Scalable Bias-Resistant Distributed Randomness

Ewa Syta[*], Philipp Jovanovic[†], Eleftherios Kokoris Kogias[†], Nicolas Gailly[†],
Linus Gasser[†], Ismail Khoffi[‡], Michael J. Fischer[§], Bryan Ford[†]

[*]Trinity College, USA
[†]École Polytechnique Fédérale de Lausanne, Switzerland
[‡]University of Bonn, Germany
[§]Yale University, USA

Abstract—**Bias-resistant public randomness is a critical component in many (distributed) protocols. Generating public randomness is hard, however, because active adversaries may behave dishonestly to bias public random choices toward their advantage. Existing solutions do not scale to hundreds or thousands of participants, as is needed in many decentralized systems. We propose two large-scale distributed protocols, RandHound and RandHerd, which provide publicly-verifiable, unpredictable, and unbiasable randomness against Byzantine adversaries. Rand-Hound relies on an untrusted client to divide a set of randomness servers into groups for scalability, and it depends on the pigeon-hole principle to ensure output integrity, even for non-random, adversarial group choices. RandHerd implements an efficient, decentralized randomness beacon. RandHerd is structurally similar to a BFT protocol, but uses RandHound in a one-time setup to arrange participants into verifiably unbiased random secret-sharing groups, which then repeatedly produce random output at predefined intervals. Our prototype demonstrates that RandHound and RandHerd achieve good performance across hundreds of participants while retaining a low failure probability by properly selecting protocol parameters, such as a group size and secret-sharing threshold. For example, when sharding 512 nodes into groups of 32, our experiments show that RandHound can produce fresh random output after 240 seconds. RandHerd, after a setup phase of 260 seconds, is able to generate fresh random output in intervals of approximately 6 seconds. For this configuration, both protocols operate at a failure probability of at most 0.08% against a Byzantine adversary.**

I. INTRODUCTION

A reliable source of randomness that provides high-entropy output is a critical component in many protocols [11], [22]. The reliability of the source, however, is often not the only criterion that matters. In many high-stakes protocols, the unbiasability and public-verifiability of the randomness generation process are as important as ensuring that the produced randomness is good in terms of the entropy it provides [31].

More concretely, Tor hidden services [25] depend on the generation of a fresh random value each day for protection against popularity estimations and DoS attacks [34]. Anytrust-based systems, such as Herbivore [32], Dissent [60], and Vuvuzela [59], as well as sharded blockchains [23], use bias-resistant public randomness for scalability by sharding participants into smaller groups. TorPath [30] critically depends on public randomness for setting up consensus groups. Public randomness can be used to transparently select parameters for cryptographic protocols or standards, such as in the generation

of elliptic curves [2], [40], where adversaries should not be able to steer the process to select curves with weak security parameters [6]. Other use-cases for public randomness include voting systems [1] for sampling ballots for manual recounts, lotteries for choosing winning numbers, and Byzantine agreement algorithms [15], [46] for achieving scalability.

The process of generating public randomness is nontrivial, because obtaining access to sources of good randomness, even in terms of entropy alone, is often difficult and error-prone [19], [36]. One approach is to rely on randomness beacons, which were introduced by Rabin [49] in the context of contract signing, where a trusted third party regularly emits randomly chosen integers to the public. The NIST beacon [45] provides hardware-generated random output from quantum-mechanical effects, but it requires trust in their centralized beacon—a problematic assumption, especially after the Dual EC DRBG debacle [8], [54].

This work is concerned primarily with the trustworthiness, rather than the entropy, of public randomness sources. Generating public randomness without a trusted party is often desirable, especially in decentralized settings such as blockchains, where many mutually-distrustful users may wish to participate. Producing and using randomness in a distributed setting presents many issues and challenges, however, such as how to choose a subset of available beacons, or how to combine random outputs from multiple beacons without permitting bias by an active adversary. Prior approaches to randomness without trusted parties [48] employ Bitcoin [4], [13], slow cryptographic hash functions [40], lotteries [2], or financial data [21] as sources for public randomness.

Our goal is to provide bias-resistant public randomness in the familiar (t, n)-threshold security model already widely-used both in threshold cryptography [24], [47] and Byzantine consensus protocols [15]. Generating public randomness is hard, however, as active adversaries can behave dishonestly in order to bias public random choices toward their advantage, *e.g.*, by manipulating their own explicit inputs or by selectively injecting failures. Although addressing those issues is relatively straightforward for small values of $n \approx 10$ [15], [38], we address scalability challenges of using larger values of n, in the hundreds or thousands, for enhanced security in real-world scenarios. For example, scalable randomness is relevant for public cryptocurrencies [39], [44] which tend to have

hundreds to thousands of distinct miners or for countries with thousands of national banks that might want to form a national permissioned blockchain with secure random sharding.

This paper's contributions are mainly pragmatic rather than theoretical, building on existing cryptographic primitives to produce more scalable and efficient distributed randomness protocols. We introduce two scalable public-randomness generation protocols: RandHound is a "one-shot" protocol to generate a single random output on demand, while RandHerd is a randomness beacon protocol that produces a regular series of random outputs. Both protocols provide the same key security properties of unbiasability, unpredictability, availability, and third-party verifiability of their random outputs.

RandHound is a client-server randomness scavenging protocol enabling a client to gather fresh randomness on demand from a potentially large set of nearly-stateless randomness servers, preferably run by independent parties. A party that occasionally requires trustworthy public randomness, such as a lottery association, can use RandHound to produce a random output that includes contributions of – and trustworthiness attestations from – all participating servers. The RandHound client (*e.g.*, the lottery association) first publicly commits to the parameters of a unique RandHound protocol run, such as the time and date of the lottery and the set of servers involved, so a malicious client cannot bias the result by secretly rerunning the protocol. The client then splits the servers into balanced subgroups for scalability. Each subgroup uses publicly verifiable secret sharing (PVSS) [52], [56] to produce secret inputs such that an honest threshold of participants can later recover them and form a third-party-verifiable proof of their validity. To tolerate server failures, the client selects a subset of secret inputs from each group. Application of the pidgeonhole principle ensures ensures the integrity of RandHound's final output even if some subgroups are compromised, *e.g.*, due to biased grouping. The client commits to his choice of secrets, to prevent equivocation, by obtaining a collective signature [58] from participating servers. After the servers release the selected secrets, the client combines and publishes the collective random output along with a third-party verifiable transcript of the protocol run. Anyone can subsequently check this transcript to verify that the random output is trustworthy and unbiased, provided not too many servers were compromised.

RandHerd is a complementary protocol enabling a potentially large collection of servers to form a distributed public randomness beacon, which proactively generates a regular series of public random outputs. RandHerd runs continually and need not be initiated by any client, but requires stateful servers. No single or sub-threshold group of failing or malicious servers can halt the protocol, or predict or significantly bias its output. Clients can check the trustworthiness of any published beacon output with a single, efficient check of one collective signature [58]. RandHerd first invokes RandHound once, at setup or reconfiguration time, to divide the set of servers securely into uniformly random groups, and to generate a short-term aggregate public key used to produce

and verify individual beacon outputs. RandHerd subsequently uses a threshold collective signing protocol based on Shamir secret sharing [9], [53], to generate random outputs at regular intervals. Each of RandHerd's random outputs doubles as a collective Schnorr signature [57], [58], which clients can validate efficiently against the group's aggregate public key.

The dominant cost in both protocols is publicly verifiable secret sharing (PVSS), which normally incurs $O(n^3)$ communication and computation costs on each of n participants. RandHound and RandHerd run PVSS only among smaller groups, however, whose configured size c serves as a security parameter. RandHound therefore reduces asymptotic cost to $O(n)$ if c is constant. By leveraging efficient tree-structured communication, RandHerd further reduces the cost of producing successive beacon outputs to $O(\log n)$ per server.

