Leveraging Democracy to Optimize Distributed Random Beacons

Alejandro Ranchal-Pedrosa[†]
Protocol Labs and University of Sydney
Sydney, Australia
alejandro.ranchalpedrosa@sydney.edu.au

Vincent Gramoli
University of Sydney and Redbelly Network
Sydney, Australia
vincent.gramoli@sydney.edu.au

ABSTRACT

Random beacons, protocols that produce a reliable source of randomness, are crucial in a variety of applications. However, solving the random beacon problem has recently been shown to be at least as hard as solving consensus. In this work, we propose Kleroterion, a random beacon protocol that builds on top of recent works in order to ensure a trustless setup that is not costly, and that tolerates up to less than a third of Byzantine processes under partial synchrony. Kleroterion executes n instances of Pinakion, our novel Publicly-Verifiable Secret Sharing (PVSS) protocol, in order to share one input per process. Then, Kleroterion runs a consensus protocol that selects and aggregates a third of these shared inputs.

Compared to previous works that are also quadratic in the communication complexity, Kleroterion allows for inputs to be shared without having to be routed through a specific node, a so-called leader. We refer thus to Kleroterion as a democratic protocol. We show that democratizing protocols improves both communication and computation performance, in that shared bits and computation are scattered across all channels and processes, thus removing the bottleneck at the leader. This is shown in that Kleroterion has linear computation complexity and a number of bits sent per channel of the network independent of the number of processes, except for the reconstruction phase and for one message per leader during agreement. Contrary to leader-less protocols, Kleroterion has a leader of the embedded consensus protocol that proposes a bitmask referencing one bit per shared input. This bitmask can thus reference more information shared by processes, enabling batching with other information. An example of this is a blockchain application in which the output of the random beacon can be used for a secure committee sortition protocol, and the bitmask references both a set of proposed blocks of transactions and of shared inputs.

† The author was with the University of Sydney, Sydney, Australia. He is now with Protocol Labs.

ConsensusDay '22, November 7, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9879-4/22/11…$15.00
https://doi.org/10.1145/3560829.3563558

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**; • **Security and privacy → Distributed systems security**; • **Theory of computation → Cryptographic protocols**.

KEYWORDS

Blockchain, consensus, random beacon, secret sharing, leaderless

1 INTRODUCTION

A reliable public source of randomness, also known as a random beacon, plays an important role in a variety of applications. Some applications like traditional auctions and lotteries [29] or electronic voting allow for a centralized process to compute the random output. However, these applications are vulnerable to a centralized point of failure. Furthermore, many other applications such as solving consensus under asynchrony [48], committee sortition [2, 4, 28, 36], sharding [30, 45], or anonymous communications, cannot fulfil their purposes by relying on a centralized source of randomness, and require the implementations of distributed random beacons.

To implement a distributed random beacon, a set of processes of size n, the committee, execute a protocol that produces the random beacon. These protocols must satisfy the properties of availability, in that they must produce an output; public verifiability, in that every process must be able to verify the output; bias-resistance, in that the output must not be biased by an adversary; and unpredictability, in that the adversary must not be able to predict any non-trivial information about the output of the beacon, or any future output.

The goal of distributed random beacons is thus to generate random outputs in the presence of an adversary. On the one hand, some works rely on synchronous random beacon protocols [1, 3, 8, 11, 13, 20, 23, 39, 42, 44, 49, 56, 57, 63], which are known to be vulnerable to network delays, which has already resulted in the loss of funds from honest processes in some applications of these works [41, 53, 67]. On the other hand, asynchronous protocols do not deterministically satisfy availability.

We therefore rely on partial synchrony, a weak form of synchrony that provides deterministic guarantees, in which messages can be delayed by up to a bound that is unknown. The state of the art already provides random beacons under partial synchrony that tolerate the resilient-optimal bound of $f < n/3$ Byzantine faults [27, 44, 61]. This is because it has recently been shown that if a protocol solves the random beacon problem, then it solves the consensus problem [44].

Our result. Inspired from this recent result that relates random beacons with consensus protocols, we propose Kleroterion, the first democratic protocol that solves the random beacon problem. We refer to a consensus protocol being democratic in that inputs are distributed, i.e. scattered around the pairwise network channels, instead of routed through a specific process, typically known as the leader. Kleroterion extends SPURT's [27] random beacon by making it democratic. The SPURT random beacon protocol relies on a leader to aggregate inputs and propose a digest of these inputs in order to reduce its bit complexity. Contrary to SPURT, Kleroterion asks each process to locally generate n inputs, and aggregate their own inputs locally, to then broadcast them. This allows Kleroterion to relieve SPURT's strong dependency on the leader computation and communication, as seen in Table 1, resulting in as little as a number of bits exchanged per each pair of processes per decision that is independent of the number of processes running the protocol, except for one message sent by the leader and for the reconstruction phase.

Table 1: Comparison of bits sent per each pairwise channel of the network and computation performed by each process, per decision, in SPURT [27] and Kleroterion.

	Phase	Computation		Bits per channel	
		Leader	Non-leader	Leader	non-leaders
SPURT	Commitment	$O(n)$	$O(n)$	$O(\lambda n)$	$O(\lambda)$
	Aggregation	$O(n^2)$	-	-	-
	Agreement	$O(n)$	$O(n)$	$O(\lambda n)$	$O(\lambda)$
Kleroterion	Commitment	$O(n)$	$O(n)$	$O(\lambda)$	$O(\lambda)$
	Aggregation	$O(n)$	$O(n)$	$O(\lambda)$	$O(\lambda)$
	Agreement	$O(n)$	$O(n)$	$O(n)$	$O(\lambda)$

Kleroterion also presents interesting observations compared to more recent leader-less protocols [26, 27, 59, 60, 62, 65]. The majority of leader-less protocols that we know of, like Kleroterion, start by having processes exchange their inputs through an all-to-all broadcast. However, unlike Kleroterion, these protocols then execute one binary consensus instance per input, resulting in a bit complexity of at least $\Omega(n)$ for the combined execution of all binary consensus instances (since there are $\Omega(n)$ binary executions, each exchanging at least $\Omega(1)$ bits). By contrast, Kleroterion proceeds instead by executing a leader-based consensus to simply propose a digest of a bitmask, providing a proposal of size $O(\lambda)$ bits, where $\lambda \leq n$ is the security parameter. Although we believe that it is possible to create implementations of leader-less protocols that benefit from a similar optimization, Kleroterion emphasises the importance of democratizing protocols, and not necessarily removing its leader altogether.

In order to present Kleroterion, we first present our publicly-verifiable secret sharing (PVSS) scheme, Pinakion[1]. Kleroterion then consists of the integration of n parallel executions of the Pinakion protocol, in order to share encrypted secrets, followed by a consensus protocol, which decides on encrypted secret shares. After deciding on at least $n/3$ proposed secretly shared inputs, processes reconstruct each of them with the reconstruction procedure of the Pinakion protocol, in order to aggregate them into the random output.

To the best of our knowledge, our Kleroterion protocol competes with recent works that solve the random beacon problem under partial synchrony by offering a quadratic bit complexity per decision. However, thanks to the democratization of the consensus protocol, Kleroterion offers two advantages. First, whereas previous works that are not democratic saturate network channels to the leader by sending $O(\lambda n)$ bits through them (where λ is the security parameter), our democratic implementation exploits all channels of the network, sending only $O(\lambda)$ through each pairwise channel. This distribution of information scattered around processes also distributes its computation and verification, which removes the bottleneck caused by overusing the leader. This has been shown to be a significant performance improvement [26, 27, 59, 60, 62, 65].

Second, since we decouple the dissemination of secrets from the consensus proposal, but instead the consensus proposal is a bitmask referencing the previously shared secrets, the consensus leader allows for the consensus proposal to reference not just the secret shared by a process, but any other additional information shared by the same process, which enables batching. This is particularly meaningful in applications like blockchains, where consensus is typically used to decide on blocks of transactions. Hence, the cost of adding a random beacon (e.g. for committee sortition) in a blockchain application that uses our proposed protocol is only the cost of disseminating the secrets shared from our Pinakion PVSS scheme.

The rest of this paper is structured as follows: in Section 1.1 we detail the related work, Section 2 sets the background and model, we present our Pinakion PVSS scheme in Section 3, we use Pinakion to present our Kleroterion random beacon protocol in Section 4, to then illustrate optimizations and observations that justify designing democratic protocols and using Kleroterion intead of other works in Section 5, and we finally conclude in Section 6.

1.1 Related work

PVSS. Kokoris Kogias et al. [43] present a high-threshold asynchronous verifiable secret sharing scheme with a dual $(f, 2f+1)$-threshold where the reconstruction threshold is some k for $t+1 < k \leq 2t+1$, with $t = \lceil n/3 \rceil - 1$. This way, the secret can only be reconstructed if k processes participate in the reconstruction, while allowing honest processes that did not participate in the sharing phase to recover their share with the help of $t+1$ other processes. Alhaddad et al. [7] propose an asynchronous verifiable secret sharing (AVSS) protocol with optimal communication complexity in the same model. Tomescu et al. [9] propose the first PVSS with share recovery solution with an optimistic constant number of cryptographic operations per process, while Trek et al. [66] offer a resilient optimal asynchronous complete secret sharing protocol of quasi-linear computation and communication complexity. Boyle et al [13] present a synchronous VSS protocol that tolerates $f < n/3$

[1]The names Kleroterion and Pinakion are inspired from the first known democracy in history, that of the Athenian polis. This democracy consisted on a yearly randomized dictatorship. Kleroterion was the randomization device on to which each citizen would place a token with their name, known as a Pinakion.

Byzantine faults, while Schindler et al [57] present a PVSS protocol under the same model and fault-tolerance.

Random Beacons. Syta et al. [61] propose a random beacon tolerant to t Byzantine faults in asynchrony relying on a setup based on distributed key generation (DKG). Contrary to random beacons with a setup based on common reference string (CRS) like Kleroterion, those relying on DKG require executing the setup phase after every change in the committee. Das et al. [27] propose a PVSS protocol based on a $(t+1,n)$-threshold secret sharing scheme, with a communication complexity of $O(\lambda n^2)$ per beacon output. Algorand [36], Ouroboros Praos [28], Elrond [33], or Polkadot [2, 4] implement random beacons based on verifiable random functions (VRFs) that tolerate up to t Byzantine faults. Algorand [36] relies on a weak synchrony assumption for safety, and a strong synchrony assumption for liveness. While strong synchrony refers to the classical synchronous definition where there is a known bound for the communication delay, weak synchrony states the need for synchronous periods of length s (e.g. hours) for every non-synchronous periods of length b (e.g. a day). Aleph et al. [37] use DKG in order to implement an asynchronous randomness beacon while tolerating $f \leq t$ Byzantine faults, removing the requirement of a trusted dealer setup present in the HoneyBadger's common coin [48]. Gao et al. [35] propose a common coin with $O(\lambda n^3)$ bits and constant asynchronous rounds in order to solve asynchronous Byzantine agreement. RandSolomon [31] propose a random beacon tolerating t Byzantine faults with linear message complexity.

