcasts $d_i$ to all other nodes. On receiving share $d_j$ from another node $p_j$, it verifies it using $\mathsf{ShVrfy}(g_1, g_2, e_j, d_j)$. On receiving a set $\mathbf{S}$ of $t+1$ valid shares, it reconstructs $R_r \leftarrow \mathsf{Recon}(g_1, g_2, \mathbf{S})$. In addition, it also broadcasts the aggregated secret $R_r$. Any node can verify the correctness of beacon value without reconstruction by checking $e(g_1, R_r) = e(g_1^s, g_2)$. An epoch $r$ beacon output $\mathcal{O}_r$ is the hash of the aggregated secret $R_r$, i.e., $\mathcal{O}_r \leftarrow H(R_r)$.

Observe that the size of aggregated secret $R_r$ is $O(1)$ and all-to-all broadcast of the aggregated secret does not blow up communication. Moreover, the aggregated secret $R_r$ cannot be reconstructed without an honest node sending its secret share. Thus, we use the aggregated secret $R_r$ to synchronize all other nodes and responsively change epochs.

**Latency and communication complexity.** When the epoch leader is Byzantine, not all honest nodes may be locked on a certificate for a common block at the end of the epoch. When the epoch leader is honest, at least one honest node commits block $B_h$ proposed by an honest epoch leader and all honest nodes lock on a certificate for common block $B_h$ and do not act on block proposals that do not extend $B_h$ afterwards. Thus, block $B_h$ and all its ancestors are finalized in an honest epoch. Due to round-robin leader selection, there will be at least one honest leader every $t+1$ epochs and all honest nodes finalize on common blocks up to the honest epoch. Thus, our protocol has a commit latency of $t+1$ epochs. Our protocol has communication cost of $O((\kappa + w)n^2)$ bits per epoch.

**Why is it safe to commit a block $B_\ell$ proposed $t+1$ epochs earlier if the highest ranked chain extends $B_\ell$?** The round robin leader selection policy ensures that there will be at least one honest leader in last $t+1$ epochs. An honest epoch leader $L_r$ ensures it extends the highest ranked block certificate from all honest nodes. Our protocol ensures that the block $B_h$ proposed by the leader $L_r$ is committed by at least one honest node in epoch $r$ and all honest nodes receive and lock on a certificate for block $B_h$. Thus, no honest node acts on the future block proposals that do not extend $B_h$ and the highest ranked chain after epoch $r$ always extends $B_h$, and all its ancestors. This concludes that if block $B_\ell$ proposed $t+1$ epochs earlier is extended by the highest ranked chain, there will never be an equivocating chain that does not extend $B_\ell$ and it is safe commit a block $B_\ell$.

**Optimistically responsive BFT SMR for free.** While our protocol designs an optimistically responsive random beacon protocol, the same protocol can be used as optimistically responsive rotating leader BFT SMR protocol by simply adding additional payload that meets application level validity conditions to $b_h$. Rotating leader protocols provide better *fairness* and *censorship resistance* compared to stable leader protocols [3], [52]. Compared to prior optimistically responsive schemes, our BFT SMR protocol has a communication cost of $O(\kappa n^2)$ *without* using threshold signatures.

An example execution is presented in Fig. 4. Due to space constrains, we present detailed security analysis in the full version of the paper [9].

## VI. Reconfiguration

In this section, we present a reconfiguration scheme for our beacon protocol to restore the resilience of the protocol after some Byzantine nodes have been removed. We adapt the reconfiguration scheme of RandPiper [10] and make efficiency improvements in terms of the number of epochs before a new joining node becomes an active participant in the system. We make following modifications to obtain this efficiency.

A reconfiguration scheme for a synchronous protocol requires new joining nodes to synchronize with all other nodes such that the clocks of all honest nodes differ by at most $\Delta$ time. In RandPiper, they designed a clock synchronization primitive to sychronize the joining nodes. In the clock synchronization primitive, they first secret shared $t+1$ secrets from distinct leaders using verifiable secret sharing (VSS) scheme. To ensure all the nodes agree on the shared secret, the clock synchronization protocol had to be executed for $2t+2$ epochs. The agreed upon $t+1$ secrets were homomorphically combined to obtain a aggregated secret that is used to synchronize new joining nodes. In our protocol, the reconstructed secret is already a aggregated secret combined using $t+1$ secrets from different nodes. Moreover, we are generating and using the aggregated secret to synchronize in every epoch. Thus, the same aggregated secret can be used to synchronize the new joining nodes and avoid the need to execute a separate clock synchronization primitive. In the process, the new joining nodes can become active $2t+2$ epochs earlier than RandPiper. In addition, due to optimistic responsiveness, the length of each epoch is considerably shorter during optimistic conditions and new nodes can join the system much quicker. In this regard, our reconfiguration scheme is strictly better compared to RandPiper.

Observe that in OptRand, nodes in $\mathcal{P} \setminus \mathcal{P}_r$ are rotated in round robin manner and when some node $p_j$ becomes an epoch leader in an epoch, the secrets node $p_j$ shared the last time it became an epoch leader is used. To be specific, the secrets in $\mathcal{Q}(p_j)$ is used. Thus, our reconfiguration scheme needs to ensure that when some node $p_k$ joins the system, all nodes $\mathcal{P} \setminus \mathcal{P}_r$ have $\mathcal{Q}(p_k)$ filled with aggregated PVSS tuple before $p_k$ becomes an epoch leader. We accomplish this by having