We implemented the RandHound and RandHerd protocols in Go, and made these implementations freely available as part of the EPFL DEDIS lab's Cothority framework.[1] Experiments with our prototype implementations show that, among a collective of 512 globally-distributed servers divided into groups of 32, RandHerd can produce a new 32-byte collective random output every 6 seconds, following a one-time setup process using RandHound that takes approximately 260 seconds. The randomness verification overhead of RandHerd is equivalent to verifying a single Schnorr multisignature [51], typically less than 100 bytes in size, which clients can check in constant time. Using RandHound alone to produce a random output on demand, it takes approximately 240 seconds to produce randomness and approximately 76 seconds to verify it using the produced 4 MByte transcript. In this configuration, a Byzantine adversary can compromise the availability of either protocol with a probability of at most 0.08%.

This paper is organized as follows. Section II explores background and motivation for public randomness. Sections III and IV introduces the design and security properties of RandHound and RandHerd, respectively. Section V evaluates the prototype implementations of both protocols. Finally, Section VI summarizes related work and Section VII concludes.

II. BACKGROUND AND MOTIVATION

We first introduce notation and summarize techniques for secret sharing and Schnorr signing, which RandHound and RandHerd build on. We then consider a series of strawman protocols illustrating the key challenges in distributed randomness generation of commitment, selective aborts, and malicious secret shares. We end with RandShare, a protocol that offers the desired properties, but unlike RandHound and RandHerd is not third-party verifiable and does not scale well.

For the rest of the work, we denote by $\mathcal{G}$ a multiplicatively written cyclic group of order q with generator G, where the set of non-identity elements in $\mathcal{G}$ is written as $\mathcal{G}^*$. We denote by $(x_i)_{i \in I}$ a vector of length $|I|$ with elements x_i, for $i \in I$. Unless stated otherwise, we denote the private key of a node i by x_i and the corresponding public key by $X_i = G^{x_i}$.

---

[1] https://github.com/dedis/cothority

A. Publicly Verifiable Secret-Sharing

A (t, n)-secret sharing scheme [9], [53] enables an honest dealer to share a secret s among n trustees such that any subset of t honest trustees can reconstruct s, whereas any subset smaller than t learns nothing about s. Verifiable secret-sharing (VSS) [20], [26], [50] adds protection from a dishonest dealer who might intentionally produce bad shares and prevent honest trustees from recovering the same, correct secret.

A publicly verifiable secret sharing (PVSS) [52], [56] scheme makes it possible for any party to verify secret-shares without revealing any information about the shares or the secret. During the share distribution phase, for each trustee i, the dealer produces an encrypted share $E_i(s_i)$ along with a non-interactive zero-knowledge proof (NIZK) [18], [27], [28] that $E_i(s_i)$ correctly encrypts a valid share s_i of s. During the reconstruction phase, trustees recover s by pooling t properly-decrypted shares. They then publish s along with all shares and NIZK proofs that show that the shares were properly decrypted. PVSS runs in three steps:

1) The dealer chooses a degree $t - 1$ secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ and creates, for each trustee $i \in \{1, \ldots, n\}$, an encrypted share $\widehat{S}_i = X_i^{s(i)}$ of the shared secret $S_0 = G^{s(0)}$. He also creates commitments $A_j = H^{a_j}$, where $H \neq G$ is a generator of $\mathcal{G}$, and for each share a NIZK encryption consistency proof $\widehat{P}_i$. Afterwards, he publishes $\widehat{S}_i$, $\widehat{P}_i$, and A_j.

2) Each trustee i verifies his share $\widehat{S}_i$ using $\widehat{P}_i$ and A_j, and, if valid, publishes the decrypted share $S_i = (\widehat{S}_i)^{x_i^{-1}}$ together with a NIZK decryption consistency proof P_i.

3) The dealer checks the validity of S_i against P_i, discards invalid shares and, if there are at least t out of n decrypted shares left, recovers the shared secret S_0 through Lagrange interpolation.

B. Schnorr Signature Schemes

RandHound and RandHerd rely on variations of the well-known Schnorr (multi-)signature schemes [3], [42], [51].

1) Threshold Signing: TSS [57] is a distributed (t, n)-threshold Schnorr signature scheme. TSS allows any subset of t signers to produce a valid signature. During setup, all n trustees use VSS to create a long-term shared secret key x and a public key $X = G^x$. To sign a statement S, the n trustees first use VSS to create a short-term shared secret v and a commitment $V = G^v$ and then compute the challenge $c = H(V \parallel S)$. Afterwards, each trustee i uses his shares v_i and x_i of v and x, respectively, to create a partial response $r_i = v_i - cx_i$. Finally, when t out of n trustees collaborate they can reconstruct the response r through Lagrange interpolation. The tuple (c, r) forms a regular Schnorr signature on S, which can be verified against the public key X.

2) Collective Signing: CoSi [58] enables a set of witnessing servers coordinated by a leader to efficiently produce a collective Schnorr signature (c, r) under an aggregate public key $\widehat{X} = \prod_{i=0}^{n-1} X_i$. CoSi scales Schnorr multisignatures to thousands of participants by using aggregation techniques and communication trees.

A CoSi round runs in four steps over two round-trips between a leader and his witnesses. To sign a statement S sent down the communication tree by the leader, each server i computes a commitment $V_i = G^{v_i}$ and in a bottom-up process, all commitments are aggregated until the leader holds the aggregate commit $\widehat{V} = \prod_{i=0}^{n-1} V_i$. Once the leader computes and multicasts down the tree the collective challenge $c = H(\widehat{V} \parallel S)$, each server i responds with a partial response $r_i = v_i - cx_i$. Lastly, the servers aggregate all responses into $r = \sum_{i=0}^{n-1} r_i$ in a final bottom-up process.

C. Insecure Approaches to Public Randomness

For expositional clarity, we now summarize a series of inadequate strawman designs: (I) a naive, trivially insecure design, (II) one that uses a commit-then-reveal process to ensure unpredictability but fails to be unbiasable, and (III) one that uses secret sharing to ensure unbiasability in an honest-but-curious setting, but is breakable by malicious participants.

Strawman I. The simplest protocol for producing a random output $r = \bigoplus_{i=0}^{n-1} r_i$ requires each peer i to contribute their secret input r_i under the (false) assumption that a random input from any honest peer would ensure unbiasability of r. However, a dishonest peer j can force the output value to be $\hat{r}$ by choosing $r_j = \hat{r} \bigoplus_{i:i \neq j} r_i$ upon seeing all other inputs.

Strawman II. To prevent the above attack, we want to force each peer to commit to their chosen input *before* seeing other inputs by using a simple *commit-then-reveal* approach. Although the output becomes unpredictable as it is fixed during the commitment phase, it is not unbiasable because a dishonest peer can choose not to reveal his input upon seeing all other openings of committed inputs. By repeatedly forcing the protocol to restart, the dishonest peer can obtain output that is beneficial for him, even though he cannot choose its exact value. The above scenario shows an important yet subtle difference between an output that is *unbiased* when a single, successful run of the protocol is considered, and an output that is *unbiasable* in a more realistic scenario, when the protocol repeats until some output is produced. An attacker's ability to re-toss otherwise-random coins he does not like is central to the reason peer-to-peer networks that use cryptographic hashes as participant IDs are vulnerable to clustering attacks [41].