Synchronous random beacon protocols [1, 3, 11, 13, 20, 23, 39, 42, 44, 49, 56, 57, 63] also range from a variety of primitives and setup assumptions, but all of them are tolerant to at most $n/2$ Byzantine faults. Protocols based on Proof-of-Delay rely on strong and new assumptions about verifiable time-lock puzzles or verifiable delay functions [10, 21, 56].

Some works implement random beacons, or varieties of the random beacon problem, under a myriad of denominations, such as randomness beacons [37], global coins [6, 19], common coins [35, 43, 48], random number generators [31, 44], or coin tossing protocols [13, 20].

We highlight state-of-the-art random beacons and compare them with Kleroterion in Table 2, where we show that Kleroterion. We will detail in Section 4 our Kleroterion protocol.

2 BACKGROUND & MODEL

Let $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ be cyclic groups of prime order q and $\mathbb{Z}_q$ the group of integer modulo q, and let λ be the security parameter $\lambda = \log_2(q)$. We assume that at the start of the protocol, all processes agree on public parameters g_0, $h_0 \in \mathbb{G}_0$ and g_1, $h_1 \in \mathbb{G}_1$, which are randomly and independently chosen generators of each cyclic group [27]. This is known as a *common reference string* (CRS) setup. We denote an element x sampled uniformly at random from a finite set $\mathcal{M}$ by $x \overset{\$}{\leftarrow} \mathcal{M}$. We denote vectors using bold face lowercase letters such as $\mathbf{y}$.

We consider a partially synchronous communication network, in which there is a known bound Δ on the communication delay that will hold after an unknown Global Stabilization Time (GST) [32]. Processes communicate via secure and authenticated pairwise communication channels, meaning that messages cannot be modified, lost or duplicated. We iteratively execute our random beacon protocol, Kleroterion, with a static committee N of size $|N| = n$. We assume a standard public-key infrastructure (PKI) that associates processes' identities with their public keys, and that is common to all processes, only for the setup.

Adversary. We denote $t = \lceil n/3 \rceil - 1$ as the maximum number of tolerated Byzantine faults. The adversary $\mathcal{A}$ thus controls $f \leq t$ Byzantine processes. We model processes as probabilistic polynomial-time interactive Turing machines (ITMs) [16, 17, 46]. A process that is not Byzantine is honest.

Bilinear Pairings. Similarly to previous work [27], we rely on the decisional *bilinear Diffie-Hellman* assumption [12] (DBDH), for which we assume the reader is familiar with the standard definition of computationally indistinguishable distribution ensembles [38, 64]:

Definition 2.1 (Bilinear Pairing)**.** Let $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A pairing is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ satisfying the following properties:

(1) bilinearity: For all u, $u' \in \mathbb{G}_0$ and v, $v' \in \mathbb{G}_1$ we have:

$$e(u \cdot u', v) = e(u, v) \cdot e(u', v), \text{ and} \tag{1}$$

$$e(u, v \cdot v') = e(u, v) \cdot e(u, v') \tag{2}$$

(2) non-degeneracy: $g_T = e(g_0, g_1)$ is a generator of $\mathbb{G}_T$.

We refer to $\mathbb{G}_0$ and $\mathbb{G}_1$ as the pairing groups or source groups, and refer to $\mathbb{G}_T$ as the target group.

Definition 2.2 (Decisional bilinear Diffie-Hellman)**.** Given pairing groups G_0, G_1, target group G_T, each of size q, let $e : G_0 \times G_1 \rightarrow G_T$ be an efficient bilinear pairing map. For generators $g_0 \in G_0$, $g_1 \in G_1$, random values $\alpha, \beta, \gamma, \delta \overset{\$}{\leftarrow} Z_q$ and $a_0 \leftarrow g_0^{\alpha}, a_1 \leftarrow g_1^{\alpha}, b_0 \leftarrow g_0^{\beta}, b_1 \leftarrow g_1^{\gamma}$, the following distributions D_0 and D_1 are computationally indistinguishable:

$$D_0 = (a_0, a_1, b_0, b_1, e(g_0, g_1)^{\alpha\beta\gamma}),$$
$$D_1 = (a_0, a_1, b_0, b_1, e(g_0, g_1)^{\delta}).$$

In order to implement the Kleroterion protocol, we will use a variant of Shamir's threshold secret sharing [58] to implement a variant of the publicly-verifiable secret sharing (PVSS) Π_{DBDH} protocol [27]. We define threshold secret sharing here below, and our PVSS, Pinakion, in Section 2.3.

Zero Knowledge Proof of Knowledge. Our Kleroterion protocol uses zero-knowledge proofs about equality of discrete logarithms in order to satisfy *knowledge soundness*, in that processes know the secret that they each are sharing. This guarantees to honest processes that their shared secrets are independent of any other shared secret.

In particular, given a CRS setup as mentioned above, $x \in \mathbb{G}_0$, $y \in \mathbb{G}_1$, each process p_i wants to prove that there exists a witness α such that $x = g_0^{\alpha}$ and $y = g_1^{\alpha}$, and that p_i knows α.

	Network	fault tolerance	Adaptive adversary	Liveness	Unpredictability	Bias-resistance	Bit complexity	Computational complexity	Public-verifiability complexity	Cryptographic primitives	Setup
Cachin et al. [17]	A.	1/3	✗	✓	✓	✓	$O(\lambda n^2)$	$O(n)$	$O(1)$	Uniq. th-sig.	DKG
RandHerd [61]	P.	1/3[†]	✗	✓	✓	✓	$O(\lambda c^2 \log n)$[†]	$O(\lambda c^2 \log n)$	$O(1)$	PVSS+CoSi	DKG
Dfinity [18]	S.	1/3	✗	✓	✓	✓	$O(\lambda n^2)$	$O(n)$	$O(1)$	Uniq. th-sig.	DKG
Drand [3]	S.	1/2	✗	✓	✓	✓	$O(\lambda n^2)$	$O(n)$	$O(1)$	Uniq. th-sig.	DKG
HERB [23]	P.	1/3	✗	✓	✓	✓	$O(\lambda n^4)$[‡]	$O(n)$	$O(n)$	Partial HE	DKG
Algorand [36]	P.	1/3[†]	✗	✓	≈[*]	✗[**]	$O(\lambda cn)$[†]	$O(c)$	$O(1)$	VRF	CRS
Bitcoin [49]	S.	1/2	✗	✓	≈[*]	✗[**]	$O(\lambda n)$	very high	$O(1)$	Hash func.	CRS
Ouroboros [42]	S.	1/2	✗	✓	✓	✓	$O(\lambda n^4)$[‡]	$O(n^3)$	$O(n^3)$	PVSS	CRS
Scrape [20]	S.	1/2	✗	✓	✓	✓	$O(\lambda n^4)$[‡]	$O(n^2)$	$O(n^2)$	PVSS+Broadcast	CRS
Hydrand [57]	S.	1/3	✗	✓	≈[*]	✓	$O(\lambda n^2 \log n)$	$O(n)$	$O(n)$	PVSS	CRS
RandRunner [56]	S.	1/2	✓	✓	≈[*]	✓	$O(\lambda n^2)$	VDF	$O(1)$	VDF	CRS
GRandPiper [11]	S.	1/2	✗	✓	≈[*]	✓	$O(\lambda n^2)$	$O(n^2)$	$O(n^2)$	PVSS	q-SDH
BRandPiper [11]	S.	1/2	✓	✓	✓	✓	$O(\lambda n^3)$	$O(n^2)$	$O(n^2)$	VSS	q-SDH
Nguyen et al.[50]	S.	1	✗	✗	✓	✓	$O(n)$	$O(1)$	$O(1)$	FHE+VRF	–
ProofOfDelay[15]	S.	1/2	✗	✓	✓	✓	$O(n)$[§]	high	$O(1)$	VDF	–
No-dealer[44]	S.	1/2	✗	✓	✓	✓	$O(n^2)$	$O(n^2)$	$O(1)$	Shamir+HE	–
SPURT[27]	P.	1/3	✗	✓	✓	✓	$O(\lambda n^2)$	$O(n^2)$	$O(n)$	PVSS+Pairing	CRS
Kleroterion	P.	1/3	✗	✓	✓	✓	$O(\lambda n^2)$	$O(n)$	$O(n)$	PVSS+Pairing	CRS

† Algorand and Randherd use a randomly sampled committee of size c. This improves scalability at the cost of slightly reducing fault tolerance [27].
‡ The complexity counting that of their broadcast channel (blockchain) [27].
§ $O(n)$+ Ethereum [31].
∗ The adversary can withhold inputs and try to compute output which can break unpredictability [27, 52].
∗∗ The adversary can discard undesirable beacon outputs [27].

Table 2: Comparison of distributed random beacons [27, 31].

Thus, in addition to bilinear pairings, we use the non-interactive version of the Chaum-Pedersen Σ-protocol [22, 34, 51] in the random oracle model. The knowledge soundness of this protocol implies that if p_i convinces an honest process p_j with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from p_i with non-negligible probability. Let us denote by dleq the call to a non-interactive version of the Chaum-Pedersen protocol, such that dleq.Prove(α, g_0, x, g_1, y) generates the proof π and dleq.Verify(π, g_0, x, g_1, y) verifies the proof [27].

Threshold secret sharing. A (t, n)-threshold secret sharing scheme allows a process, known as the dealer, to share a secret $s \in \mathbb{Z}_q$ with n other processes, such that any $t + 1$ of them can reconstruct the message, but no t of them can. Analogously to SPURT [27], we also base off Shamir's secret sharing [58] scheme, in which a secret $s \in \mathbb{Z}_q$ is embedded in a polynomial $p(\cdot)$ of degree t such that $p(0) = a_0 = s$. The remaining t coefficients $\{a_i\}_{i=1}^{t}$ are chosen uniformly at random being thus $p(x) = \sum_{i=0}^{t} a_i x^i$.

The dealer then shares with process p_i the evaluation of $p(i)$. One can efficiently reconstruct the polynomial using Lagrange interpolation upon obtaining $t + 1$ evaluations of $p(x)$. Moreover, an adversary cannot learn the secret with any t or less evaluations of $p(x)$, except with the same probability of randomly guessing the secret.