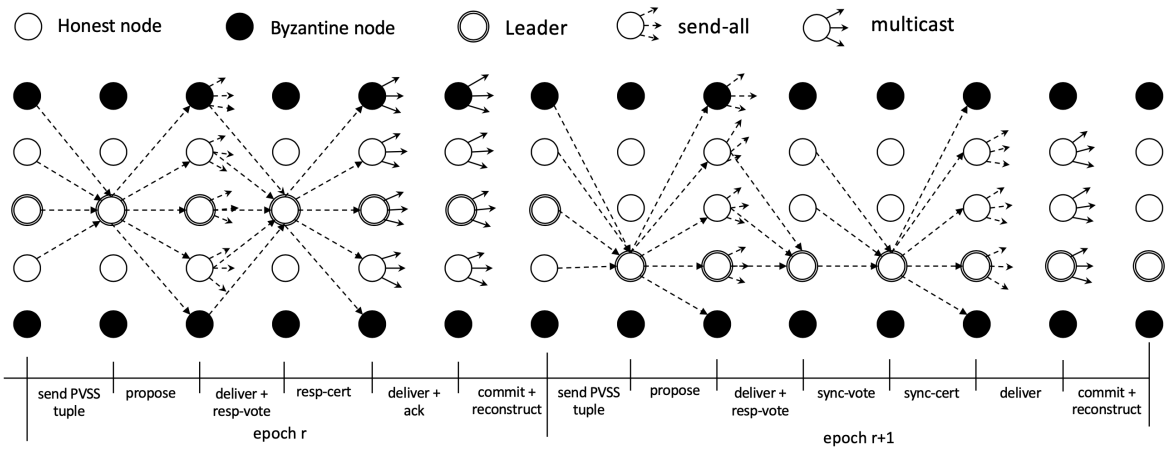



Fig. 4: An example execution of two epochs of OptRand. Here, epoch $r$ is responsive and epoch $r+1$ is synchronous. *send-all* implies a node sending different messages to different nodes. This differs from multicast as multicast involves sending same message to all nodes.

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 set $\mathcal{P} \setminus \mathcal{P}_r$ along with PVSS tuple $(\mathbf{v}_i, \mathbf{c}_i, \widetilde{\pi}_{K,i})$ at the end of some epoch $r'$ in which the inquire request was received. Node $p_k$ waits for at least $t+1$ consistent responses from the same epoch $r'$ and forms an inquire certificate. An inquire certificate is valid if it contains $t+1$ inquire responses that belong to the same epoch $r'$ and contains the same set of active nodes. In addition, node $p_k$ aggregates $t+1$ PVSS tuples to obtain $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ (refer Aggregation Step in Fig. 1).
2) **Join.** Node $p_k$ sends a join request with the inquire certificate and aggregated PVSS tuple $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ to node $p_j$ $\forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$.
3) **Accept.** Upon receiving the join request with valid inquire certificate and aggregated PVSS tuple $(\mathbf{v}, \mathbf{c}, \widetilde{\pi}_K)$ , node $p_i$ checks the validity of the received PVSS tuple (refer Commit Step in Fig. 1). If valid, send $\langle \mathsf{accept}, H(\mathbf{v}, \mathbf{c}), r \rangle_{p_i}$ to node $p_k$.
4) **Accept Cert.** Upon receiving $t+1$ accept messages, node $p_k$ broadcasts the accept certificate to all nodes $\mathcal{P} \setminus \mathcal{P}_r$.
5) **Propose.** Upon receiving the join request with valid inquire certificate, aggregated PVSS tuple and accept certificate, the leader $L_r$ of current epoch $r$ adds the join request containing inquire certificate, PVSS tuple and accept certificate in its block proposal $B_h$ if (i) $L_r$ 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 $r'$.
6) **Update.** If the block $B_h$ with the join request from node $p_k$ proposed in epoch $r$ gets committed by epoch $r+t$, update $n_t \leftarrow n_t - 1$ in epoch $r+t$, update $\mathcal{Q}(p_j)$ with aggregated PVSS tuple $(\mathbf{v}, \mathbf{c})$ and send set $\mathcal{P} \setminus \mathcal{P}_r$ to node $p_k$. Henceforth, node $p_k$ becomes a *passive* node and receives all protocol messages from active nodes.
7) **Synchronize.** The first time node $p_i$ receives a valid aggregated secret $R_{r+t}$, it
   - resets its $\mathsf{epoch\text{-}timer}_{r+t+1}$ to the beginning of epoch $r+t+1$.
   - broadcasts aggregated secret $R_{r+t}$ to all other nodes.

Node $p_k$ becomes an active node when it has $\mathcal{Q}(p_j) \neq \perp \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$.
If node $p_k$ fails to join the system, it restarts reconfiguration process again after some time.

Fig. 5: Reconfiguration protocol

the joining node aggregate $t+1$ PVSS tuple and send it to all nodes $\mathcal{P} \setminus \mathcal{P}_r$ before it can join the system.

The reconfiguration protocol is presented in Fig. 5. 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 decremented by one when a new node joins the system. The value of $n_t$ can be at most $t$.

Node $p_k$ that intends to join the system uses the reconfiguration protocol to join the system. All nodes update $\mathcal{Q}(p_k)$ with the aggregated PVSS tuple provided by node $p_k$ once the join request from node $p_k$ get committed.

Node $p_k$ becomes an *active* node when it has $\mathcal{Q}(p_j) \neq \perp \forall p_j \in \mathcal{P} \setminus \mathcal{P}_r$. This happens when all nodes in $\mathcal{P} \setminus \mathcal{P}_r$ becomes a leader at least once after node $p_k$ joins the system. Due to round robin leader election, node $p_k$ will have a full queue after $n+t+1$ epochs.

Due to space constraints, we present detailed security analysis in the full version [9].

## VII. Performance Evaluation

In this section, we evaluate the performance of our protocols at various system sizes. For OptRand, we evaluate the throughput for fast optimistic mode and slow synchronous mode. For reconfiguration, we evaluate throughput and latency when nodes leave and join the system. We also compare the performance of our protocols with state-of-art synchronous random beacon protocols Drand [28] and RandPiper [10].

### A. Implementation Details

We implement a prototype of OptRand in C++. We also implement our reconfiguration scheme. Our implementation is publicly available at our github repository [53] for artifact evaluation. Our implementation uses the open source implementation of HotStuff [57] and their networking library.

**Instantiation.** We instantiate pairings with the Type-III pairing BN128 [38] family by the team at Zcash. We use the provided default values for $g_1$ and $g_2$. We generate and use config files with the PVSS public keys and secp256k1 [56] public keys for digital signatures. We also pre-populate $n$ PVSS sharing for queue $\mathcal{Q}$.