Strawman III. To address this issue, we wish to ensure that a dishonest peer either cannot force the protocol to abort by refusing to participate, or cannot benefit from doing so. Using a (t, n)-secret sharing scheme, we can force the adversary to commit to his action *before* knowing which action is favorable to him. First, all n peers, where at most f are dishonest, distribute secret shares of their inputs using a $t = f + 1$ recovery threshold. Only after each peer receives n shares will they reconstruct their inputs and generate r. The threshold $t = f + 1$ prevents a dishonest peer from learning anything about the output value. Therefore, he must blindly choose to abort the protocol or to distribute his share. Honest peers can then complete the protocol even if he stops participating upon

seeing the recovered inputs. Unfortunately, a dishonest peer can still misbehave by producing bad shares, preventing honest peers from successfully recovering identical secrets.

D. RandShare: Small-Scale Unbiasable Randomness Protocol

RandShare is an unbiasable randomness protocol that ensures unbiasability, unpredictability, and availability, but is practical only at small scale due to $O(n^3)$ communication overhead. RandShare introduces key concepts that we will reuse in the more scalable RandHound protocol (Section III).

RandShare extends the approach for distributed key-generation in a synchronous model of Gennaro et al. [29] by adopting a point-of-no-return strategy implemented through the concept of a *barrier*, a specific point in the protocol execution after which the protocol always completes successfully, and by extending it to the asynchronous setting, where the adversary can break timing assumptions [14], [15].

In RandShare, the protocol output is unknown but fixed as a function of $f+1$ inputs. After the barrier point, the protocol output cannot be changed and all honest peers eventually output the previously fixed value, regardless of the adversary's behavior. In RandShare, we define the barrier at the point where the first honest member reveals the shares he holds.

We assume a Byzantine adversary and an asynchronous network where messages are eventually delivered. Let $N = \{1, \ldots, n\}$ denote the list of peers that participate in Rand-Share and $n = 3f + 1$, where f is the number of dishonest peers. Let $t = f + 1$ be the VSS threshold. We assume every peer has a copy of a public key X_j for all $j \neq i$, and that only valid, properly-signed messages are accepted.

Each RandShare peer $i \in N$ executes the following steps:

1. Share Distribution.
1) Select coefficients $a_{ik} \in_R \mathbb{Z}_q^*$ of a degree $t-1$ secret sharing polynomial $s_i(x) = \sum_{k=0}^{t-1} a_{ik} x^k$. The secret to be shared is $s_i(0) = a_{i0}$.
2) Compute polynomial commitments $A_{ik} = G^{a_{ik}}$, for all $k \in \{0, \ldots, t-1\}$, and calculate secret shares $s_i(j)$ for all $j \in N$.
3) Securely send $s_i(j)$ to peer $j \neq i$ and start a Byzantine agreement (BA) run on $s_i(0)$, by broadcasting $\widehat{A}_i = (A_{ik})_{k \in \{0, \ldots, t-1\}}$.

2. Share Verification.
1) Initialize a bit-vector $V_i = (v_{i1}, \ldots, v_{in})$ to zero, to keep track of valid secrets $s_j(0)$ received. Then wait until a message with share $s_j(i)$ from each $j \neq i$ has arrived.
2) Verify that each $s_j(i)$ is valid using $\widehat{A}_j$. This may be done by checking that $S_j(i) = G^{s_j(i)}$ where:

$$S_j(x) = \prod_{k=0}^{t-1} A_{jk}^{x^k} = G^{\sum_{k=0}^{t-1} a_{jk} x^k} = G^{s_j(x)}$$

3) If verification succeeds, confirm $s_j(i)$ by broadcasting the prepare message $(\mathrm{p}, i, j, 1)$ as a positive vote on the BA instance of $s_j(0)$. Otherwise, broadcast $(\mathrm{p}, i, j, s_j(i))$ as a negative vote. This also includes the scenario when $\widehat{A}_j$ was never received.

4) If there are at least $2f+1$ positive votes for secret $s_j(0)$, broadcast $(\mathrm{c}, i, j, 1)$ as a positive commitment. If there are at least $f+1$ negative votes for secret $s_j(0)$, broadcast $(\mathrm{c}, i, j, 0)$ as a negative commitment.
5) If there are at least $2f+1$ commits (c, i, j, x) for secret $s_j(0)$, set $v_{ij} = x$. If $x = 1$, consider the secret recoverable else consider secret $s_j(0)$ invalid.

3. Share Disclosure.
1) Wait until a decision has been taken for all entries of V_i and determine the number of 1-entries n' in V_i.
2) If $n' > f$, broadcast for each 1-entry j in V_i the share $s_j(i)$ and abort otherwise.

4. Randomness Recovery.
1) Wait until at least t shares for each $j \neq i$ have arrived, recover the secret sharing polynomial $s_j(x)$ through Lagrange interpolation, and compute the secret $s_j(0)$.
2) Compute and publish the collective random string as:

$$Z = \bigoplus_{j=1}^{n'} s_j(0)$$

RandShare achieves *unbiasability*, because the secret sharing threshold $t = f + 1$ prevents dishonest peers from recovering the honest peers' secrets before the barrier. The Byzantine agreement procedures ensure that all honest peers have a consistent copy of V_i and therefore know which $n' > f$ secrets will be recovered after the barrier or if the protocol run has already failed as $n' \leq f$. Furthermore, if at least $f + 1$ honest members sent a success message for each share, and thus Byzantine agreement (with at least $2f + 1$ prepares) has been achieved on the validity of these shares, each honest peer will be able to recover *every* other peer's secret value. *Unpredictability* follows from the fact that the final random string Z contains $n' \geq f + 1$ secrets; there are at most f malicious peers, and no honest peer will release his shares before the barrier. *Availability* is ensured because $f+1$ honest nodes out of the total $2f+1$ positive voters are able to recover the secrets, given the secret-sharing threshold $t = f + 1$, without the collaboration of the dishonest nodes.

III. RandHound: Scalable, Verifiable Randomness Scavenging

This section presents RandHound, a scalable client/server protocol for producing public, verifiable, unbiasable randomness. RandHound enables a client, who initiates the protocol, to "scavenge" public randomness from an arbitrary collection of servers. RandHound uses a commit-then-reveal approach to generate randomness, implemented via publicly verifiable secret sharing (PVSS) [52], and it uses CoSi [58] as a witnessing mechanism to fix the protocol output and prevent client equivocation. We first provide an overview of RandHound and introduce the notation and threat model. We then describe randomness generation and verification in detail, analyze the protocol's security properties, and discuss protocol extensions.

A. Protocol Overview

RandHound employs a client/server model, in which a client invokes the services of a set of RandHound servers to produce a random value. RandHound assumes the same threat model as RandShare, *i.e.*, that at most f out of at least $3f+1$ participants are dishonest. If the client is honest, we allow at most f servers to be malicious and if the adversary controls the client then we allow at most $f - 1$ malicious servers. We assume that dishonest participants can send different but correctly signed messages to honest participants in stages where they are supposed to broadcast the same message to all. Furthermore, we assume that the goal of the adversary is to bias or DoS-attack the protocol run in the honest-client scenario, and to bias the output in the malicious-client scenario.

We assume the client gets only one attempt to run Rand-Hound. A dishonest client might try to run the protocol many times until he obtains a favorable output. However, each protocol run uses a session configuration file C that uniquely identifies a protocol run and binds it to the intended purpose of the random output. To illustrate RandHound's deployment model, the client might be a lottery authority, which must commit ahead of time to all lottery parameters including the time and date of the lottery. A cryptographic hash of the configuration parameters in C uniquely identifies the RandHound protocol instance. If that protocol run fails to produce an output, this failure triggers an alarm and an investigation, and not a silent re-run of the protocol.

Honest RandHound servers enforce this "one-shot" rule by remembering and refusing to participating in a second protocol run with session configuration C until the time-window defined by C has passed. This memory of having recently participated in a session for configuration C is the only state RandHound servers need to store for significant time; the servers are otherwise largely stateless.