2.1 Random Beacon Problem

We restate the random beacon (RB) problem [11, 27, 31]:

Definition 2.3 (Random beacon problem)**.** Let λ be a security parameter. Let a committee N of $|N| = n$ processes execute an epoch based protocol σ which outputs an output $Z \in Z_q$ per iteration. Then σ is an RB protocol if it satisfies all of the following properties with probability at least $1 - \epsilon(\lambda)$:

- **Agreement:** All honest processes agree on the same random output Z.
- **Availability:** Every honest process eventually outputs one value Z.
- **Verifiability:** If an honest process decides Z, then every honest process can verify it.
- **Unpredictability:** Before at least $t + 1$ processes output Z, no process can predict the value of Z with probability greater than $1/q + \epsilon(\lambda)$ (i.e. randomly guessing the secret).
- **Bias-resistance:** No process can fix some c bits of Z for any epoch with probability better than $\epsilon(c) + \epsilon(\lambda)$.

The first two properties, namely agreement and availability, are common to the consensus problem, although typically referred to as agreement and termination. Verifiability states that all honest processes can verify the validity of the output, which extends the simpler property of validity in the consensus problem. In fact, the fault-tolerance bounds known for consensus [32] are proven to also apply to the random beacon problem [44]. The properties of unpredictability and bias-resistance guarantee the randomness of the output with respect to all processes.

2.2 Set Byzantine Consensus

Since solving the RB problem is at least as hard as solving the consensus problem [44], and given that we are interested in scattering the distribution of inputs without having a leader broadcast them, we will solve a variant of consensus known as *Set Byzantine Consensus* [25, 26] (SBC).

Definition 2.4 (Set Byzantine Consensus). A protocol solves the Set Byzantine Consensus problem if it satisfies the following properties:

- **SBC-Termination.** every honest process eventually decides a set of values;
- **SBC-Agreement:** no two honest processes decide on different sets of values;
- **SBC-Validity:** the decided set of values is a subset of the union of the proposed values;
- **SBC-Nontriviality**: if all processes are honest and propose the same value v, then the decided set is $\{v\}$.

SBC-Termination and SBC-Agreement are analogous to the properties of the same name of the classical consensus problem, while SBC-Validity states that the decided set must contain proposed values, and SBC-Nontriviality is necessary to prevent trivial algorithms that decide a pre-determined value from solving the problem.

2.3 PVSS problem

Many RB implementations, like the one we present in this work, execute n instances of a publicly-verifiable secret-sharing (PVSS) protocol, one per process, after which some of the secrets shared by processes are selected to be aggregated into one final random output. A PVSS protocol has four phases [27]:

1. Setup: The dealer p_d generates and publishes the parameters of the scheme. Every process p_i publishes a public key pk_i and withholds the corresponding secret key sk_i.
2. Distribution: The dealer creates the secret shares $\mathbf{c}_d = \{c_{i,d}\}$ for each process p_i, along with a proof $\mathbf{v}_d$ that these are indeed valid encrypted shares of some secret.
3. Verification: Each process (or an external verifier) verify that the secret shares $\mathbf{c}_d$ are indeed valid shares of some secret.
4. Reconstruction: In this phase, each process p_i decrypts their respective share $c_{i,d}$ with their secret key sk_i, obtaining their decrypted share $s_{i,d} = pk_i^{c_{i,d}}$, and shares $s_{i,d}$ along with a (non-interactive) zero-knowledge proof that $s_{i,d}$ is a correct decryption of $c_{i,d}$. Each process (or an external verifier) verifies the decrypted shares, and applies a reconstruction procedure to recover the original secret s_d shared by the dealer p_d.

Let n be the number of processes. We define the APVSS problem here below.

Definition 2.5 (Publicly-verifiable secret-sharing). Let λ be a security parameter, and let a dealer p_d share a secret s with $n-1$ additional processes following a protocol σ. Then, σ is a PVSS protocol if it satisfies the following properties:

- *Verifiability*: If the check in the verification step returns 1, i.e. succeeds, then with probability at least $1-\epsilon(\lambda)$ the encryptions $\mathbf{c}$ are valid shares of some secret. Furthermore, if the check in the Reconstruction phase passes then the communicated values $\mathbf{c}$ are indeed the shares of a secret distributed by the dealer.
- *Correctness*: if p_d is honest, then with probability at least $1-\epsilon(\lambda)$ the checks in the verification and reconstruction steps succeed, and honest processes can reconstruct s.
- *Secrecy*: If p_d is honest, then the probability of $\mathcal{A}$ learning any information about p_d's secret s prior to the reconstruction phase is at most $\epsilon(\lambda)$.

The property of secrecy [11] has been formally described as indistinguishability of secrets (IND1-Secrecy) [20, 40, 55].

3 THE PINAKION PROTOCOL

In this section, we illustrate our Pinakion protocol that solves PVSS. In order to solve PVSS, we modify the SPURT's PVSS protocol, Π_{DBDH} [27], adding one major difference, in that instead of relying on the dealer to share the same value to all processes, we require all processes to reliably broadcast their inputs. This modification allows the random beacon that we propose in Section 4 to use one bit to reference this secret, while ensuring all honest processes store locally the same value without the need for a leader broadcasting a digest of the secret.

The setup phase is the same to that of Π_{DBDH}: $\mathsf{PVSS.Setup}(1^\lambda) \rightarrow (g_0, h_0, g_1, h_1, (sk_i, pk_i))$: The setup algorithm chooses uniformly random and independent generators g_0, $h_0 \in \mathbb{G}_0$ and g_1, $h_1 \in \mathbb{G}_1$ and publishes them in a trusted PKI (which is only used in this step). Each process p_i also generates a secret key $sk_i \in \mathbb{Z}_q$ and public key $pk_i = h_0^{sk_i}$, and publishes pk_i in the public ledger.

Algorithm 1 illustrates the rest of the Pinakion protocol. After the setup phase, the dealer p_d selects a secret s to share. For this purpose, processes select a polynomial $p(x)$ of degree t whose coefficients have been chosen uniformly at random from $\mathbb{Z}_q$, such that $p(0) = s$ (line 9). Then, p_d computes the secret shares $p(j)\,\forall j \in [n]\backslash\{i\}$ which it encrypts with the public key of the recipient $c_{j,d} = pk_j^{p(j)}$, obtaining the vector $\mathbf{c}_d$ (line 11). Additionally, p_d also computes a non-interactive zero-knowledge proof vector $\mathbf{v_d}$ such that $v_{j,d} = g_1^{p(j)}$ that serves as a commitment to the secret shares and to verify the validity of the encrypted shares $\mathbf{c_d}$ (lines 12-14).

Following, in line 15 process p_d calls RBV-broadcast with the commitments and encrypted shares $(\mathbf{v_d}, \mathbf{c_d})$. The RBV-broadcast protocol is almost identical to the reliable broadcast protocol outlined by recent works [14]. The only modification we add is for honest processes to only deliver a message containing $\mathbf{c}_d$ and $\mathbf{v}_d$ from p_i if the verification of Pinakion checks in the calls to $\mathsf{Pinakion.verify}$. We call this variant *reliable broadcast with verification* (RBV). We show in Algorithm 2 the RBV-broadcast protocol, which consists of an accountable reliable broadcast that covers the distribution and verification steps of the Pinakion protocol. If process p_i terminates an execution of the RBV-broadcast protocol returning a value v, then we say the p_i RBV-delivers v. In this case, p_d RBV-broadcasts the list of shares $\mathbf{c}_d$ and zero-knowledge proofs $\mathbf{v}_d$. Our

Algorithm 1 Pinakion protocol with dealer p_d

```
1:  State:
2:      $g_0, h_0 \in \mathbb{G}_0$; $g_1, h_1 \in \mathbb{G}_1$, uniformly random and independent generators.
3:      $sk_i \in \mathbb{Z}_q$ secret key of $p_i$
4:      $pk_j = h_0^{sk_j}$ public key of $p_j$, for $j \in [n]$
5:      $S = e(h_0^s, h_1)$, secret that process $p_d$ shares, with $s \overset{\$}{\leftarrow} \mathbb{Z}_q$

6:  Pinakion.Share:                                         ▷ executed by $p_d$
7:      for $j \in [1, t]$ do
8:          $a_k \overset{\$}{\leftarrow} \mathbb{Z}_q$
9:      $p(x) \leftarrow s + a_1 x + ... + a_t x^t$
10:     for $j \in [0, n-1]$ do
11:         $c_{j,d} \leftarrow pk_j^{p(j)}$                   ▷ encrypt with $p_j$'s public key
12:         $v_{j,d} \leftarrow g_1^{p(j)}$
13:     $\mathbf{v_d} \leftarrow \{v_{0,d}, v_{1,d}, ..., v_{n-1,d}\}$
14:     $\mathbf{c_d} \leftarrow \{c_{0,d}, c_{1,d}, ..., c_{n-1,d}\}$
15:     RBV-broadcast($\{\mathbf{v_d}, \mathbf{c_d}\}$)                 ▷ Distribution & Verification

16: Pinakion.Reconstruction:                                ▷ executed by each $p_i$
17:     $shares \leftarrow \{\}$
18:     when ($p_i$ RBV-delivers $\{\mathbf{v_d}, \mathbf{c_d}\}$) do
19:         $s_{i,d} \leftarrow c_{i,d}^{1/sk_i}$                    ▷ decrypt secret share
20:         broadcast($s_{i,d}$)                             ▷ broadcast secret share
21:     when ($s_{j,d}$ is delivered) do
22:         if (Pinakion.check($h_0, v_{j,d}, s_{j,d}, g_1$)) then
23:             $shares[j] \leftarrow s_{j,d}$
24:             if (size($shares$) $> t$) then
25:                 $h_0^s \leftarrow$ Pinakion.interpolate($h_0, h_1, shares$)        ▷ reconstruct
26:                 return($e(h_0^s, h_1)$)

27: Pinakion.check($a, b, c, d$):
28:     return $e(a, b) = e(c, d)$
```

RBV-broadcast satisfies that honest processes only RBV-deliver values that pass the verification, along with the already properties of reliable broadcast [24, 25]:

- RBV-Validity: If an honest process RBV-delivers a message m from an honest process p_i, then p_i RBV-broadcast m.
- RBV-Unicity: an honest process RBV-delivers at most one message from p_i.
- RBV-Termination-1: If p_i is honest and RBV-broadcasts a message m, then honest processes eventually RBV-deliver m from p_i.
- RBV-Termination-2: If an honest process RBV-delivers a message m from p_i (possibly faulty) then honest process eventually RBV-deliver the same message m from p_i.