**Optimizations.** We make the following optimizations:

(1) *Swap groups.* BN128 curve being a Type III pairing, has two different groups $\mathbb{G}_1$ and $\mathbb{G}_2$. A point in $\mathbb{G}_2$ is roughly equal to 2 $\mathbb{G}_1$ points, and the underlying curve for $\mathbb{G}_2$ is more complex than the curve in $\mathbb{G}_1$. Thus, computations in $\mathbb{G}_2$ are more expensive, and our protocol if implemented as is, will be inefficient. We swap the groups in our implementation leading to improved performance.

(2) *Share buffering.* In every epoch $r$, the leader $L_r$ needs to collect $t+1$ PVSS sharings and aggregate them before proposing. This is an $O(nt)$ computational overhead in the critical path of the protocol. We remove this computation from the critical path by buffering the aggregate PVSS vectors for each node. Nodes send PVSS vector for a future epoch $r'$ and the leader $L_{r'}$ will verify and aggregate the PVSS vectors over the course of $n$ epochs. The leader $L_{r'}$ then proposes these aggregated PVSS vectors in epoch $r'$.

(3) *Multi-exponentiation.* Multi-exponentiation is a technique where $\prod g^{x_i}$ for $1 \leq i \leq n$ for $g \in \mathbb{G}$ can be computed efficiently using pre-computation involving $g$ for any $n$ scalars $x_i$. We use this for efficient commitment generation, and for reconstruction by pre-computing tables for $g_1$, $g_2$ and $g_2'$.

(4) *Reduce pairings.* We significantly reduce the overhead of pairings from the critical path of the consensus by replacing a pairing check with a discrete log proof of equality. A pairing check for decryption $e(d_i, g_2) = e(g_1, v_i)$ can be replaced with a discrete log proof of equality $\mathsf{NIZKPK}(sk_i, d_i, e_i)$ thereby reducing the pairing operations performed. Where pairings cannot be avoided, we use a partial pairing optimization. A pairing consists of two almost equally expensive steps: (i) miller loop, and (ii) full exponentiation. To check pairing equalities, it is sufficient to perform two miller loops, and then subtracting the partial results and then finally performing the final exponentiation once and checking if it is $1_{\mathbb{G}_T}$, reducing computational overheads by $\approx 25\%$.

(5) *Merkle trees as cryptographic accumulators*. We use computationally efficient Merkle trees as a cryptographic accumulator instead of bilinear accumulator at the cost of increasing communication complexity by $O(\kappa \log n)$ factor.

### B. Experiments

**Experimental setup.** We evaluate our implementation of OptRand and the baselines on Amazon Web Services (AWS) t3-medium Ubuntu 18.04 virtual machine (VM) with one node per VM. All VMs have two vCPUs and 4 GB RAM. We distributed the $n$ nodes equally across eight different AWS regions: N. Virginia, Ohio, Oregon, N. California, Canada, Ireland, Singapore, and Tokyo. When spawning $n$ nodes, we spawn node 1 in region 1, node 2 in region 2, and so on.

**Baselines.** We compare OptRand with the two state-of-the-art random beacon protocols: BRandPiper [10] and Drand [28].

*Drand.* Drand is a synchronous random beacon protocol inspired by Cachin et al. [14]. It is a synchronous protocol using the unique-signature and random oracle assumptions. It implements the protocol in Golang over BLS-12-381 [13] family of curves. We use their open-source public implementation [27] for comparison.

*BRandPiper.* BRandPiper is a synchronous random beacon protocol offering immediate unpredictability with an amortized communication complexity of $O(f\kappa n^2)$ where $f \leq t$ is the actual number of faults. BRandPiper also has a constant latency of $11\Delta$ per beacon. We used their Rust implementation [39] for comparison.

### C. Random Beacon Performance

Fig. 6 shows the performance of OptRand and the baselines.

*Drand.* Drand uses a parameter *period*; in every period one beacon is produced. Their practical deployment uses a period of 30 seconds using a system consisting of $n = 9$ nodes. To more accurately capture what their system could achieve, we measure a value logged by the protocol called *time discrepancy ms*, which is documented to be the time from the start of the epoch until the beacon is reconstructed. Giving the benefit of doubt, we use 99% percentile of this number across all the nodes to estimate the throughput of the system. Note that the actual performance at this high throughput can vary as the implementation is in Go-lang whose garbage collector can violate synchrony.

We observe that Drand generally has constant throughput, except when $n = 9$. As shown in Fig. 6, in this case, the time discrepancy was lower. We repeated the experiment several times but got the same anomaly. A possible reason for this could be the centralization of the servers towards the American sub-continent due to the presence of 6 nodes.

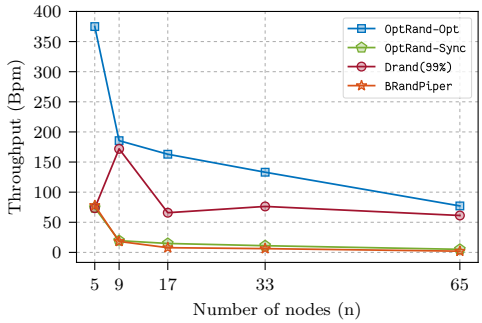



Fig. 6: **Performance of random beacon protocols: Drand and BRandPiper in contrast to OptRand**. Drand uses time discrepancies, i.e., time from the start of a proposal to the time the beacon is produced to indirectly measure throughputs.

*BRandPiper.* BRandPiper produces a beacon every $11\Delta$ similar to OptRand in the slow mode. We run the experiments

for decreasing values of $\Delta$ until no warnings show up in the logs, indicating with high confidence that the chosen $\Delta$ is correct and safe. We use this value of $\Delta$ to estimate the throughput. We observe that as $n$ increases, the throughput drops. This is due to increased synchronization overheads and cryptographic overheads incurred by the round-robin nature of the protocol as a single slow node can slow the system down.