RandHound improves on RandShare's lack of scalability by sharing secrets not directly among all other servers but only within smaller groups of servers. RandHound servers share their secrets only with their respective group members, decreasing the number of shares they create and transmit. This reduces the communication and computational overhead from $O(n^3)$ to $O(nc^2)$, where c is the average (constant) size of a group. The client arranges the servers into disjoint groups. The protocol remains secure even if the client chooses a non-random adversarial grouping, however, because the client must employ all groups and the pidgeonhole principle ensures that at least one group is secure.

Each server chooses its random input value and creates shares only for other members of the same group using PVSS. The server sends the encrypted shares to the client together with the NIZK proofs. The client chooses a subset of server inputs from each group, omitting servers that did not respond on time or with proper values, thus fixing each group's secret and consequently the output of the protocol. After the client receives a sign-off on his choice of inputs in a global run of CoSi, the servers decrypt and send their shares to the client.

The client, in turn, combines the recovered group secrets to produce the final random output Z. The client documents the run of the protocol in a log L, or *transcript*, by recording the messages he sends and receives. The transcript serves as a third party verifiable proof of the produced randomness. Fig. 1 gives an overview on the RandHound design.
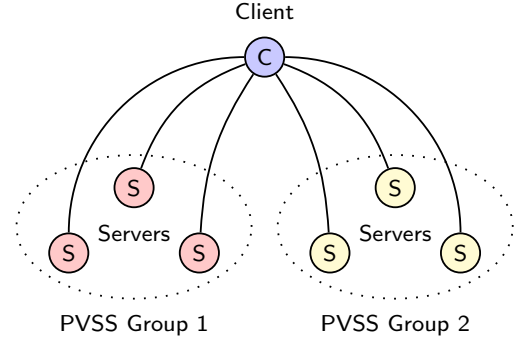

Fig. 1. An overview of the RandHound design.

B. Description

Let $\mathcal{G}$ be a group of large prime order q with generator G. Let $N = \{0, \ldots, n-1\}$ denote the list of nodes, let $S = N \setminus \{0\}$ denote the list of servers and let f be the maximum number of permitted Byzantine nodes. We require that $n = 3f+1$. We set (x_0, X_0) as the key pair of the client and (x_i, X_i) as the one of server $i > 0$. Further let $T_l \subset S$, with $l \in \{0, \ldots, m-1\}$, be pairwise disjoint trustee groups and let $t_l = \lfloor |T_l|/3 \rfloor + 1$ be the secret sharing threshold for group T_l.

The publicly available session configuration is denoted by $C = (X, T, f, u, w)$, where $X = (X_0, \ldots, X_{n-1})$ is the list of public keys, $T = (T_0, \ldots, T_{m-1})$ is the server grouping, u is a purpose string, and w is a timestamp. We call $H(C)$ the session identifier. The session configuration and consequently the session identifier have to be unique for each protocol run. We assume that all nodes know the list of public keys X.

The output of RandHound is a random string Z which is publicly verifiable through a transcript L.

1) Randomness Generation: RandHound's randomness-generation protocol has seven steps and requires three round trips between the client and the servers; see Figure 2 for an overview. All exchanged messages are signed by the sending party, messages from the client to servers include the session identifier, and messages from servers to the client contain a reply identifier that is the hash of the previous client message. We implicitly assume that client and servers always verify message signatures and session and reply identifiers and that they mark non-authentic or replayed messages and ignore them from the rest of the protocol run.

RandHound consists of three *inquiry-response* phases between the client and the servers followed by the client's randomness recovery.

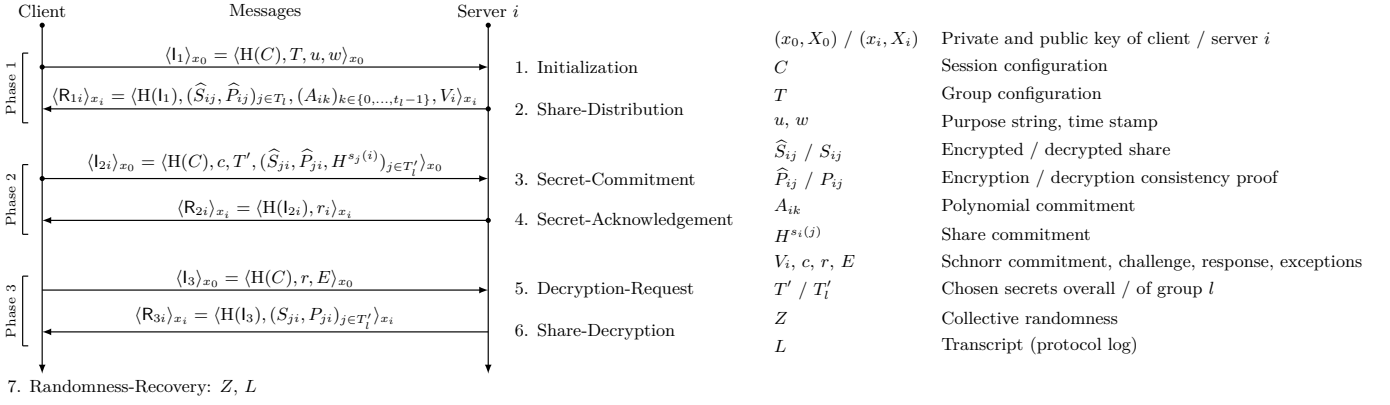1) **Initialization (Client).** The client initializes a protocol run by executing the following steps:

Client	Messages	Server i		
	$\langle I_1 \rangle_{x_0} = \langle H(C), T, u, w \rangle_{x_0}$		1. Initialization	$(x_0, X_0) \ / \ (x_i, X_i)$

Fig. 2. An overview of the RandHound randomness generation process

Glossary (right side of figure):

Symbol	Description
$(x_0, X_0) \ / \ (x_i, X_i)$	Private and public key of client / server i
C	Session configuration
T	Group configuration
u, w	Purpose string, time stamp
$\widehat{S}_{ij} \ / \ S_{ij}$	Encrypted / decrypted share
$\widehat{P}_{ij} \ / \ P_{ij}$	Encryption / decryption consistency proof
A_{ik}	Polynomial commitment
$H^{s_i(j)}$	Share commitment
V_i, c, r, E	Schnorr commitment, challenge, response, exceptions
$T' \ / \ T'_l$	Chosen secrets overall / of group l
Z	Collective randomness
L	Transcript (protocol log)

Protocol steps (figure):

1. Initialization
2. Share-Distribution
3. Secret-Commitment
4. Secret-Acknowledgement
5. Decryption-Request
6. Share-Decryption
7. Randomness-Recovery: Z, L

Messages:
$\langle I_1 \rangle_{x_0} = \langle H(C), T, u, w \rangle_{x_0}$
$\langle R_{1i} \rangle_{x_i} = \langle H(I_1), (\widehat{S}_{ij}, \widehat{P}_{ij})_{j \in T_l}, (A_{ik})_{k \in \{0, \ldots, t_l - 1\}}, V_i \rangle_{x_i}$
$\langle I_{2i} \rangle_{x_0} = \langle H(C), c, T', (\widehat{S}_{ji}, \widehat{P}_{ji}, H^{s_j(i)})_{j \in T'_l} \rangle_{x_0}$
$\langle R_{2i} \rangle_{x_i} = \langle H(I_{2i}), r_i \rangle_{x_i}$
$\langle I_3 \rangle_{x_0} = \langle H(C), r, E \rangle_{x_0}$
$\langle R_{3i} \rangle_{x_i} = \langle H(I_3), (S_{ji}, P_{ji})_{j \in T'_l} \rangle_{x_i}$

---

a) Set the values in C and choose a random integer $r_T \in_R \mathbb{Z}_q$ as a seed to pseudorandomly create a balanced grouping T of S. Record C in L.

b) Prepare the message

$$\langle I_1 \rangle_{x_0} = \langle H(C), T, u, w \rangle_{x_0},$$

record it in L, and broadcast it to all servers.