Notice that processes verify all messages as soon as they are received and discard all messages that do not pass the verification. This is because we outline in Algorithm 2 the main idea of the algorithm, but there are simple optimizations, such as not verifying messages that were already verified, or only sharing the hash of the message in all broadcast except for the INITIAL message. We refer to previous work for more details on these optimizations [25].

The verification shown in Pinakion.verify reuses properties of error-correcting code already used in SPURT [27]. We however restate the properties of the verification step, particularly that sharing a secret s using a degree t polynomial among n processes is equivalent to encoding the message $(x, a_1, a_2, ..., a_t)$ using a $[n, t+1, n-t]$ Reed-Solomon code C [47, 54], where a $[n, k, d]$ linear error correcting code over Z_q of length n, minimum distance d and dimension k. Also, we define $C^\perp$ as the dual code of C i.e., $C^\perp$ consists of vectors $x^\perp \in \mathbb{Z}_q^n$ such that for all $x \in C$, $x \cdot x^\perp = 0$ where $\cdot$ is the inner product operation. The call to Pinakion.verify uses the result of Lemma 3.1 on linear error correcting code, proved by Cascudo et al. [20].

Lemma 3.1. *If $x \in Z_q^n \backslash C$, and $y^\perp$ is chosen uniformly at random from $C^\perp$, then the probability that $x \cdot y^\perp = 1$ is exactly $1/q$.*

Finally, once process p_j RBV-delivers the values $(\mathbf{v_d}, \mathbf{c_d})$, it decrypts its share of the secret and broadcasts it in line 20. Then, it waits until it delivers at least another t valid decrypted secret shares to reconstruct the secret using Lagrange interpolation in line 25, finalizing the reconstruction of the secret. All received decrypted shares have to first pass a simpler check than the verification at the RBV-broadcast, represented in the call to Pinakion.check($h_0, v_{j,d}, s_{j,d}, g_1$) in line 22, which consists of checking whether $e(a, b) = e(c, d)$. honest processes construct h_0^s in the call to Pinakion.interpolate using Lagrange interpolation:

$$\prod_{k \in T} (s_{k,d})^{\mu_k} = \prod_{k \in T} h_0^{\mu_k \cdot p(k)} = h^{p(0)}, \tag{3}$$

where T is the set of processes from which p_i received valid decrypted shares, $|T| > t$, and $\mu_k = \prod_{j \neq k} \frac{j}{j-k}$ are the Lagrange coefficients [27].

4 THE (NAIVE) KLEROTERION PROTOCOL

Algorithm 2 RBV-broadcast

```
1:  State:
2:      $g_0, h_0 \in \mathbb{G}_0$; $g_1, h_1 \in \mathbb{G}_1$, uniformly random and independent generators.
3:      $sk_i \in \mathbb{Z}_q$ secret key of $p_i$
4:      $pk_j = h_0^{sk_j}$ public key of $p_j$, for $j \in [n]$

5:  RBV-broadcast($\{\mathbf{v_d}, \mathbf{c_d}\}$):                        ▷ executed by $p_d$
6:      if (Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v_d}, \mathbf{c_d}\}$)) then          ▷ Verification
7:          broadcast(INITIAL), $\{\mathbf{v_d}, \mathbf{c_d}\}$)                ▷ Broadcast to all
8:      upon receiving a message (INITIAL, $\{\mathbf{v_d}, \mathbf{c_d}\}$) from $p_j$ do
9:          if (Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v_d}, \mathbf{c_d}\}$)) then      ▷ Verification
10:             broadcast(ECHO, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j)                   ▷ Echo to all
11:     upon receiving $n-t$ distinct (ECHO, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j) and not having sent any READY do
12:         if (Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v_d}, \mathbf{c_d}\}$)) then      ▷ Verification
13:             broadcast(READY, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j)             ▷ Send READY to all
14:     upon receiving $t+1$ distinct (READY, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j) and not having sent any READY do
15:         if (Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v_d}, \mathbf{c_d}\}$)) then      ▷ Verification
16:             broadcast(READY, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j)             ▷ Send READY to all
17:     upon receiving $n-t$ distinct (READY, $\{\mathbf{v_d}, \mathbf{c_d}\}$, j) and not having RBV-delivered any message do
18:         if (Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v_d}, \mathbf{c_d}\}$)) then      ▷ Verification
19:             return($\{\mathbf{v_d}, \mathbf{c_d}\}$)             ▷ Deliver and send READY to all

20: Pinakion.verify($g_1, \{pk\}_{j=1}^n, \{\mathbf{v}, \mathbf{c}\}$):
21:     $\mathbf{x}^\perp \overset{\$}{\leftarrow} \mathbf{C}^\perp$
22:     if ($\Pi^{j \in [n]} v_j^{x_j^\perp} \neq 1_{\mathbb{G}_1}$) then return False
23:     for $j \in [n]$ do
24:         if (not Pinakion.check($pk_j, v_j, c_j, g_1$)) then return False
25:     return True
```

In this section, we detail in depth the Kleroterion protocol shown in Algorithm 4. We first illustrate a naive implementation, that does not look at possible optimizations, to then explain the optimizations in Section 5.

The Kleroterion protocol generates a random output by running n executions of the Pinakion protocol, one per process. However, instead of having each Pinakion execution terminate independently, we execute an instance of consensus in order to have processes decide on $t+1$ inputs before reconstructing them, that is, before processes know the exact value associated with that execution of Pinakion. After deciding on exactly $t+1$ secrets, honest processes reconstruct and then aggregate these $t+1$ secrets to generate the random output.

For the consensus protocol, we use the variant of HotStuff proposed by SPURT [5, 27], for nearly-simultaneous decision, i.e. all processes learn the decision within two message delays. This protocol proceeds in epochs, with a rotating leader per epoch that proposes a value to decide, as we show in Algorithm 3. Thus, this variant differs from HotStuff only in that processes broadcast their signed PREPARE, PRECOMMIT and COMMIT messages, instead of sending them to the leader. As a result, this variant preserves the safety and liveness properties of HotStuff, as well as its responsiveness property [5] (i.e. outputs are generated at the real network latency and not at Δ at best).

However, contrary to SPURT, we do not require the leader of an epoch to propose to the rest a digest of his proposed secrets. Instead, leaders propose to decide on a bitmask of n bits, in which the i-th bit is associated with the secret shared by process p_i in a call of Pinakion.share. The bits that are set to 1 are secrets that will be used for the aggregation, while those set to 0 are not to be used for the aggregation. Honest processes only contribute to consensus in epochs whose proposed bitmask contains exactly $t+1$ bits set to 1, so that exactly $t+1$ shared secrets are aggregated into the final output. We refer to the number of bits set to 1 of a bitmask as its *Hamming weight.* We illustrate an example execution with $n = 4$ processes in Figure 1.

As such, the leader for this round starts participating in an epoch of the consensus protocol only if it first RBV-delivers $t+1$ values from the n previous RBV-broadcast executed. Then, it proposes a bitmask in which all associated bits to these $t+1$ values are set to 1, and the rest to 0. The rest of the processes execute then an exact copy of the SPURT variant of HotStuff, in which they only contribute if the Hamming weight of the proposed bitmask is $t+1$ and they RBV-delivered the $t+1$ associated secrets.

Once an honest process decides on a bitmask, it decrypts its share of each of the $t+1$ decided secrets (line 20) and invokes Kleroterion.Reconstruct(*decided_secrets*, *decrypted_shares*) with its decrypted shares and the decided vectors in order to reconstruct each of the secrets and aggregate them into one final random output. We illustrate the decision of the secrets with which to compute the random output in Algorithm 4, which integrates thus the share and verification steps of the Pinakion protocol into Kleroterion.

Setup. The Kleroterion protocol's setup phase thus consists of both consensus' and Pinakion's setup phases. That is, the setup phase of Kleroterion takes part with the creation of the keys of each process at the beginning of the Kleroterion protocol, stored in a PKI (which is not used later), along with the aforementioned CRS setup. These keys and the rest of values can be reused in all iterations of the random beacon, with the exception of the randomly chosen polynomial coefficients, which must vary in each execution [27].

Algorithm 3 SPURT's [27] Steady state of a modified HotStuff [5] protocol that does not use threshold signatures and has a bit complexity $O(|M|n^2)$ per decision.

Let r be the current epoch and L be its leader. Also, let $ht-1$ be the latest finalized iteration of the protocol.

- **Propose.** L proposes a value M to be finalized at height ht by sending $\langle PROPOSE, M, r, ht, X\rangle$ message to all the processes. X is the view change certificate (if any) that validates that the proposal is safe.
- **Prepare.** Each process p_j, upon receiving the proposal, checks whether the proposal is consistent with HotStuff specifications using X, and $P(M)$ is true for an external predicate $P()$. If both checks pass, process p_j sends $\langle \mathsf{PREPARE}, M, r, ht\rangle$ to all processes.
- **Pre-Commit.** Upon receiving $2t+1$ PREPARE messages for the proposal M at height ht and epoch r, process p_j sends $\langle \mathsf{PRECOMMIT}, M, r, ht\rangle$ message to every process.
- **Commit.** Upon receiving $2t+1$ PRECOMMIT messages for the proposal M at height ht and epoch r, process p_j sends $\langle \mathsf{COMMIT}, M, r, ht\rangle$ message to every process.
- Each process outputs M upon receiving $2t+1$ COMMIT messages corresponding to M.

Share and decide. The call to Pinakion.Share in line 6 creates a random input per instance of Pinakion, and RBV-broadcasts the secret shares and commitments of the random input. However, in this case each process also computes and shares their dleq proofs $\pi_{i,j} = \mathsf{dleq.Prove}(g_1, v_{i,j}, pk_j, c_{i,j}, p_i(j))$ in the call to Pinakion.Share, where $p_i(j)$ is the share of p_i's secret for process p_j. Process p_i then RBV-broadcasts $\{\mathbf{v_i}, \mathbf{c_i}, \boldsymbol{\pi}_\mathbf{i}\}$, and each process p_j verifies the dleq proof by calling $\mathsf{dleq.Verify}(\pi_{i,j}, pk_j, c_{i,j}, g_1, v_{i,j})$ when delivering messages from p_i, instead of calling Pinakion.check, as this call becomes redundant with dleq.Verify [27]. This guarantees that the secrets of honest processes are independent of all other secrets by the knowledge soundness property.

Once process p_i RBV-delivers $t+1$ proposals (line 30), and if p_i is the leader of this epoch, then p_i starts the respective consensus with the intention to decide 1 on such proposals (line 16), selecting the secrets with which to compute the final random output. If a process p_i is not the leader of this epoch but p_i receives a bitmask proposed from the leader in this epoch, then p_i checks whether p_i RBV-delivered all the proposed secrets associated to the bitmask (line 17). If it does, then it contributes to consensus in this epoch. Otherwise, it does not respond in this epoch.