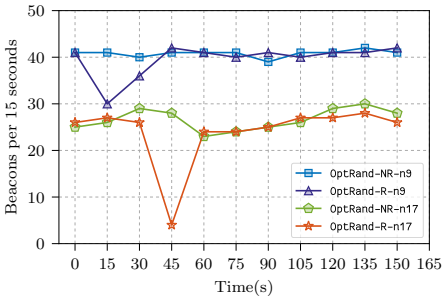



Fig. 7: **Throughput in beacons per 15 seconds during reconfiguration.**

*OptRand.* Compared to both baselines, our protocol generates beacons at significantly higher rate during optimistic conditions (denoted by OptRand-Opt in Fig. 6). This is because our protocol progresses at the network speed while other protocols depend on pessimistic delay $\Delta$ for their progress. The throughput is significantly higher in proportion when $n = 5$. This is because all the nodes are in US and Canada and have low network latency between them.

For the non-optimistic mode (denoted by OptRand-Sync in Fig. 6), our protocol generates a beacon every $11\Delta$ time. To learn the best throughput we can obtain, similar to Drand, we choose value of $\Delta$ where the system does not report any timing errors. In this mode, our protocol performs slower than Drand, as Drand can output a beacon in every round (which is $2\Delta$ time) whereas our protocol outputs a beacon every $11\Delta$ time. The performance of our protocol in non-optimistic mode is similar compared to BRandPiper which also generates beacons every $11\Delta$ time.

### D. Reconfiguration

While reconfiguration-friendliness is discussed by recent random beacon protocols [10], [23], the treatment was theoretical in nature. We implement and measure throughput and latency of reconfiguration in the optimistic path.

There are two important aspects of our implementation. First, while our reconfiguration scheme expects $t+1$ identical responses from the same epoch as a response to the inquiry, in the optimistically responsive mode, we may not get responses from the same epoch. In our implementation, the new joining node just waits for a consistent set of active nodes from $t+1$ nodes, irrespective of the epoch. This is sufficient if the churn in active nodes is not a lot. Second, the new joining node needs to download the entire blockchain from the genesis from the active nodes. In our implementation, this happens *online* causing other nodes to wait until the blockchain has been downloaded. Consequently, to account for this wait and prevent active nodes from removing the new joining node due to delay in proposing, we set $\Delta$ to be slightly larger (e.g., 2 seconds). In practice, this deficiency can be fixed by requiring the joining node to download the blockchain ahead of time.

Additionally, we remove an optimization where nodes send their PVSS transcripts to the leader some $k$ epochs earlier. This is because as the active set of nodes change, the leaders for future epochs can not be determined beforehand. Hence, the leaders need to verify the PVSS transcripts and perform aggregation in the critical path of the protocol; thus reducing the throughput.

In the evaluation, we show latency and throughput when an old node leaves and a new node joins the system. We evaluate the performance of our implementation during reconfiguration and in steady state when no reconfiguration occurs. In the reconfiguration experiment, we first remove an active node after around 100 beacons have been generated and then add a new node. Our evaluation shows that we can seamlessly add and remove nodes without halting the system. Fig. 7 shows the throughput as a function of time under steady state (denoted by OptRand-NR) and when reconfiguration occurs (denoted by OptRand-R) for system sizes of 9 and 17. As shown in Fig. 7, the throughput decreases slightly during reconfiguration, this is because when the old node leaves the system, it does not propose in its epoch and the system cannot progress optimistically; thus incurring $11\Delta$ in an epoch. Additionally, the new joining node needs to download the blockchain before it proposes. Afterwards, the throughput is similar to the steady state.

Fig. 8a shows the throughput with no reconfiguration (denoted by OptRand-NR) and when the reconfiguration occurs (denoted by OptRand-R). As shown in Fig. 8a, the throughput of the system remains consistent before and after reconfiguration as the beacons are generated with minimal interruption. However, for $n = 5$, the throughput reduces slightly. This is because the first 5 active nodes are in US and Canada while the new joining nodes is in Ireland and the node in Ireland has higher network latency.

Additionally, we also measure the time taken for a new joining node to be active from the time it sends its inquire request at various system sizes. As shown in Fig. 8b, the nodes join the system in a matter of seconds. The latency to join the system varies slightly across various system sizes and depends on various factors such network latency of the joining node and the time taken to download the prior blockchain.

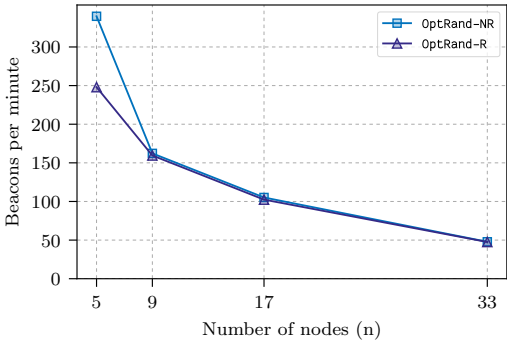
## VIII. Acknowledgements

We thank Sourav Das and Ling Ren for helpful discussions on SPURT [23]. This research was supported in part by grants by VMware Research and Novi Research. This work has been partially supported by the Army Research Laboratory (ARL) under grant W911NF-20-2-0026, the National Institute of Food and Agriculture (NIFA) under grant 2021-67021-34251, and the National Science Foundation (NSF) under grant CNS-1846316.

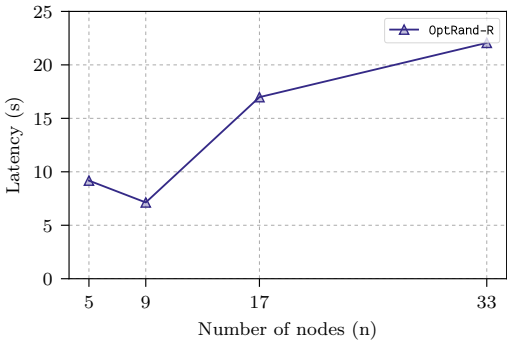
## References

[1] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience," *Financial Cryptography and Data Security (FC)*, 2019.

[2] I. Abraham, D. Malkhi, K. Nayak, and L. Ren, "Dfinity consensus, explored." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 1153, 2018.