2) **Share Distribution (Server).** To distribute shares, each trustee $i \in T_l$ executes step 1 of PVSS:

a) Map $H(C)$ to a group element $H \in \mathcal{G}^*$, set $t_l = \lfloor |T_l|/3 \rfloor + 1$, and (randomly) choose a degree $t_l - 1$ secret sharing polynomial $s_i(x)$. The secret to-be-shared is $S_{i0} = G^{s_i(0)}$.

b) Create polynomial commitments A_{ik}, for all $k \in \{0, \ldots, t_l - 1\}$, and compute encrypted shares $\widehat{S}_{ij} = X_j^{s_i(j)}$ and consistency proofs $\widehat{P}_{ij}$ for all $j \in T_l$.

c) Choose $v_i \in_R \mathbb{Z}_q$ and compute $V_i = G^{v_i}$ as a Schnorr commitment.

d) Prepare the message

$$\langle R_{1i} \rangle_{x_i} = \langle H(I_1), (\widehat{S}_{ij}, \widehat{P}_{ij})_{j \in T_l}, (A_{ik})_{k \in \{0, \ldots, t_l-1\}}, V_i \rangle_{x_i}$$

and send it back to the client.

3) **Secret Commitment (Client).** The client commits to the set of shared secrets that contribute to the final random string, and asks servers to co-sign his choice:

a) Record each received $\langle R_{1i} \rangle_{x_i}$ message in L.

b) Verify all $\widehat{S}_{ij}$ against $\widehat{P}_{ij}$ using X_i and A_{ik}. Buffer each (correct) $H^{s_i(j)}$ created in the process. Mark each share that does not pass the verification as invalid, and do not forward the corresponding tuple $(\widehat{S}_{ij}, \widehat{P}_{ij}, H^{s_i(j)})$ to the respective trustee.

c) Create the commitment to the final list of secrets $T' = (T'_0, \ldots, T'_{m-1})$ by randomly selecting $T'_l \subset T_l$ such that $|T'_l| = t_l$ for all $l \in \{0, \ldots, m-1\}$.

d) Compute the aggregate Schnorr commit $V = \prod_i V_i$ and the Schnorr challenge $c = H(V \parallel H(C) \parallel T')$.

e) Prepare the message

$$\langle I_{2i} \rangle_{x_0} = \langle H(C), c, T', (\widehat{S}_{ji}, \widehat{P}_{ji}, H^{s_j(i)})_{j \in T'_l} \rangle_{x_0},$$

record it in L, and send it to trustee $i \in T_l$.

4) **Secret Acknowledgment (Server).** Each trustee $i \in T_l$ acknowledges the client's commitment by executing the following steps:

a) Check that $|T'_l| = t_l$ for each T'_l in T' and that $f+1 \leq \sum_{l=0}^{m-1} t_l$. Abort if any of those conditions does not hold.

b) Compute the Schnorr response $r_i = v_i - cx_i$.

c) Prepare the message

$$\langle R_{2i} \rangle_{x_i} = \langle H(I_{2i}), r_i \rangle_{x_i}$$

and send it back to the client.

5) **Decryption Request (Client).** The client requests the decryption of the secrets from the trustees by presenting a valid Schnorr signature on his commitment:

a) Record each received $\langle R_{2i} \rangle_{x_i}$ message in L.

b) Compute the aggregate Schnorr response $r = \sum_i r_i$ and create a list of exceptions E that contains information on missing server commits and/or responses.

c) Prepare the message

$$\langle I_3 \rangle_{x_0} = \langle H(C), r, E \rangle_{x_0},$$

record it in L, and broadcast it to all servers.

6) **Share Decryption (Server).** To decrypt received shares, each trustee $i \in T_l$ performs step 2 of PVSS:

a) Check that (c, r) forms a valid Schnorr signature on T' taking exceptions recorded in E into account and verify that at least $2f + 1$ servers signed. Abort if any of those conditions does not hold.

b) Check for all $j \in T'_l$ that $\widehat{S}_{ji}$ verifies against $\widehat{P}_{ji}$ using $H^{s_j(i)}$ and public key X_i.

c) If the verification fails, mark $\widehat{S}_{ji}$ as invalid and do not decrypt it. Otherwise, decrypt $\widehat{S}_{ji}$ by computing $S_{ji} = (\widehat{S}_{ji})^{x_i^{-1}} = G^{s_j(i)}$ and create a decryption consistency proof P_{ji}.

d) Prepare the message

$$\langle R_{3i} \rangle_{x_i} = \langle H(I_3), (S_{ji}, P_{ji})_{j \in T'_l} \rangle_{x_i}$$

and send it back to the client.

7) **Randomness Recovery (Client).** To construct the collective randomness, the client performs step 3 of PVSS:
   a) Record all received $\langle \mathsf{R}_{3i} \rangle_{x_i}$ messages in L.
   b) Check each share S_{ji} against P_{ji} and mark invalid ones.
   c) Use Lagrange interpolation to recover the individual S_{i0} that have enough valid shares S_{ij} and abort if even a single one of the secrets previously committed to in T' cannot be reconstructed.
   d) Compute the collective random value as

$$Z = \prod_{i \in \bigcup T_l'} S_{i0} \ ,$$

   and publish Z and L.

2) Randomness Verification: A verifier who wants to check the validity of the collective randomness Z against the transcript

$$L = (C, \langle \mathsf{I}_1 \rangle_{x_0}, \langle \mathsf{R}_{1i} \rangle_{x_i}, \langle \mathsf{I}_{2i} \rangle_{x_0}, \langle \mathsf{R}_{2i} \rangle_{x_i}, \langle \mathsf{I}_3 \rangle_{x_0}, \langle \mathsf{R}_{3i} \rangle_{x_i})$$

has to perform the following steps:
1) Verify the values of arguments included in the session configuration $C = (X, T, f, u, w)$. Specifically, check that $|X| = n = 3f + 1$, that groups T_l defined in T are non-overlapping and balanced, that $|X| = \sum_{l=0}^{m-1} |T_l|$, that each group threshold satisfies $t_l = |T_l|/3 + 1$, that u and w match the intended use of Z, and that the hash of C matches $\mathrm{H}(C)$ as recorded in the messages.
2) Verify all signatures of $\langle \mathsf{I}_1 \rangle_{x_0}$, $\langle \mathsf{R}_{1i} \rangle_{x_i}$, $\langle \mathsf{I}_{2i} \rangle_{x_0}$, $\langle \mathsf{R}_{2i} \rangle_{x_i}$ $\langle \mathsf{I}_3 \rangle_{x_0}$, and $\langle \mathsf{R}_{3i} \rangle_{x_i}$. Ignore invalid messages for the rest of the verification.
3) Verify that $\mathrm{H}(\mathsf{I}_1)$ matches the hash recorded in R_{1i}. Repeat for I_{2i} and R_{2i}, and I_3 and R_{3i}. Ignore messages that do not include the correct hash.
4) Check that T' contains at least $f + 1$ secrets, that the collective signature on T' is valid and that at least $2f + 1$ servers contributed to the signature (taking into account the exceptions in E).
5) Verify each recorded encrypted share $\widehat{S}_{ij}$, whose secret was chosen in T', against the proof $\widehat{P}_{ij}$ using X_i and A_{ik}. Abort if there are not enough shares for any secret chosen in T'.
6) Verify each recorded decrypted share S_{ij} against the proof P_{ij} where the corresponding $\widehat{S}_{ij}$ was found to be valid. Abort if there are not enough shares for any secret chosen in T'.
7) Verify Z by recovering Z' from the recovered individual secrets S_{i0} and by checking that $Z = Z'$. If the values are equal, then the collective randomness Z is valid. Otherwise, reject Z.