Once consensus terminates deciding a bitmask, process p_i waits until it RBV-delivers the proposals associated with each bit set to 1 in line 24. This can happen if consensus terminated without the participation of p_i. Following, p_i decrypts its corresponding secret share of each decided secret in line 27. Finally, p_i calls Kleroterion.Reconstruct (line 28) with its decrypted shares and list

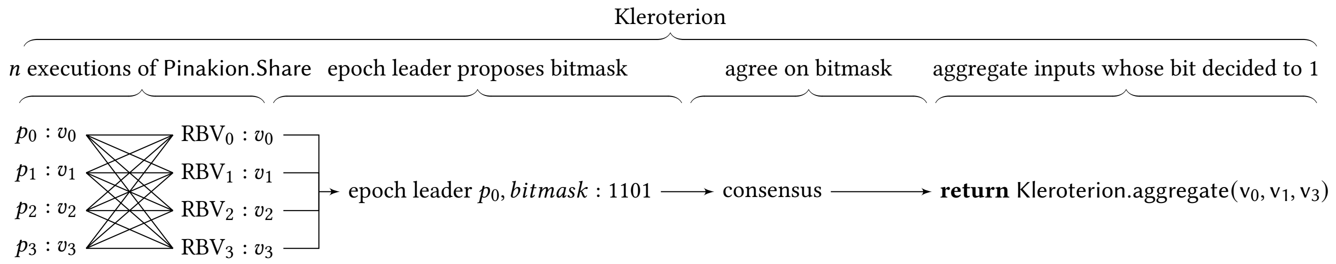
Kleroterion
n executions of Pinakion.Share
epoch leader proposes bitmask
agree on bitmask
aggregate inputs whose bit decided to 1
$p_0 : v_0$
$p_1 : v_1$
$p_2 : v_2$
$p_3 : v_3$
$\mathrm{RBV}_0 : v_0$
$\mathrm{RBV}_1 : v_1$
$\mathrm{RBV}_2 : v_2$
$\mathrm{RBV}_3 : v_3$
epoch leader p_0, $bitmask : 1101$
consensus
return Kleroterion.aggregate($\mathrm{v}_0, \mathrm{v}_1, \mathrm{v}_3$)

Figure 1: Kleroterion execution example with $n = 4$ processes. First, each process p_i selects their input value v_i (we omit secret shares and reconstruction for simplicity), which they share with everyone executing their respective instance of RBV-broadcast as part of their respective call to Pinakion.Share. Then, processes execute one leader-based consensus protocol that proceeds in epoch (HotStuff), in which the leader proposes a bitmask of n bits, with Hamming weight $t + 1$. Upon deciding on a bitmask, processes reconstruct and aggregate the $t + 1$ input secrets whose associated bit of the bitmask decided to 1.

of decided proposals, and returns the random output resulted from the call to Kleroterion.Aggregate (line 29) which aggregates the decided reconstructed secrets. honest processes can then return the computed final random output of this iteration of the random beacon, and move on to the next iteration of the random output.

Algorithm 4 Kleroterion.Decide for process p_i

```
 1: State:
 2:   $g_0, h_0 \in \mathbb{G}_0$; $g_1, h_1 \in \mathbb{G}_1$, uniformly random and independent generators.
 3:   $sk_i \in \mathbb{Z}_q$ secret key of $p_i$
 4:   $pk_j = h_0^{sk_j}$ public key of $p_j$, for $j \in [n]$
 ------------------------------------------------------------
 5: $s \xleftarrow{\$} \mathbb{Z}_q$
 6: Pinakion.Share(s)                         ▷ create and RBV-broadcast secret
 7: repeat:
 8:   if ($consensus$.epoch_leader() $= p_i$) then
 9:     if ($|proposals_i| \geq t + 1$) then
10:       $bitmask \leftarrow \{\}$
11:       $k = 0$
12:       while $|bitmask| < n$ do
13:         if ($k \in proposals_i$.keys() and hamming_weight($bitmask$) $< t + 1$) then
14:           $bitmask[k] \leftarrow 1$
15:           $k \leftarrow k + 1$
16:       ⟨PROPOSE, $bitmask$, $consensus.epoch$, $ht$, $X$⟩
17:   else if (received ⟨PROPOSE, $bitmask$, $r$, $ht$, $X$⟩ and $consensus.epoch = r$) then
18:     if ($j \in proposals_i$.keys() $\forall j$ s.t. $bitmask[j] = 1$ and $\sum_j bitmask[j] = t + 1$) then
19:       contribute to consensus in epoch $r$
20: until $consensus$.finished()              ▷ $t + 1$ secrets chosen
21: $decrypted\_shares \leftarrow \{\}$
22: $decided\_secrets \leftarrow \{\}$
23: for each $j$ s.t. $consensus.decision[j] = 1$ do    ▷ for each decided secret
24:   wait_until($proposals_i[j] \neq \perp$)
25:   $\mathrm{c_j} \leftarrow proposals_i[j]$
26:   $decided\_secrets[j] \leftarrow proposals_i[j]$
27:   $decrypted\_shares[j] \leftarrow c_{i,j}^{1/sk_i}$          ▷ decrypt $p_i$'s share
28: $decisions \leftarrow$ Kleroterion.Reconstruct($decided\_secrets$, $decrypted\_shares$)
29: return Kleroterion.Aggregate($decisions$)     ▷ aggregate into random output
 ------------------------------------------------------------
30: when $p_i$ RBV-delivers $\{\mathrm{v_j}, \mathrm{c_j}, \pi_\mathrm{j}\}$ do:
31:   $proposals_i[j] \leftarrow \mathrm{c_j}$
```

Reconstruct and aggregate. After terminating consensus, the reconstruction phase starts to reconstruct the corresponding $t + 1$ decided secrets. We show in algorithm 5 the call to Kleroterion.Reconstruct. First, p_i broadcasts its decrypted shares of the decided secrets in line 7. Upon receiving a list of decided secrets and decrypted shares from p_j, p_i verifies them in the call to Kleroterion.Verify (line 11). Kleroterion.Verify checks first that the local and delivered list of decided secrets is the same (line 17) and that each decrypted shares passes the Pinakion.check (line 20) previously described. If the received message verifies, then its decrypted shares are added to the list of decrypted shares (line 12), which is used to reconstruct all the decided secrets with the call to Kleroterion.MultipleRecon (line 14) once the list contains at least $t + 1$ decrypted shares for each secret. The call to Kleroterion.MultipleRecon gathers all the decrypted shares (line 27) for each secret and reconstructs them calling Pinakion.interpolate in line 28.

4.1 Proofs of correctness

The Pinakion protocol shown in Section 3 is an instantiation of SPURT's Π_{DBDH} in which the broadcast primitive is replaced by our RBV-broadcast. Similarly, our RBV-Broadcast is almost identical to the implementation by Bracha et al. [14], with the only modification that processes only deliver a value if it passes the verification step from Π_{DBDH}. The proofs of Pinakion solving PVSS and of RBV-broadcast solving reliable broadcast are thus analogous to those of these two previous works. Liveness of both reliable broadcast and PVSS are subject to the dealer (or source) being honest, meaning that replacing the broadcast primitive by our RBV-broadcast preserves liveness. On the safety side, RBV-broadcast further enhances safety thanks to the rest of the properties of reliable broadcast compared with a general broadcast primitive.

The same occurs with Kleroterion. The only additional difference between the above-shown unoptimized Kleroterion and SPURT is that while in SPURT the leader of the epoch shares a digest of the $t+1$ selected shares, in Kleroterion the leader shares a bitmask with Hamming weight $t+1$, that associates each bit to a particular secret RBV-broadcast by each process. We thus need to prove only that if an honest process decides a bitmask in an epoch r, then all honest processes eventually reconstruct and output the same final random output from that epoch. We show this in Lemma 4.2. We show first however that Kleroterion solves SBC.

Algorithm 5 Kleroterion.Reconstruct for process p_i

```
1: State:
2:   $g_0, h_0 \in \mathbb{G}_0$; $g_1, h_1 \in \mathbb{G}_1$, uniformly random and independent generators.
3:   $sk_i \in \mathbb{Z}_q$ secret key of $p_i$
4:   $pk_j = h_0^{sk_j}$ public key of $p_j$, for $j \in [n]$
5:   $decided\_secrets_i$, list of decided encrypted secret shares of $p_i$
6:   $decided\_shares_i$, list of decided decrypted shares of $p_i$

7: broadcast($decrypted\_shares$)
8: $list\_decrypted\_shares \leftarrow \{\}$
9: $random\_outputs \leftarrow \bot$
10: Upon receiving $\{decided\_secrets_j, decrypted\_shares_j\}$ from $p_j$:
11:   if (Kleroterion.verify($decided\_secrets_j, decrypted\_shares_j, h_0, g_1$) and $random\_outputs = \bot$) then   ▷ check decryption
12:     $list\_decrypted\_shares[j] \leftarrow decrypted\_shares_j$
13:     if (size($list\_decrypted\_shares$) $> t$) then   ▷ enough to reconstruct
14:       $random\_outputs \leftarrow$ Kleroterion.MultipleRecon($h_1, h_0, list\_decrypted\_shares$)
15:       if ($random\_outputs \neq \bot$) then return $random\_outputs$

16: Kleroterion.verify($decided\_secrets_j, decrypted\_shares_j, h_0, g_1$):
17:   if ($decided\_secrets_j \neq decided\_secrets_i$) then return False   ▷ different secrets
18:   for each $s_{j,k}$ in $decrypted\_shares_j$ do
19:     $v_{j,k} \leftarrow decided\_secrets[j].\mathbf{v}_j[k]$
20:     if (not Pinakion.check($h_0, v_{j,k}, s_{j,k}, g_1$)) then return False
21:   return True

22: Kleroterion.MultipleRecon($h_1, h_0, list\_decrypted\_shares, decided\_secrets$):
23:   $random\_outputs \leftarrow \{\}$
24:   for $k$ in $decided\_secrets$.keys() do
25:     $aux \leftarrow \{\}$
26:     for j in $list\_decrypted\_shares$.keys() do   ▷ for each secret
27:       $aux[j] \leftarrow list\_decrypted\_shares[j][k]$   ▷ gather all decrypted shares
28:     $random\_outputs[k] \leftarrow$ Pinakion.interpolate($h_0, h_1, aux$)   ▷ and reconstruct
29:   return $random\_outputs$
```

Theorem 4.1. Kleroterion solves SBC.

Proof. SBC-Agreement derives from RBV-Unicity, RBV-Termination-2 and the agreement property of the HotStuff protocol. That is, by the agreement property of HotStuff all processes agree on the bitmask. By RBV-Unicity and RBV-Termination-2 all processes agree on the values that the bits set to 1 of the bitmask refer to.