(a) Throughput vs $n$




(b) Latency of reconfiguration vs $n$.

Fig. 8: Throughput and latency of reconfiguration at various system sizes.


[3] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 654–667.

[4] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[5] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Optimal good-case latency for byzantine broadcast and state machine replication," *arXiv preprint arXiv:2003.13155*, 2020.

[6] I. Abraham, K. Nayak, and N. Shrestha, "Optimal good-case latency for rotating leader synchronous bft," in *25nd International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2021.

[7] D. Beaver, K. Chalkias, M. Kelkar, L. K. Kogias, K. Lewi, L. de Naurois, V. Nicolaenko, A. Roy, and A. Sonnino, "Strobe: Stake-based threshold random beacons," *Cryptology ePrint Archive*, 2021.

[8] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 27–30.

[9] A. Bhat, N. Shrestha, A. Kate, and K. Nayak, "Optrand: Optimistically responsive distributed random beacons," *Cryptology ePrint Archive*, 2022.

[10] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "Randpiper – reconfiguration-friendly random beacons with quadratic communication," ACM SIGSAC CCS 2021, 2021.

[11] M. Blum, "Coin flipping by telephone a protocol for solving impossible problems," *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.

[12] D. Boneh and V. Shoup, "A graduate course in applied cryptography," 2017, http://toc.cryptobook.us/book.pdf.

[13] S. Bowe, "Bls12-381: New zk-snark elliptic curve construction," *Zcash Company blog, URL: https://z. cash/blog/new-snark-curve*, 2017.

[14] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[15] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *ACM Symposium on Theory of computing (STOC)*, 1993, pp. 42–51.

[16] I. Cascudo and B. David, "Scrape: Scalable randomness attested by public entities," in *International Conference on Applied Cryptography and Network Security*. Springer, 2017, pp. 537–556.

[17] I. Cascudo, B. David, O. Shlomovits, and D. Varlakov, "Mt. random: Multi-tiered randomness beacons," Cryptology ePrint Archive, Report 2021/1096, 2021, https://ia.cr/2021/1096.

[18] "Generate random numbers for smart contracts using chainlink vrf," https://docs.chain.link/docs/chainlink-vrf. [Online]. Available: https://docs.chain.link/docs/chainlink-vrf

[19] T.-H. H. Chan, R. Pass, and E. Shi, "Pili: An extremely simple synchronous blockchain." *IACR Cryptology ePrint Archive*, vol. 2018, p. 980, 2018.

[20] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual international cryptology conference*. Springer, 1992, pp. 89–105.

[21] A. Cherniaeva, I. Shirobokov, and O. Shlomovits, "Homomorphic encryption random beacon." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1320, 2019.

[22] I. T. L. Computer Security Division, "Interoperable randomness beacons: Csrc," https://csrc.nist.gov/projects/interoperable-randomness-beacons. [Online]. Available: https://csrc.nist.gov/projects/interoperable-randomness-beacons

[23] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2502–2517.

[24] D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "Dynamic fault-tolerant clock synchronization," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 143–185, 1995.

[25] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[26] J. Drake, "Minimal vdf randomness beacon. ethereum research post (2018)," https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566.

[27] Drand, "drand." [Online]. Available: https://github.com/drand/drand

[28] ——, "Drand - a distributed randomness beacon daemon," https://github.com/drand/drand. [Online]. Available: https://github.com/drand/drand

[29] P. Feldman and S. Micali, "An optimal probabilistic protocol for synchronous byzantine agreement," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 873–933, 1997.

[30] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.

[31] M. Fitzi and J. A. Garay, "Efficient player-optimal protocols for strong and differential consensus," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003, pp. 211–220.

[32] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Aggregatable distributed key generation," Cryptology ePrint Archive, Report 2021/005, 2021, https://eprint.iacr.org/2021/005, Appearing in EUROCRYPT '21.

[33] M. Haahr, "True random number service," https://www.random.org/. [Online]. Available: https://www.random.org/

[34] R. Han, H. Lin, and J. Yu, "Randchain: Decentralised randomness beacon from sequential proof-of-work," Cryptology ePrint Archive, Report 2020/1033, 2020, https://eprint.iacr.org/2020/1033.

[35] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.

[36] S. Heidarvand and J. L. Villar, "Public verifiability from pairings in secret sharing schemes," in *International Workshop on Selected Areas in Cryptography*. Springer, 2008, pp. 294–308.

[37] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," in *Annual International Cryptology Conference*. Springer, 2006, pp. 445–462.

[38] "scipr-lab/libff: C++ library for finite fields and elliptic curves." [Online]. Available: https://github.com/scipr-lab/libff

[39] Z. Luo, "zhtluo/randpiper-rs," https://github.com/zhtluo/randpiper-rs.

[40] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2111–2128.

[41] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[42] A. Momose, J. P. Cruz, and Y. Kaji, "Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 406, 2020.

[43] A. Momose and L. Ren, "Optimal communication complexity of byzantine consensus under honest majority," *arXiv preprint arXiv:2007.13175*, 2020.

[44] L. Nguyen, "Accumulators from bilinear pairings and applications," in *Cryptographers' track at the RSA conference*. Springer, 2005, pp. 275–292.

[45] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.

[46] "blockchain oracle service, enabling data-rich smart contracts," https://provable.xyz/. [Online]. Available: https://provable.xyz/

[47] M. O. Rabin, "Randomized byzantine generals," in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 1983, pp. 403–409.

[48] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[49] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. Weippl, "Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness," Cryptology ePrint Archive, Report 2020/942, https://eprint.iacr. org/2020/942, Tech. Rep., 2020.

[50] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "Hydrand: Practical continuous distributed randomness," in *2020 IEEE Symposium on Security and Privacy (SP). IEEE*, 2020.

[51] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[52] N. Shrestha, I. Abraham, L. Ren, and K. Nayak, "On the Optimality of Optimistic Responsiveness," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 839–857.