C. Security Properties

RandHound provides the following security properties:
1) **Availability.** For an honest client, the protocol successfully completes and produces the final random output Z with high probability.

2) **Unpredictability.** No party learns anything about the final random output Z, except with negligible probability, until the secret shares are revealed.
3) **Unbiasability.** The final random output Z represents an unbiased, uniformly random value, except with negligible probability.
4) **Verifiability.** The collective randomness Z is third-party verifiable against the transcript L, that serves as an unforgeable attestation that the documented set of participants ran the protocol to produce the one-and-only random output Z, except with negligible probability.

In the discussion below, we assume that each honest node follows the protocol and that all cryptographic primitives RandHound uses provide their intended security properties. Specifically, the (t, n)-PVSS scheme ensures that a secret can be recovered only by using a minimum of t shares and that the shares do not leak information about the secret.

Availability. Our goal is to ensure that an honest client can successfully complete the protocol, even in the presence of adversarial servers that misbehave arbitrarily, including by refusing to participate. A dishonest client can always abort the protocol, or simply not run it, so we do not consider a "self-DoS" by the client to be an attack on availability. In the remaining security properties, we can thus restrict our concern to attacks in which a dishonest client might corrupt (*e.g.* bias) the output without affecting the output's availability.

According to the protocol specification, an honest client randomly assigns (honest and dishonest) nodes to their groups. Therefore, each group's ratio of honest to dishonest nodes will closely resemble the overall ratio of honest to dishonest nodes in the entire set. Given that $n = 3f + 1$, the expected number of nodes in a group T_l is about $3f/m$. The secret-sharing threshold of $t_l = |T_l|/3 + 1 = (3f/m)/3 + 1 = f/m + 1$ enables $2f/m$ honest nodes in each group to recover its group secret without the collaboration of malicious nodes. This ensures availability, with high probability, when the client is honest. Section V-C analyzes of the failure probability of a RandHound run for different parameter configurations.

Unpredictability. We want to ensure that output Z remains unknown to the adversary until step 7 of the protocol, when honest nodes decrypt and reveal the secret shares they hold.

The random output Z is a function of m group secrets, where each group contributes exactly one secret that depends on t_l inputs from group members. Further, each input is recoverable using PVSS with t_l shares. In order to achieve unpredictability, there must be at least one group secret that remains unknown to the adversary until step 7.

We will show that there exists at least one group for which the adversary cannot prematurely recover the group's secret. An adversary who controls the dishonest client can deviate from the protocol description and arbitrarily assign nodes to groups. Assuming that there are h honest nodes in total and m groups, then by the generalized pigeonhole principle, regardless of how the dishonest client assigns the groups, there will be at least one group which contains at least $\lceil h/m \rceil$ nodes. In other words, there must be at least

one group with at least an average number of honest nodes. Therefore, we set the threshold for secret recovery for each group l such that the number of nodes needed to recover the group secret contains at least one honest node, that is, $|T_l| - h/m + 1 = f/m + 1$. In RandHound, we have $n = 3f + 1$ and $t_l = |T_l|/3 + 1 = (3f/m)/3 + 1 = f/m + 1$ as needed.

Consequently, the adversary will control at most $m - 1$ groups and obtain at most $m - 1$ group secrets. Based on the properties of PVSS, and the fact that Z is a function of all m group secrets, the adversary cannot reconstruct Z without the shares held by honest nodes that are only revealed in step 7.

Unbiasability. We want to ensure that an adversary cannot influence the value of the random output Z.

In order to prevent the adversary from controlling the output Z, we need to ensure that there exists at least one group for which the adversary does not control the group's secret. If, for each group, the adversary can prematurely recover honest nodes' inputs to the group secret and therefore be able to prematurely recover all groups' secrets, then the adversary can try many different valid subsets of the groups' commits to find the one that produces the Z most beneficial to him. If, for each group, the adversary can exclude honest nodes from contributing inputs to the group secret, then the adversary has full control over all group secrets, hence Z.

As argued in the discussion of unpredictability, there exists at least one group for which the adversary does not control its group secret. Furthermore, the requirement that the client has to select t_l inputs from each group in his commitment T' ensures that at least $\sum_{l=0}^{m-1} t_l = \sum_{l=0}^{m-1} f/m + 1 = f + m$ inputs contribute to the group secrets, and consequently to the output Z. Combining these two arguments, we know that there is at least one group that is not controlled by the adversary and at least one honest input from that group contributes to Z. As a result, the honest member's input randomizes the group's secret and Z, regardless of the adversary's actions.

Lastly, the condition that at least $2f + 1$ servers must sign off on the client's commitment T' ensures that a malicious client cannot arrange malicious nodes in such a way that would enable him to mount a view-splitting attack. Without that last condition the adversary could use different arrangements of honest and dishonest inputs that contribute to Z and generate multiple collective random values with valid transcripts from which he could choose and release his preferred one.

Verifiability. In RandHound, only the client obtains the final random output Z. In order for Z to be usable in other contexts and by other parties, any third party must be able to independently verify that Z was properly generated. Therefore, the output of RandHound consists of Z and a transcript L, which serves as third-party verifiable proof of Z. The transcript L must (a) enable the third party to replay the protocol execution, and (b) be unforgeable.

L contains all messages sent and received during the protocol execution, as well as the session configuration C. If the verifying party finds C acceptable, specifically the identities of participating servers, he can replay the protocol execution and verify the behavior of the client and the servers, as outlined in Section III-B2. After a successful protocol run completes, the only relevant protocol inputs that remain secret are the private keys of the client and the servers. Therefore, any third party on its own can verify L and decide on its validity since the private keys are only used to produce signatures and the signatures are verified using the public keys.

If an adversary can forge the transcript, producing a valid transcript without an actual run of the protocol, then the adversary must be in possession of the secret keys of all participant listed in C, violating the assumption that at most f nodes are controlled by the adversary.

Therefore, under the assumption that all cryptographic primitives used in RandHound offer their intended security properties, it is infeasible for any party to produce a valid transcript, except by legitimately running the protocol to completion with the willing participation of the at least $\sum_{l=0}^{m-1} |T'_l|$ servers listed in the client's commitment vector T' (step 3).

Further Considerations. In each protocol run, the group element H is derived from the session identifier $\mathrm{H}(C)$, which mitigates replay attacks. A malicious server that tries to replay an old message is immediately detected by the client, as the replayed PVSS proofs will not verify against the new H. It is also crucial for RandHound's security that none of the participants knows a logarithm a with $G = H^a$. Otherwise the participant can prematurely recover secret shares since $(H^{s_i(j)})^a = H^{as_i(j)} = G^{s_i(j)} = S_{ij}$, which violates RandHound's unpredictability property and might even enable a malicious node to bias the output. This has to be taken into account when deriving H from $\mathrm{H}(C)$. The naive way to map $\mathrm{H}(C)$ to a scalar a and then set $H = G^a$ is obviously insecure as $G = H^{1/a}$. The Elligator mappings [7] provide a secure option for elliptic curves.

D. Extensions

Each Lagrange interpolation that the client has to perform to recover a server's secret can be replaced by the evaluation of a hash function as follows: Each server i sends, alongside his encrypted shares, the value $\mathrm{H}(s_i(0))$ as a commitment to the client in step 2. After the client's request to decrypt the shares, each server, whose secret was chosen in T', replies directly with $s_i(0)$. The client checks the received value against the server's commitment and, if valid, integrates it into Z.

Note that the verification of the commitment is necessary, as a malicious server could otherwise just send an arbitrary value as his secret that would be integrated into the collective randomness thereby making it unverifiable against the transcript L. The client can still recover the secret as usual from the decrypted shares with Lagrange interpolation if the above check fails or if the respective server is unavailable.