SBC-Termination derives from RBV-Termination-1 and the termination property of HotStuff. By RBV-Termination-1 all processes eventually deliver at least the $2t + 1$ values shared by honest processes. If honest processes keep trying different bitmasks when they are the leader of an epoch, eventually there is an epoch after GST whose leader is honest, and proposes RBV-delivered values that have been RBV-delivered by all other honest processes. Processes can terminate in that epoch.

SBC-Validity and SBC-Nontriviality are trivial as the decision is a bitmask of proposals. □

Lemma 4.2. Suppose an honest process decides a bitmask bitmask in epoch r, let I be the set of decrypted polynomials references by the bits of bitmask set to 1, and let $\hat{p}$ be the aggregated polynomial $\hat{p}() = \sum_{i \in I} p_i(\cdot)$. Then, every honest process outputs $e(h_0^a, h^1)$ where $a \in \mathbb{Z}_q$, for $a = \hat{p}(0)$.

Proof. For an honest process to terminate in epoch r, at least $t + 1$ honest processes participated in the consensus protocol in r. By construction, honest processes only participate in an epoch of the consensus protocol if they have first RBV-delivered all the values referenced by the bitmask, and if the Hamming weight of the bitmask is exactly $t + 1$. Also by construction, an honest process only RBV-delivers a share of secrets if it passes the verification step.

By Theorem 4.1, all honest processes will eventually decide on the same bitmask and all honest processes will eventually RBV-deliver all the values associated with each bit set to 1 of the bitmask.

As a result, except with negligible probability, the degree of $\hat{p}(\cdot)$ is at most t. This is because any polynomial of degree greater than t passes the verification step of RBV-broadcast with probability only $1/q$; hence, the probability that it passes the check at $t + 1$ honest nodes is $\binom{2t+1}{t+1}\frac{1}{q^{t+1}} \leq \frac{1}{q}$, which is negligible [27].

Every honest process j that participated in consensus holds the witness $h_0^{sk_j \cdot \hat{p}(j)}$ if the dleq.Verify check passed [27]. Thus, at least $t + 1$ honest processes will broadcast their decrypted shares during the reconstruction such that anyone can verify them.

Finally, after performing Lagrange interpolation over the exponent with these $t + 1$ decrypted shares (which pass the equality check [27]), all honest processes can recover the beacon output $e(h_0^{\hat{p}(0)}, h_1)$. □

Lemma 4.2 is our analogous proof of SPURT's Lemma 2 [27]. We refer to SPURT for the rest of the proofs of correctness, as they are identical. We analyze in Section 4.2 the complexities of an unoptimized Kleroterion, and provide optimizations that reduce the complexity of Kleroterion to $O(n^2)$ per decision, with the advantage of scattering the shared bits throughout all pairwise channels of the network (instead of channels to and from the leader, as is the case for leader-based protocols).

4.2 Complexities of naive Kleroterion

Table 3 shows the time, computational, message and bit complexities of Kleroterion, Pinakion, and RBV-broadcast. RBV-broadcast requires each n processes to broadcast to all n processes the n encrypted secret shares, meaning a message of size $O(\lambda \cdot n)$, which needs to be verified for each of its elements. Pinakion provides the same complexities, as the bottleneck of Pinakion is the RBV-broadcast. The Kleroterion protocol runs n concurrent executions of Pinakion, increasing message and bit complexities by a linear factor compared to Pinakion's complexities. The HotStuff protocol's message complexity is $O(n^2)$. However, SPURT's variant requires all processes to broadcast messages in order to satisfy nearly-simultaneous decision, instead of sending messages to the leader for aggregation. As a result, this variant has message complexity $O(n^3)$, and since the proposal is a bitmask of n bits, the resulting communication complexity of the naive approach is $O(n^4)$. The bottleneck of a naive Kleroterion implementation is thus the n instances of Pinakion, resulting in a message complexity of $O(n^3)$ and a bit complexity of $O(\lambda n^4)$.

	Complexities		
Protocol	Time	Message	Bit
Naive RBV-broadcast	$O(1)$	$O(n^2)$	$O(\lambda n^3)$
Naive Pinakion	$O(1)$	$O(n^2)$	$O(\lambda n^3)$
Naive Consensus	$O(t)$	$O(n^3)$	$O(n^4)$
Naive Kleroterion	$O(t)$	$O(n^3)$	$O(\lambda n^4)$

Table 3: time, computation, message and bit complexities of naive RBV-broadcast, Pinakion and Kleroterion.

5 OPTIMIZATIONS AND OBSERVATIONS

The straw man implementation we showed in Section 4, while correct, does not provide the bit complexity of $O(\lambda n^2)$ per decision that we claim in Table 2. We detail here the optimizations that decrease the bit complexity of Kleroterion from $O(\lambda n^4)$ to $O(\lambda n^2)$ per decision. Furthermore, we observe in this section that Kleroterion is in fact better-suited to be implement in Wide Area Networks (WANs), e.g. the Internet, than other PVSS-based random beacons with the same bit complexity, as the number of bits processes send per each channel per decision is independent of the number of participants. In contrast, SPURT saturates channels to the leader sending $O(\lambda n)$ bits through them. We also observe the advantages of decoupling the consensus proposal from the actual shared values by proposing a bitmask, in that the associated bit enables batching.

5.1 From quartic to quadratic bit complexity

In this section, we show how to reduce Kleroterion's bit complexity from quartic to quadratic. For this purpose, we present three optimizations.

Bitmask digest. The first optimization refers to the HotStuff consensus protocol. In order to reduce the bit complexity, instead of having processes decide on a bitmask of size $O(n)$ bits, the leader broadcasts the bitmask initially and then proposes a digest of the bitmask, of size $O(\lambda)$ bits. This results in a bit complexity of $O(\lambda n^3)$ for the consensus protocol, since processes do not need to broadcast a message of size $O(n)$ but instead only the digest of size $O(\lambda)$. Honest processes can still satisfy correctness of the protocol by contributing to consensus only if they received the bitmask that corresponds to the proposed digest. This is an advantage of having a leader propose the decision to decide during consensus.

Aggregated inputs. In Algorithm 1, each dealer p_d shares a list of secrets $\boldsymbol{c}_d$ and commitments $\boldsymbol{v}_d$, with the addition of the vector of dleq proofs $\boldsymbol{\pi}_d$ for Kleroterion. SPURT required all processes to send these vectors to a leader so that the leader aggregates them, but this saturates the number of bits sent through the channels to the leader. However, Kleroterion cannot aggregate commitments and secrets from different processes, since it does not have a leader that can receive all these vectors from all processes. As such, instead, Kleroterion requests processes to generate n secrets per process, and aggregate them locally before sharing them, such that process p_i generates n secrets $\{s_{i,k}\}_{k\in[n]}$ and then generates his shares $\{c_{i,j,k}\}_{k\in[n],j\in[n]}$, commitments $\{v_{i,j,k}\}_{k\in[n],j\in[n]}$, and dleq proofs $\{\pi_{i,j,k}\}_{k\in[n],j\in[n]}$. Then, p_i aggregates the shares

Algorithm 6 Optimized RBV-broadcast

```
 1: State:
 2:    $g_0, h_0 \in \mathbb{G}_0$; $g_1, h_1 \in \mathbb{G}_1$, uniformly random and independent generators
 3:    $sk_i \in \mathbb{Z}_q$ secret key of $p_i$
 4:    $pk_j = h_0^{sk_j}$ public key of $p_j$, for $j \in [n]$
 5: RBV-broadcast($\{\{c_{d,j,k}\}, \{v_{d,j,k}\}, \{\pi_{d,j,k}\}\}$):        ▷ executed by $p_d$
 6:    for $j \in [n]$ do
 7:       $\hat{c}_{d,j} \leftarrow \prod_{k\in[n]} c_{d,j,k}$
 8:       $\hat{v}_{d,j} \leftarrow \prod_{k\in[n]} v_{d,j,k}$
 9:    for $j \in [n]$ do
10:       if (Pinakion.verify($g_1$, $\{pk_k\}$, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d\}$)) then        ▷ Verification
11:          send(INITIAL, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \overline{\boldsymbol{c}}_{d,j}, \overline{\boldsymbol{v}}_{d,j}, \overline{\boldsymbol{\pi}}_{d,j}\}$)        ▷ Broadcast to all
12:    upon receiving a message (INITIAL, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \overline{\boldsymbol{c}}_{d,j}, \overline{\boldsymbol{v}}_{d,j}, \overline{\boldsymbol{\pi}}_{d,j}\}$) from $p_j$ do
13:       if (Pinakion.verify-opt($g_1$, $\{pk\}_{j=1}^{n}$, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \overline{\boldsymbol{c}}_{d,j}, \overline{\boldsymbol{v}}_{d,j}, \overline{\boldsymbol{\pi}}_{d,j}\}$)) then
14:          broadcast(ECHO, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d\}$, $j$)        ▷ Echo to all
15:    upon receiving $n - t_\ell$ distinct (ECHO, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d\}$, $j$) and not having sent any
   READY do
16:       broadcast(READY, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d\}$, $j$)        ▷ Send READY to all
17: Pinakion.verify-opt($g_1$, $\{pk\}_{j=1}^{n}$, $\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \overline{\boldsymbol{c}}_{d,i}, \overline{\boldsymbol{v}}_{d,i}, \overline{\boldsymbol{\pi}}_{d,i}\}$):
18:    $\mathbf{x}^\perp \xleftarrow{\$} \mathbf{C}^\perp$
19:    if ($\Pi^{j\in[n]} \hat{v}_{d,j}^{x_j^\perp} \neq 1_{\mathbb{G}_1}$) then return False
20:    for $j \in [n]$ do
21:       if (not dleq.Verify($\overline{\pi}_{d,i,j}, pk_i, \overline{c}_{d,i,j}, g_1, \overline{v}_{d,i,j}$)) then return False
22:    if ($\hat{v}_{d,i} \neq \prod_{j\in[n]} \overline{v}_{d,i,j}$ or $\hat{c}_{d,i} \neq \prod_{j\in[n]} \overline{c}_{d,i,j}$) then return False
23:    return True
```

and commitments by multiplying the shares and commitments encrypted with the public key of the same recipient, for example for process p_0 the aggregation of the shares results in $\hat{c}_{i,0} = \prod_{k=0}^{n-1} c_{i,0,k}$, and those of the commitment results in $\hat{v}_{i,0} = \prod_{k=0}^{n-1} v_{i,0,k}$. Following previous terminology [27], we use $\overline{\boldsymbol{c}}_{i,j}$ to refer to all the shares encrypted by process p_i with the public key of process p_j, i.e. $\overline{\boldsymbol{c}}_{i,j} = \{c_{i,j,k}\}_{k=0}^{n-1}$, and the same for $\overline{\boldsymbol{v}}_{i,j}$ and $\overline{\boldsymbol{\pi}}_{i,j}$. We show the updated RBV-broadcast protocol with this optimization in Algorithm 6. Figure 2 shows this optimization for process p_d's secret shares $c_{d,i,k}$ (the same occurs for the commitments $v_{d,i,k}$). Each column contains the secret shares of a different new input secret (i.e. $s_{d,i}$, shown at the top of the matrix). Each row contains the secret share from each different secret encrypted with the public key of the same recipient. We use $\overline{c}_{d,i}$ to refer then to the secret shares from each secret with the same recipient p_i (i.e. the i-th row of the matrix). Then, $\hat{c}_{d,i}$ is the result of multiplying each value in the i-th row (i.e. aggregated secret shares by recipient). Finally, $\hat{\mathbf{c}}_{\mathbf{d}}$ is the list of all the aggregated secret shares by recipients.