[53] N. Shrestha and A. Bhat, "Optrand implementation." [Online]. Available: https://github.com/nibeshrestha/optrand

[54] N. Shrestha, A. Bhat, A. Kate, and K. Nayak, "Synchronous distributed key generation without broadcasts," *Cryptology ePrint Archive*, vol. 2021, p. 1635, 2021.

[55] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2017, pp. 444–460.

[56] B. Wiki, "Secp256k1," *Accessed: Feb*, vol. 11, 2016.

[57] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

## APPENDIX A
## EXTENDED PRELIMINARIES

**co-Decisional Bilinear Squaring assumption.** This is a modified version of the symmetric pairing based assumption in SCRAPE and related works [16], [36]. We formally show that this is the correct generalization of the DBS assumption for Type-III pairings by showing that it implies the *Decisional Bilinear Diffie Hellman (DBDH)* assumption in Definition A.2.

**Definition A.1** (co-Decisional Bilinear Squaring (co-DBS) Assumption)**.** *Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficient pairing scheme, with $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ being two independent generators. We say that the co-DBS assumption holds if the following is true for any PPT adversary $\mathcal{A}$:*

$$\Pr\left[\begin{array}{c|c} \alpha, \beta, \gamma \leftarrow_{\$} \mathbb{Z}_q, b \leftarrow_{\$} \{0,1\}, & \\ u_1 \leftarrow g_1^{\alpha}, u_2 \leftarrow g_2^{\alpha}, & \\ v_1 \leftarrow g_1^{\beta}, & \\ T_0 \leftarrow e(g_1, g_2)^{\alpha^2 \beta}, & \\ T_1 \leftarrow e(g_1, g_2)^{\gamma}, & \\ b' \leftarrow \mathcal{A}(g_1, g_2, u_1, u_2, v_1, T_b) & b' = b \end{array}\right] \leq \mathsf{negl}(\kappa)$$

Prior works such as SCRAPE [16] define the problem in the symmetric pairing setting where $\mathbb{G}_1 = \mathbb{G}_2$. However, no known explicit construction exists for the asymmetric pairing definitions, although most of these works argue that the generalization is easy. SCRAPE and its sources [36] mention that it is easy to generalize it to the Type III pairings. In this work, we make explicit the assumption and show implication to the known *Decisional Bilinear Diffie Hellman (DBDH)* assumption.

**Definition A.2** (Decisional Bilinear Diffie Hellman (DBDH) Assumption [12])**.** *Let $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficient non-degenrate Type-III pairing. Let $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ be two independent generators. The assumption is said to hold if for any PPT adversary $\mathcal{A}$ the following is true:*

$$\Pr\left[\begin{array}{c|c} \alpha, \beta, \gamma, \delta \leftarrow_{\$} \mathbb{Z}_q, b \leftarrow_{\$} \{0,1\} & \\ u_1 \leftarrow g_1^{\alpha}, u_2 \leftarrow g_2^{\alpha} & \\ v_1 \leftarrow g_1^{\beta}, w_2 \leftarrow g_2^{\gamma} & \\ T_0 \leftarrow e(g_1, g_2)^{\alpha\beta\gamma} & \\ T_1 \leftarrow e(g_1, g_2)^{\delta} & \\ b' \leftarrow \mathcal{A}(g_1, g_2, u_1, u_2, v_1, w_2, T_b) & b' = b \end{array}\right] \leq \mathsf{negl}(\kappa)$$

**Lemma 2.** *The co-DBS assumption implies the DBDH assumption.*

*Proof: Given an instance of co-DBS ($g_1$,$g_2$,$u_1$,$u_2$,$v_1$,$T_b$), we can construct a correct DBDH instance using $(g_1, g_2, u_1, u_2, v_1, w_2 \leftarrow g_2^{\gamma}, T_b' \leftarrow T_b^{\gamma})$ for a randomly chosen $\gamma \leftarrow_{\$} \mathbb{Z}_q$. If $b = 0$, then it is the correct instance of DBDH with $T_0' = e(g_1, g_2)^{\alpha\beta\gamma}$. If $b = 1$, then it is the correct instance of DBDH with $T_1' = e(g_1, g_2)^{\gamma'\gamma}$ where $\gamma'$ originated from $T_1$ in the original co-DBS instance.* ∎

**Chaum-Pedersen Scheme for** NIZKPK**.** Concretely,

Let $(g, u) \in \mathbb{G}^2$ be public values with $u \leftarrow g^s$ for some $s \in \mathbb{Z}_q$. A prover $P$ runs the following interactive protocol:
(1) $P$ first sends to $V$, the values $a \leftarrow g^w$ for a randomly drawn $w \leftarrow_{\$} \mathbb{Z}_q$.
(2) The verifier $V$ chooses a random $c \leftarrow_{\$} \mathbb{Z}_q$, and sends $c$ to the prover $P$.
(3) The prover sends $r \leftarrow w + cs$ to $V$.
(4) The verifier checks if $g^r = au^c$ and outputs the result.

Fig. 9: Interactive discrete log Proof of Knowledge protocol for NIZKPK

Using Fiat-Shamir heuristic [30], we transform this into a non-interactive proof (assuming Random Oracle Model) by setting $c \leftarrow H(u, a)$ and proof $\pi \leftarrow (a, r)$.

TABLE II: **Summary of notation.**

| Symbol | Meaning |
|---|---|
| $\mathcal{A}$ | PPT Adversary |
| $b$ | Number of data shards in RS |
| $C$ | Linear Error Correcting Code |
| $\mathcal{C}$ | Challenger for cryptographic games |
| $C^{\perp}$ | Dual of the Linear Error Correcting Code space $C$ |
| $c$ | SCRAPE Ciphertexts/Encryptions |
| $d$ | SCRAPE decryptions |
| $\mathcal{D}$ | Distinguishing adversary |
| NIZKPK | Alg. to prove knowledge $x$ satisfying $g^x = u$. |
| Verify() | Algorithm that verifies the above relation |
| $\delta$ | The actual network speed |
| $\Delta$ | The worst case maximum network delay |
| $e$ | Type III Bilinear Pairing function |
| $f$ | Actual number of faults in the system |
| $\mathbb{G}_1$,$\mathbb{G}_2$,$\mathbb{G}_T$ | The pairing groups of $e$ |
| $H$ | The random oracle |