Finally, SCRAPE [16] provides a new approach to decentralized randomness that builds upon an improved version of PVSS. While this approach is orthogonal to ours, the improved PVSS scheme has a lower verification complexity and can be used to reduce the complexity of RandHound from $O(c^2 n)$ to $O(cn)$, making it more scalable.

IV. RANDHERD: A SCALABLE RANDOMNESS COTHORITY

This section introduces RandHerd, a protocol that builds a collective authority or *cothority* [58] to produce unbiasable and verifiable randomness. RandHerd serves as a decentralized randomness beacon [45], [49], efficiently generating a regular stream of random outputs. RandHerd builds on RandHound, but requires no distinguished client to initiate it, and significantly improves repeat-execution performance.

We first outline RandHerd, then detail the protocol, analyze its security properties, and explore protocol extensions.

A. Overview

RandHerd provides a continually-running decentralized service that can generate publicly verifiable and unbiasable randomness on demand, at regular intervals, or both. RandHerd's goal is to reduce communication and computational overhead of the randomness generation further from RandHound's $O(c^2 n)$ to $O(c^2 \log n)$ given a group size c. To achieve this, RandHerd requires a one-time setup phase that securely shards cothority nodes into subgroups, then leverages aggregation and communication trees to generate subsequent random outputs. As before, the random output $\widehat{r}$ of RandHerd is unbiasable and can be verified, together with the corresponding challenge $\widehat{c}$, as a collective Schnorr signature against RandHerd's collective public key. Fig. 3 illustrates RandHerd's design.

RandHerd's design builds on RandHound, CoSi [58], and a (t, n)-threshold Schnorr signature (TSS) scheme [57] that implements threshold-based witness cosigning (TSS-CoSi).

A cothority configuration C defines a given RandHerd instance, listing the public keys of participating servers and their collective public key X. The RandHerd protocol consists of RandHerd-Setup, which performs one-time setup, and RandHerd-Round, which produces successive random outputs.

The setup protocol uses RandHound to select a RandHerd leader at random and arrange nodes into verifiably unbiased random groups. Each group runs the key generation phase of TSS to establish a public group key $\widehat{X}_l$, such that each group member holds a share of the corresponding private key $\widehat{x}_l$. Each group can issue a collective signature with a cooperation of t_l of nodes. All public group keys contribute to the collective RandHerd public key $\widehat{X}$, which is endorsed by individual servers in a run of CoSi.

Once operational, to produce each random output, RandHerd generates a collective Schnorr signature $(\widehat{c}, \widehat{r})$ on some input w using TSS-CoSi and outputs $\widehat{r}$ as randomness. TSS-CoSi modifies CoSi to use threshold secret sharing (TSS) rather than CoSi's usual exception mechanism to handle node failures, as required to ensure bias-resistance despite node failures. All m RandHerd groups contribute to each output, but each group's contribution requires the participation of only t_l members. Using TSS-CoSi to generate and collectively certify random outputs allows clients to verify any RandHerd output via a simple Schnorr signature check against public key $\widehat{X}$.
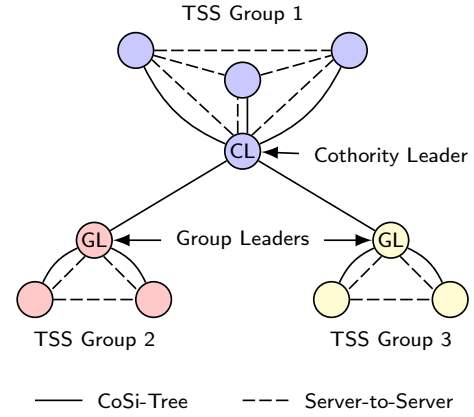

Fig. 3. An overview on the RandHerd design

B. Description

Let $N = \{0, \ldots, n-1\}$ denote the list of all nodes, and let f denote the maximum number of permitted Byzantine nodes. We assume that $n = 3f + 1$. The private and public key of a node $i \in N$ is x_i and $X_i = G^{x_i}$, respectively. Let C denote the cothority configuration file listing the public keys of all nodes, the cothority's collective public key $\widehat{X} = \prod_{j=0}^{n-1} \widehat{X}_j$, contact information such as IP address and port number, default group sizes for secret sharing, and a timestamp on when C was created. Each node has a copy of C.

1) RandHerd-Setup: The setup phase of RandHerd consists of the following four steps:

1) **Leader Election.** When RandHerd starts, each node generates a lottery ticket $t_i = H(C \parallel X_i)$ for every $i \in N$ and sorts them in an ascending order. The ticket t_i with the lowest value wins the lottery and the corresponding node i becomes the tentative RandHerd leader. If this leader is or becomes unavailable, leadership passes to the next node in ascending order. A standard view-change protocol [39], [17] manages the transition between successive leaders. In summary, any server who is dissatisfied with the current leader's progress broadcasts a view-change message for the next leader. Such messages from at least $f + 1$ nodes force a view change, and the new leader begins operation upon receiving at least $2f + 1$ such "votes of confidence." Section IV-E1 discusses an improvement to leader election to make successive leaders unpredictable.

2) **Seed Generation.** The leader assumes the role of the RandHound client and runs the protocol, with all other nodes acting as RandHound servers. Each leader has only one chance to complete this step. If he fails, the next node, as determined by the above lottery, steps in and attempts to execute RandHound. After a successful run of RandHound, the leader obtains the tuple (Z, L), where Z is a collective random string and L is the publicly verifiable transcript that proves the validity of Z. Lastly, the current leader broadcasts (Z, L) to all nodes.

3) **Group Setup.** Once the nodes receive (Z, L), they use L to verify Z, and then use Z as a seed to compute a random permutation of N resulting in N′. Afterwards N′ is sharded into m groups T_l of the same size as in RandHound, for $l \in \{0, \ldots, m-1\}$. The node at index 0 of each group becomes the group leader and the group leader of the first group takes up the role of the temporary RandHerd leader. If any of the leaders is unavailable, the next one, as specified by the order in N′, steps in. After this step, all nodes know their group assignments and the respective group leaders run a TSS-setup to establish the long-term group secret $\widehat{x}_l$ using a secret sharing threshold of $t_l = |T_l|/3 + 1$. All group leaders report back to the current RandHerd leader with the public group key $\widehat{X}_l$.

4) **Key Certification.** As soon as the RandHerd leader has received all $\widehat{X}_j$, he combines them to get the collective RandHerd public key $\widehat{X} = \prod_{j=0}^{m-1} \widehat{X}_j$ and starts a run of the CoSi protocol to certify $\widehat{X}$ by requesting a signature from each individual node. Therefore, the leader sends $\widehat{X}$ together with all $\widehat{X}_j$ and each individual node checks that $\widehat{X}_j$ corresponds to its public group key and that $\widehat{X}$ is well-formed. Only if both checks succeed, the node participates in the co-signing request, otherwise it refuses. The collective signature on $\widehat{X}$ is valid if there are least $f/m+1$ signatures from each group *and* the total number of individual signatures across the groups is at least $2f + 1$. Once a valid signature on $\widehat{X}$ is established, the setup of RandHerd is completed. The validity of $\widehat{X}$ can be verified by anyone by using the collective public key X, as specified in the configuration C.

After a successful setup, RandHerd switches to the operational randomness generation mode. Below we describe how the protocol works with an honest and available leader. A dishonest or failed leader can halt progress at any time, but RandHerd-Round uses a view-change protocol as in RandHerd-Setup to recover from leader failures.