In Figure 3 we illustrate the beginning of Algorithm 6, to showcase what p_d sends in the initial message to each process. Note that p_d sends to each process a message of size $O(n)$ bits which contains $\Omega(n)$ secret inputs, thanks to the aggregation performed in Figure 2.

One can note that it is possible that Byzantine processes broadcast shares that only pass the local verification of up to $t+1$ honest processes. Nevertheless, this is not a problem because either the consensus protocol terminates with a decided bitmask that contains some secrets shared by Byzantine processes, which would mean that at least $t+1$ honest processes can decrypt their share and reconstruct all $n \cdot (t+1)$ shared secrets associated to the decided bitmask, or instead eventually there is an epoch after GST whose leader

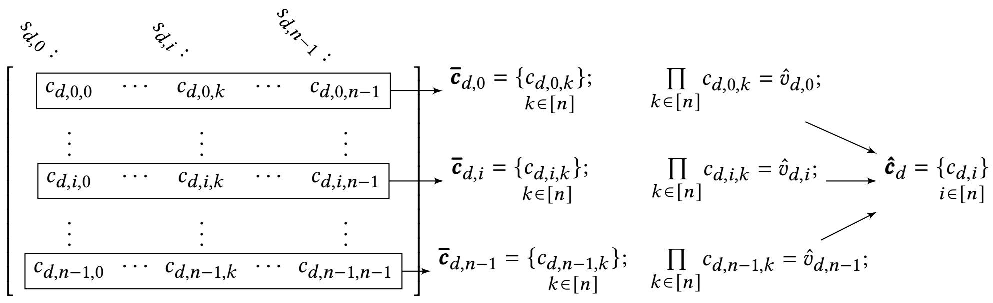
$s_{d,0}$:
$s_{d,i}$:
$s_{d,n-1}$:
$c_{d,0,0} \cdots c_{d,0,k} \cdots c_{d,0,n-1}$
$c_{d,i,0} \cdots c_{d,i,k} \cdots c_{d,i,n-1}$
$c_{d,n-1,0} \cdots c_{d,n-1,k} \cdots c_{d,n-1,n-1}$
$\bar{\boldsymbol{c}}_{d,0} = \{c_{d,0,k}\}_{k\in[n]}$; $\prod_{k\in[n]} c_{d,0,k} = \hat{v}_{d,0}$;
$\bar{\boldsymbol{c}}_{d,i} = \{c_{d,i,k}\}_{k\in[n]}$; $\prod_{k\in[n]} c_{d,i,k} = \hat{v}_{d,i}$;
$\hat{\boldsymbol{c}}_d = \{c_{d,i}\}_{i\in[n]}$
$\bar{\boldsymbol{c}}_{d,n-1} = \{c_{d,n-1,k}\}_{k\in[n]}$; $\prod_{k\in[n]} c_{d,n-1,k} = \hat{v}_{d,n-1}$;

Figure 2: Example of the aggregation of the secret shares $\{c_{d,i,k}\}$ generated from n secrets $\{s_{d,i}\}_{i\in[n]}$ locally by each process p_d (the same aggregation occurs for commitments instead of secret shares).

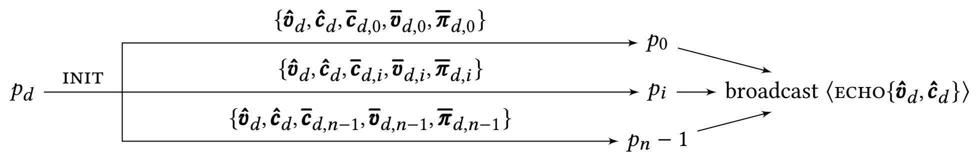
$\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \bar{\boldsymbol{c}}_{d,0}, \bar{\boldsymbol{v}}_{d,0}, \bar{\boldsymbol{\pi}}_{d,0}\}$
p_d
INIT
$\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \bar{\boldsymbol{c}}_{d,i}, \bar{\boldsymbol{v}}_{d,i}, \bar{\boldsymbol{\pi}}_{d,i}\}$
$\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d, \bar{\boldsymbol{c}}_{d,n-1}, \bar{\boldsymbol{v}}_{d,n-1}, \bar{\boldsymbol{\pi}}_{d,n-1}\}$
p_0
p_i
$p_n - 1$
broadcast $\langle$ECHO$\{\hat{\boldsymbol{v}}_d, \hat{\boldsymbol{c}}_d\}\rangle$

Figure 3: Example of the beginning of the optimized RBV-broadcast protocol for process p_d.

is an honest process and this honest process proposes a bitmask containing $t+1$ secrets RBV-delivered by all honest processes, and in this case the consensus protocol terminates in this epoch. That is, thanks to the consensus protocol it is impossible for Byzantine processes to prevent termination, or to decide a result that honest process cannot reconstruct, for $t < n/3$.

This optimization reduces the bit complexity of Kleroterion to $O(\lambda n^3)$ per decision, since one iteration generates n random outputs. We now show how to reduce the bit complexity of Kleroterion to make it quadratic per decision. We speak of the complexity per decision as the *normalized* complexity.

The optimized RBV-Broadcast showed in Algorithm 6 can be combined with the Kleroterion.Decide and Kleroterion.Reconstruct functions we showed in Algorithms 4 and 5, respectively, with the modifications that honest processes verify the correct aggregation of the received decided secrets and decrypted shares during the reconstruction, as well as replacing Pinakion.check with dleq.verify, as we did for Algorithm 6 because of their redundancy [27]. That is, within the call to $Kleroterion.verify$ of line 11 of Algorithm 5, honest processes also check $\hat{v}_{d,k} = \prod_{j\in[n]} \bar{v}_{d,k,j}$ **and** $\hat{c}_{d,k} = \prod_{j\in[n]} \bar{c}_{d,k,j}$ to verify the received shares from process p_k.

Amortized complexity. The final observation that reduces the bit (and message) complexity of Kleroterion was noted by previous work [5, 27], and relates to the time complexity of consensus. Since a linear factor of the complexity derives from the possibility that $O(t)$ leaders are faulty after GST, this means that, in the presence of a static adversary, for n consecutive executions of Kleroterion, there will be $\Omega(n)$ outputs (or $O(n^2)$ with the aforementioned aggregated inputs optimization). As such, the total bit complexity per decision and after n iterations of the protocol can be decreased by a linear factor. We define this complexity per decision as the *amortized* bit complexity.

As a result, the amortized and normalized bit complexity of Kleroterion is $O(\lambda n^2)$, same as the amortized and normalized bit complexity of SPURT. We justify in the following the advantages of a democratic protocol compared with a leader-based one.

5.2 Distributing bits and computation

To the best of our knowledge, previous PVSS-based random beacons implement a protocol that is not democratic, i.e. where all inputs are routed through the leader of the epoch. In contrast, democratic protocols like Kleroterion distribute inputs across all processes, although there is still a leader to select the subset of all inputs using a bitmask to reference inputs. Thus, the leader of a non-democratic protocol has to perform heavy computation and receive and share a significant amount of data, while our Kleroterion protocol distributes the sharing of data and computation by executing n instances of RBV-broadcast. The only message that the leader of an epoch must share compared to the rest of the processes in Kleroterion is the bitmask, of size $O(n)$ bits. We summarize the distribution of computation and communication of Kleroterion compared to SPURT in Table 4 (which is a re-instantiation of Table 1). We omit the reconstruction phase as it is equivalent in both SPURT and Kleroterion.

Distributing communication. For the case of SPURT, since the leader needs to perform the aggregations of each share of secrets, the channels to the leader transfer $O(\lambda n)$ bits per decision, all with the same recipient. Then, the leader computes the aggregated values, outputting also $O(\lambda n)$ bits to each process. For Kleroterion with the aforementioned optimizations, Kleroterion only sends $O(\lambda)$ bits per pairwise channel per decision, and distributes the verification and

Table 4: Comparison of normalized and amortized bits sent per each pairwise channel of the network and computation complexity per decision, in SPURT [27] and Kleroterion.

	Phase	Computation		Bits per channel	
		Leader	Non-leader	Leader	non-leaders
SPURT	Commitment	$O(n)$	$O(n)$	$O(\lambda n)$	$O(\lambda)$
	Aggregation	$O(n^2)$	-	-	-
	Agreement	$O(n)$	$O(n)$	$O(\lambda n)$	$O(\lambda)$
Kleroterion	Commitment	$O(n)$	$O(n)$	$O(\lambda)$	$O(\lambda)$
	Aggregation	$O(n)$	$O(n)$	$O(\lambda)$	$O(\lambda)$
	Agreement	$O(n)$	$O(n)$	$O(n)$	$O(\lambda)$

computation of aggregated values, also decreasing the computation complexity, as we show later.

This difference, sometimes referred to as the *per route* bit complexity (or per channel), means that the leader of SPURT and other non-democratic protocols will be the bandwidth of network routes to the leader, whereas Kleroterion exploits all pairwise channels of the network. Recent results prove that consensus protocols with comparable bit complexity but lower per route complexity perform at a significantly greater throughput than their counterparts [25, 26, 59, 60, 62]. Notice also that the $O(n)$ bits sent by the leader during the agreement phase of Kleroterion enables batching, as we discuss later.