| Symbol | Meaning |
|---|---|
| $\lambda$ | The security parameter |
| $\kappa$ | The normalized message size |
| $\langle m \rangle_{p_i}$ | A message $m$ along with a signature by node $p_i$ |
| negl() | A negligible function |
| $\mathcal{O}_r$ | The beacon output for round $r$ |
| $pk$ | SCRAPE Public Keys |
| $n$ | Total number of nodes |
| $\mathcal{P}$ | The set of all the nodes in the system |
| $p_i$ | The i$^{\text{th}}$ node of the system |
| $r$ | Rounds of the Random Beacon Protocol |
| $sk$ | SCRAPE Secret Keys |
| $q$ | The prime order of all the pairing groups |
| $t$ | Max. number of faults tolerated in the system |
| $\tau$ | Time instants |
| $v$ | SCRAPE Commitments |
| $\mathbb{Z}_q$ | The scalar field of all the groups of $e$ |

**Note.** This can be easily extended to prove that given $(g_1, g_2, u_1, u_2) \in \mathbb{G}^4$, a Prover knows $s$ such that $g_1^s = u_1$ and $g_2^s = u_2$ by duplicating all the steps except generating the challenge $c$. The challenge can be generated together as $c \leftarrow H(u_1, u_2, a_1, a_2)$. The same technique can also be used to prove equality of discrete logarithms, i.e., $\log_{g_1} u_1 = \log_{g_2} u_2$.

## APPENDIX B
## SECURITY ANALYSIS, IND1-SECRECY

Indistinguishability of secrets (IND1-Secrecy) refers to notion that when the dealer of a PVSS scheme is honest, the (computational) adversary does not learn any information about the secret. Formally, it is defined by Heidarvand et al. [36] in the game defined in Definition B.1. It assumes that a PVSS scheme consists of the following four phases: (i) Setup, (ii) Distribution, (iii) Verification, and (iv) Reconstruction.

**Definition B.1** ((IND1-secrecy) Indistinguishability of secrets [36])**.** *We say that an $(n, t+1)$ threshold PVSS scheme is IND1-secret if any PPT adversary $\mathcal{A}$ has a negligible advantage in the following game played against a challenger $\mathcal{C}$. During the game, $\mathcal{A}$ can corrupt a new node at any time, but up to $t$ nodes in total. When $\mathcal{A}$ corrupts a node, he receives his secret key (only after the setup). A list of corrupted nodes is maintained during the game.*

1. *$\mathcal{C}$ runs the setup subprotocol and sends the public parameters to $\mathcal{A}$ along with the public keys of still uncorrupted nodes. $\mathcal{C}$ stores the secret keys of those nodes.*
2. *$\mathcal{A}$ sends the public keys of already corrupted nodes.*
3. *$\mathcal{C}$ picks two random secrets $x_0$, $x_1$ and a random bit $b \in \{0, 1\}$. Then $\mathcal{C}$ runs the distribution subprotocol for secret $x_0$ and sends all the resulting information to $\mathcal{A}$, along with $x_b$.*
4. *$\mathcal{C}$ runs reconstruction subprotocol for the set of all uncorrupted nodes and sends all the messages exchanged via public channels (if any) to $\mathcal{A}$. No new corruptions are allowed from this point.*
5. *$\mathcal{A}$ outputs a guess bit $b'$.*

*The advantage of the adversary $\mathcal{A}$ in this game is defined as $|\mathsf{Prob}[b' = b] - \frac{1}{2}|$.*

**Note.** In this work, we will assume a static variant of this game where the adversary $\mathcal{A}$ corrupts up to $t$ nodes before Step 1. of the game.

### A. Proof of IND1-Secrecy of our modified PVSS

**Theorem 3** (IND1-Secrecy for sharing)**.** *Assuming that the hash function $H$ is random oracle and that co-DBS assumption holds, the protocol in Fig. 1 achieves IND1-Secrecy for sharing against any $t-$bounded PPT adversary $\mathcal{A}$.*

*Proof:* We show that if a $t$-bounded static PPT adversary $\mathcal{A}$ has a non-negligible advantage $\varepsilon_{\mathcal{A}}$ in breaking the IND1-secrecy of our protocol in Fig. 1, then there exists a PPT adversary $\mathcal{A}_{DBS}$ that has a non-negligible advantage in breaking the co-DBS assumption.

The $\mathcal{A}_{DBS}$ simulates our modified sharing to $\mathcal{A}$ when given an instance of co-DBS $(g_1, g_2, u_1 \leftarrow g_1^{\alpha}, u_2 \leftarrow g_2^{\alpha}, v_1 \leftarrow g_1^{\beta}, T_b)$ as follows:

1. $\mathcal{A}_{DBS}$ sends $v_1 \in \mathbb{G}_1$ and $g_2, u_2 \in \mathbb{G}_2$ as the generators for the group.
2. The static adversary $\mathcal{A}$ corrupts up to $t$ nodes and sends their public keys. WLOG, we assume that the corrupted nodes have indices $1 \leq j \leq t$.
3. $\mathcal{A}_{DBS}$ sends the public keys for the honest nodes $pk_i \leftarrow g_1^{R_i}$ for $t+1 \leq i \leq n$, where $r_i \leftarrow_{\$} \mathbb{Z}_q$. (This is equivalent to setting the secret key $sk_i = R_i/\beta$.)
4. For $1 \leq i \leq t$, $\mathcal{A}_{DBS}$ sets $v_i \leftarrow g_2^{r_i}$ and $c_i \leftarrow pk_i^{r_i}$ without knowing $\alpha$ since it knows $u_2 = g_2^{\alpha}$. For $t+1 \leq i \leq n$, $\mathcal{A}_{DBS}$ sets $\mathbf{v}$ using Lagrange interpolation of a polynomial using $u_2 = g_2^{\alpha}$; and sets $c_i = v_1^{R_i P_{\alpha}(i)}$ where $P_{\alpha}(i)$ is an $(n, t+1)$ polynomial evaluating to $\alpha$, which is constructed by first producing $v_1^{P_{\alpha}(i)}$ using $r_i$ used for $c_i$ for $i \in [t]$, and then raising to the secret key.

5. For the NIZK proof $\mathsf{NIZKPK}(\alpha, g_2, v \leftarrow g_2^{\alpha})$, $\mathcal{A}_{DBS}$ chooses $r, c \leftarrow_{\$} \mathbb{Z}_q$, sets $a \leftarrow g_2/v^c$ by simulating the random oracle and setting $c \leftarrow H(g_2, v, a)$ and outputs proof $\pi := (a, r)$.
6. Finally, $\mathcal{A}_{DBS}$ sends $T_b$ to $\mathcal{A}$.
7. $\mathcal{A}$ outputs a bit $b'$.

If $b' = 1$, $\mathcal{A}_{DBS}$ outputs $b = 1$, i.e., that $T_b = T_1$ a random element in $\mathbb{G}_T$.

Observe that this is a secret sharing of $e(v_1^{\alpha}, u_2) = e(g_1, g_2)^{\alpha^2\beta}$ and the information sent by $\mathcal{A}_{DBS}$ is distributed exactly like an actual secret sharing instance. When given $T_0$, the adversary $\mathcal{A}$ can detect the correct sharing with non-negligible probability $\varepsilon_{\mathcal{A}}$, and with the same probability, $\mathcal{A}_{DBS}$ can make a correct guess with probability $\varepsilon_{\mathcal{A}}$.

Thus, if our scheme is not IND1-secret then we can break the co-DBS assumption, leading to a contradiction. ∎

## APPENDIX C
## POSTPONED CRYPTOGRAPHIC PROOFS

### A. Proof for Warm-up Beacon Security

We use RO to refer to Random Oracles. We capture the security of our protocol in the following theorem:

**Theorem 4** (Warm-up beacon Fig. 1)**.** *Assuming RO and the co-DBS assumptions hold, the protocol in Fig. 1 is secure as per Definition IV.1.*

*Proof: Weak agreement.* follows trivially due to the guarantees of the broadcast channel. The coding check and the failure of the digital signatures introduces a negligible probability that an honest node may accept an invalid share.

*Validity.* also follows from the construction, and correctness follows from existing works [16], [20].

*Value-validity* Assume an adversary exists that can violate this property, i.e., it can make an honest node output a value that is not uniform.

Any $t$-bounded adversary must select $t + 1$ valid secret sharings. The final vector (from which an honest node outputs the beacon) must satisfy:

(i) *Discrete log equality.* For any $j \in [n]$, the combined commitment $v_j \in \mathbf{v}$ with respect to $g_2$ has the same discrete log as the combined encryption $c_j \in \mathbf{c}$ with respect to $g_1$.

(ii) *Unique degree-t polynomial.* With high probability $(1 - 1/q$, where $q$ is the order of the groups), due to coding scheme used from SCRAPE [16], we know that the polynomial $P$ encoded in $\mathbf{c}$ when reconstructed using any $t+1$ decryption will reconstruct to a unique secret $S = e(g_1^s, g_2')$.

Let $P$ be the polynomial in the commitments. We know that the $(n, t + 1)$ sharing is a valid degree $t$ polynomial, and that the product of $g_1^{p_j(0)}$ and $t$ other values produce $g_1^{P(0)}$. We know that $P(0) = p_j(0) + X$, where $X$ can be known by the adversary if it picks its own $t$ sharings.

Let some adversary $\mathcal{A}$ construct $P(0)$ such that all the checks pass but $P(0)$ is not a function of some $s_j$, and $g_1^{s_j}$ knowledge proof along with $t$ others are included in the set $\overline{\boldsymbol{\pi}}$. We know that $\prod_{i \in \overline{\boldsymbol{\pi}}} g_1^{s_i} = g_1^{P(0)}$. So we get a contradiction that $P(0)$ is not a function of $g_1^{s_j}$. Intuitively, the only way to remove $g_1^{s_i}$ is having knowledge of it or randomly but its probability is negligible.

Since no $t-$bounded adversary can know $P(0)$, $P$ which is a function of $s_j$ is also unpredictable for any $t-$bounded adversary. From Theorem 3, we know that this is not the case, leading to a contradiction.

From the decomposition proof, we know that the final polynomial being shared (from which $\mathcal{B}$ is derived) contains contributions from at least one honest node whose input is uniformly random. Therefore, the value-validity condition holds. The only way an adversary can bias this distribution is by learning some information about the value shared by some honest node. ∎

### B. Proof for Beacon Unpredictability

**Theorem 5** (Beacon unpredictability)**.** *Assuming RO and the co-DBS assumptions hold, the protocol in Fig. 1 produces an unpredictable beacon.*

*Proof:* From Theorem 4, we know that in the broadcast channel world, it guarantees *Value validity*. In OptRand, we realize this and counter the weak-agreement property by making sure the beacon is constructed from a random sharing proposed by the same leader last time.

Formally, to prove security via reduction in Definition III.2, any adversary that breaks the security of OptRand must violate synchrony, can only do so by breaking the cryptography, which remains secure with high probability.

Concretely, any adversary that wins the game defined in Definition III.2 can be simulated to by a simulator to either break Theorem 3 or the underlying digital signature scheme. The adversary $\mathcal{A}$ outputs $b'$ with probability $\varepsilon_{\mathcal{A}} = \varepsilon_{DS} + \varepsilon_{IND1}$, where $\varepsilon_{DS}$ is the probability of breaking the digital signature scheme (say EU-CMA), and $\varepsilon_{IND1}$ is the probability of breaking the IND1-secrecy. If $\varepsilon_{\mathcal{A}}$ is non-negligible, then it must be the case that it has learned a sharing of one of the honest nodes, which we use to win the co-DBS game (similar to the IND1-secrecy) simulation, implying $\varepsilon_{IND1}$ is non-negligible, or win the digital signature security, implying $\varepsilon_{DS}$ is non-negligible. ∎