Distributing computation. The democratic approach of Kleroterion also helps distributing computation between processes. In particular, while the commitment phase requires $O(n)$ exponentiations per decision (same as SPURT), the aggregation differs significantly. SPURT requires the leader to verify the PVSS shares from all processes and aggregate them. As a result, SPURT's leader performs $O(n^2)$ exponentiations per decision to verify all PVSS shares, and $O(n^2)$ multiplications to compute the aggregations, while the rest of $n-1$ non-leader processes perform no work waiting for the leader to perform these computations. In contrast, in Kleroterion each process p_i locally aggregates their n secret shares by recipient, and verify the shares from the $n-1$ other processes for which p_i is the recipient, resulting in $O(n)$ exponentiations and aggregations per decision per process, whether that is a leader or not.

Furthermore, the leader of SPURT needs to hash $O(n)$ group elements to compute the digest that will be decided during the consensus protocol. In contrast, Kleroterion's digest is just one hash of a bitmask of $O(n)$ bits. The computational complexities per decision of the remaining agreement and reconstruction phases, as well as the complexities of publicly verifying outputs, are the same for Kleroterion and SPURT. This means that Kleroterion also removes the quadratic computational bottleneck at the leader, in addition to removing the bandwidth bottleneck at the network channels involving the leader.

5.3 Decoupling proposals from consensus

Another advantage of our construction that executes first RBV-broadcast and then a consensus that references these RBV-delivered values is the possibility to make the bitmask of the consensus decision reference more information than the secret shares of that process. This does not affect correctness, as processes only decide on $t+1$ bits that they can verify and whose associated data has been reliably broadcast, and there are $n-t > t+1$ honest processes.

An example where this decoupling is useful is blockchains. In a blockchain application, processes could decide a union of blocks (also known as superblock) by reliably broadcasting these blocks and then deciding on a bitmask that represents the blocks to merge into a superblock. This blockchain could benefit from our RBV-broadcast protocol by implementing a random beacon in the blockchain (e.g. for committee sortition or as a source of randomness for other sevices) only at the additional cost of executing n Pinakion.Share iterations, but reusing the same iteration of consensus already existing in the blockchain.

6 CONCLUSION

In this paper, we presented Kleroterion, a democratic random beacon with quadratic bit complexity per output. For this purpose, we first presented Pinakion, our proposal for a PVSS scheme, that uses RBV-broadcast, a reliable broadcast implementation, in order to broadcast secret shares. Then, we justified the advantages of Kleroterion compared to recent works in that Kleroterion's democratization of SPURT by disseminating inputs without routing them through the leader removes the computational and bandwidth bottleneck at the leader, and that Kleroterion allows for batching of secret shares with other information, which can be a significant improvement in applications like blockchains, where consensus is typically used to decide on blocks of transactions.

As future work, we are working in further improving Kleroterion by adding accountability and ensuring it preserves unpredictability and bias-resistance against an adversary controlling more than t faults.

ACKNOWLEDGEMENTS

We wish to thank Daniel Collins and Sara Tucci for their feedback on earlier versions of this paper. This research is supported in part under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things",

REFERENCES

[1] 2019. RANDAO: A DAO working as RNG of Ethereum. https://github.com/randao/randao
[2] 2020. Babe: Blind Assignment for Blockchain Extension protocol. https://w3f-research.readthedocs.io/en/latest/polkadot/block-production/Babe.html
[3] 2020. Drand - a distributed randomness beacon daemon. https://github.com/drand/drand.
[4] 2020. Polkadot: Learn Randomness. https://wiki.polkadot.network/docs/en/learn-randomness
[5] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. *S&P* (2020).
[6] Marcos K Aguilera and Sam Toueg. 2012. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing* (2012).

[7] Nicolas Alhaddad, Mayank Varia, and Haibin Zhang. 2021. High-threshold avss with optimal communication complexity. *Financial Cryptography and Data Security* (2021).

[8] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. 2018. Helix: A scalable and fair consensus algorithm resistant to ordering manipulation. (2018). https://eprint.iacr.org/2018/863

[9] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. 2019. Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols. *CCS* (2019).

[10] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. 2020. *CRAFT: Composable Randomness and Almost Fairness from Time.* Technical Report. Cryptology ePrint Archive. https://eprint.iacr.org/2020/784

[11] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2020. *RandPiper – Reconfiguration-Friendly Random Beacons with Quadratic Communication.* Technical Report. Cryptology ePrint Archive. https://eprint.iacr.org/2020/1590

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* (2004).

[13] Elette Boyle, Shafi Goldwasser, and Yael Tauman Kalai. 2014. Leakage-resilient coin tossing. *Distributed computing* (2014).

[14] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* (1987).

[15] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. 2017. Proofs-of-delay and randomness beacons in ethereum. *S&P* (2017).

[16] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. *CRYPTO* (2001).

[17] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* (2005).

[18] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2021. Internet Computer Consensus. https://ia.cr/2021/632

[19] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. *Symposium on Theory of computing* (1993).

[20] Ignacio Cascudo and Bernardo David. 2017. SCRAPE: Scalable Randomness Attested by Public Entities. *Applied Cryptography and Network Security* (2017).

[21] Ignacio Cascudo and Bernardo David. 2020. ALBATROSS: Publicly AttestabLe BATched Randomness Based On Secret Sharing. *ASIACRYPT* (2020).

[22] David Chaum and Torben Pryds Pedersen. 1993. Wallet Databases with Observers. *CRYPTO* (1993).

[23] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. 2019. *Homomorphic Encryption Random Beacon.* Technical Report. Cryptology ePrint Archive. https://eprint.iacr.org/2019/1320

[24] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2021. Polygraph: Accountable Byzantine Agreement. *ICDCS* (2021).

[25] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains. *NCA* (2018).

[26] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: A Secure, Fair and Scalable Open Blockchain. *S&P* (2021).

[27] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. 2022. SPURT: Scalable Distributed Randomness Beacon with Transparent Setup. *S&P* (2022).

[28] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. *EUROCRYPT* (2018).

[29] Bernardo David, Lorenzo Gentile, and Mohsen Pourpouneh. 2022. FAST: Fair Auctions via Secret Transactions. *Applied Cryptography and Network Security* (2022).

[30] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. 2021. *GearBox: An Efficient UC Sharded Ledger Leveraging the Safety-Liveness Dichotomy.* Technical Report. Cryptology ePrint Archive. https://eprint.iacr.org/2021/211

[31] Luciano Freitas de Souza, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. 2021. RandSolomon: optimally resilient multi-party random number generation protocol. (2021). http://arxiv.org/abs/2109.04911

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* (1988).

[33] A Elrond. 2019. Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake. https://elrond.com/assets/files/elrond-whitepaper.pdf

[34] Amos Fiat and Adi Shamir. 1987. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. *CRYPTO* (1987).

[35] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2021. Efficient Asynchronous Byzantine Agreement without Private Setups. (2021). http://arxiv.org/abs/2106.07831

[36] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. *SOSP* (2017).

[37] Adam Gągol, Damian Leundefinedniak, Damian Straszak, and Michał undefinedwiundefinedtek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. *Advances in Financial Technologies* (2019).

[38] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications.* Cambridge university press.

[39] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. Dfinity technology overview series, consensus system. (2018). http://arxiv.org/abs/1805.04548

[40] Somayeh Heidarvand and Jorge L. Villar. 2009. Public Verifiability from Pairings in Secret Sharing Schemes. *Selected Areas in Cryptography* (2009).

[41] C. Edward Kelso. 2018. Bitcoin Gold Hacked for $18 Million. https://news.bitcoin.com/bitcoin-gold-hacked-for-18-million/

[42] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. *CRYPTO* (2017).

[43] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. 2020. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures. *CCS* (2020).

[44] Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. 2020. No-Dealer: Byzantine Fault-Tolerant Random Number Generator. *INFOCOM workshops* (2020).

[45] Yizhong Liu, Jianwei Liu, Marcos Antonio Vaz Salles, Zongyang Zhang, Tong Li, Bin Hu, Fritz Henglein, and Rongxing Lu. 2021. *Building Blocks of Sharding Blockchain Systems: Concepts, Approaches, and Open Problems.* Technical Report. arXiv. http://arxiv.org/abs/2102.13364

[46] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. 2020. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *PODC* (2020).

[47] Robert J. McEliece and Dilip V. Sarwate. 1981. On sharing secrets and Reed-Solomon codes. *Commun. ACM* (1981).

[48] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. *CCS* (2016).

[49] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[50] Thanh Nguyen-Van, Tuan Nguyen-Anh, Tien-Dat Le, Minh-Phuoc Nguyen-Ho, Tuong Nguyen-Van, Nhat-Quang Le, and Khuong Nguyen-An. 2019. Scalable distributed random number generation based on homomorphic encryption. *International Conference on Blockchain* (2019).

[51] David Pointcheval and Jacques Stern. 1996. Security Proofs for Signature Schemes. *EUROCRYPT* (1996).

[52] Mayank Raikwar. 2022. Competitive Decentralized Randomness Beacon Protocols. *International Symposium on Blockchain and Secure Critical Infrastructure* (2022).

[53] Jamie Redman. 2020. Bitcoin Gold 51% Attacked - Network Loses $70,000 in Double Spends. https://news.bitcoin.com/bitcoin-gold-51-attacked-network-loses-70000-in-double-spends/

[54] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* (1960).

[55] Alexandre Ruiz and Jorge L Villar. 2005. Publicly verifiable secret sharing from Paillier's cryptosystem. *Western European Workshop on Research in Cryptology* (2005).

[56] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar Weippl. 2021. *Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness.* Technical Report. https://eprints.cs.univie.ac.at/6629/1/2020-942.pdf

[57] Berry Schoenmakers. 1999. A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting. *CRYPTO* (1999).

[58] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* (1979).

[59] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput Robust BFT for Decentralized Networks. http://arxiv.org/abs/1906.05552

[60] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State Machine Replication Scalability Made Simple. *EuroSys* (2022).

[61] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. 2017. Scalable Bias-Resistant Distributed Randomness. *S&P* (2017).

[62] Gauthier Voron and Vincent Gramoli. 2019. *Dispel: Byzantine SMR with Distributed Pipelining.* Technical Report. arXiv. http://arxiv.org/abs/1912.10367

[63] Gang Wang and Mark Nixon. 2020. RandChain: Practical Scalable Decentralized Randomness Attested by Blockchain. *International Conference on Blockchain* (2020).

[64] A Yao. 1982. Theory and applications of trapdoor functions. FOCS'82. *Symposium on Foundations of Computer Science (FOCS'97)* (1982).

[65] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. *PODC* (2019).

[66] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. 2021. hbACSS: How to Robustly Share Many Secrets. https://eprint.iacr.org/2021/159

[67] Terence Zimwara. 2020. $5.6 Million Double Spent: ETC Team Finally Acknowledges the 51% Attack on Network. https://news.bitcoin.com/5-6-million-stolen-as-etc-team-finally-acknowledge-the-51-attack-on-network/