2) RandHerd-Round: In this mode, we distinguish between communications from the RandHerd leader to group leaders, from group leaders to individual nodes, and communications between all nodes within their respective group. Each randomness generation run consists of the following seven steps and can be executed periodically:

1) **Initialization (Leader).** The RandHerd leader initializes a protocol run by broadcasting an announcement message containing a timestamp w to all group leaders. All groups will cooperate to produce a signature $(\widehat{c}, \widehat{r})$ on w.

2) **Group Secret Setup / Commitment (Groups / Servers).** Upon the receipt of the announcement, each group creates a short-term secret $\widehat{v}_l$, using a secret sharing threshold t_l, to produce a group commitment $\widehat{V}_l = G^{\widehat{v}_l}$ that will be used towards a signature of w. Furthermore, each individual node randomly chooses $v_i \in_R \mathbb{Z}_q$, creates a commitment $V_i = G^{v_i}$ that will be used to globally witness, hence validate the round challenge $\widehat{c}$, and sends it to the group leader. The group leader aggregates the received individual commitments into $\widetilde{V}_l = \prod_{i \in T_l} V_i$ and sends $(\widehat{V}_l, \widetilde{V}_l)$ back to the RandHerd leader.

3) **Challenge (Leader).** The RandHerd leader aggregates the respective commitments into $\widehat{V} = \prod_{l=0}^{m-1} \widehat{V}_l$ and $\widetilde{V} = \prod_{l=0}^{m-1} \widetilde{V}_l$, and creates two challenges $\widehat{c} = \mathrm{H}(\widehat{V} \parallel w)$ and $\widetilde{c} = \mathrm{H}(\widetilde{V} \parallel \widehat{V})$. Afterwards, the leader sends $(\widehat{c}, \widetilde{c})$ to all group leaders that in turn re-broadcast them to the individual servers of their group.

4) **Response (Servers).** Server i stores the round group challenge $\widehat{c}$ for later usage, creates its individual response $r_i = v_i - \widetilde{c}x_i$, and sends it back to the group leader. The latter aggregates all responses into $\widetilde{r}_l = \sum_{i \in T_l} r_i$ and creates an exception list $\widetilde{E}_l$ of servers in his group that did not respond or sent bad responses. Finally, each group leader sends $(\widetilde{r}_l, \widetilde{E}_l)$ to the RandHerd leader.

5) **Secret Recovery Request (Leader).** The RandHerd leader gathers all exceptions $\widetilde{E}_l$ into a list $\widetilde{E}$, and aggregates the responses into $\widetilde{r} = \sum_{l=0}^{m-1} \widetilde{r}_l$ taking $\widetilde{E}$ into account. If at least $2f + 1$ servers contributed to $\widetilde{r}$, the RandHerd leader sends the global group commitment $\widehat{V}$ and the signature $(\widetilde{c}, \widetilde{r}, \widetilde{E})$ to all group leaders thereby requesting the recovery of the group secrets.

6) **Group Secret Recovery (Groups / Servers).** The group leaders re-broadcast the received message. Each group member individually checks that $(\widetilde{c}, \widetilde{r}, \widetilde{E})$ is a valid signature on $\widehat{V}$ and only if it is the case and at least $2f+1$ individual servers signed off, they start reconstructing the short-term secret $\widehat{v}_l$. The group leader creates the group response $\widehat{r}_l = \widehat{v}_l - \widehat{c}\widehat{x}_l$ and sends it to the RandHerd leader.

7) **Randomness Recovery (Leader).** The RandHerd leader aggregates all responses $\widehat{r} = \sum_{l=0}^{m-1} \widehat{r}_l$ and, only if he received a reply from all groups, he releases $(\widehat{c}, \widehat{r})$ as the collective randomness of RandHerd.

3) Randomness Verification: The collective randomness $(\widehat{c}, \widehat{r})$ of RandHerd is a collective Schnorr signature on the timestamp w, which is efficiently verifiable against the aggregate group key $\widehat{X}$.

C. Security Properties

RandHerd provides the following security properties:
1) **Availability.** Given an honest leader, the protocol successfully completes and produces the final random output Z with high probability.
2) **Unpredictability.** No party learns anything about the final random output Z, except with negligible probability, until the group responses are revealed.
3) **Unbiasability.** The final random output Z represents an unbiased, uniformly random value, except with negligible probability.
4) **Verifiability.** The collective randomness Z is third-party verifiable as a collective Schnorr signature under $\widehat{X}$.

We make the same assumptions as in the case of RandHound (Section III-C) on the behavior of the honest nodes and the cryptographic primitives RandHerd employs.

RandHerd uses a simple and predictable ahead-of-time election mechanism to choose the temporary RandHerd leader in the setup phase. This approach is sufficient because the group assignments and the RandHerd leader for the randomness phase of the protocol are chosen based on the output of RandHound. RandHound's properties of unbiasability and unpredictability hold for honest and dishonest clients. Therefore, the resulting group setup has the same properties in both cases.

Availability. Our goal is to ensure that with high probability the protocol successfully completes, even in the presence of an active adversary.

As discussed above, the use of RandHound in the setup phase ensures that all groups are randomly assigned. If the RandHerd leader makes satisfactory progress, the secret sharing threshold $t_l = f/m+1$ enables $2f/m$ honest nodes in each group to reconstruct the short-term secret $\widehat{v}_l$, hence produce the group response $\widehat{r}_l$ without requiring the collaboration of malicious nodes. An honest leader will make satisfactory progress and eventually output $\widehat{r}$ at the end of step 7. This setup corresponds to a run of RandHound by an honest client. Therefore, the analysis of the failure probability of a RandHound run described in Section V-C is applicable to RandHerd in the honest leader scenario.

In RandHerd, however, with a probability f/n, a dishonest client will be selected as the RandHerd leader. Although the choice of a dishonest leader does not affect the group assignments, he might arbitrarily decide to stop making progress at any point of the protocol. We need to ensure RandHerd's availability over time, and if the current leader stops making adequate progress, we move to the next leader indicated by the random output of RandHound and, as with common BFT protocols, we rely on "view change" [17], [39] to continue operations.

Unpredictability. We want to ensure that the random output of RandHerd remains unknown until the group responses $\widehat{r}_l$ are revealed in step 6.

The high-level design of RandHerd closely resembles that of RandHound. Both protocols use the same thresholds, assign n nodes into m groups, and each group contributes an exactly one secret towards the final random output of the protocol. Therefore, as in RandHound, there will similarly be at least one RandHerd group with at least an average number of honest nodes. Furthermore, the secret-sharing and required group inputs threshold of $t_l = f + 1$ guarantees that for at least one group, the adversary cannot prematurely recover $\widehat{v}_l$ and reconstruct the group's response $\widehat{r}_l$. Therefore, before step 6, the adversary will control at most $m - 1$ groups and obtain at most $m - 1$ out of m responses that contribute to $\widehat{r}$.

Unbiasability. Our goal is to prevent the adversary from biasing the value of the random output $\widehat{r}$.

As in RandHound, we know that for at least one group the adversary cannot prematurely recover $\widehat{r}_l$ and that $\widehat{r}_l$ contains a contribution from at least one honest group member. Further, the requirement that the leader must obtain a sign-off from $2f + 1$ individual nodes in step 4 on his commitment $\widehat{V}$, fixes

the output value $\widehat{r}$ before any group secrets $\widehat{r}_l$ are produced. This effectively commits the leader to a single output $\widehat{r}$.

The main difference between RandHound and RandHerd is the fact that an adversary who controls the leader can affect unbiasability by withholding the protocol output $\widehat{r}$ in step 7, if $\widehat{r}$ is not beneficial to him. A failure of a leader would force a view change and therefore a new run of RandHerd, giving the adversary at least one alternative value of $\widehat{r}$, if the next selected leader is honest, or several tries if multiple successive leaders are dishonest or the adversary can successfully DoS them. The adversary cannot freely choose the next value of $\widehat{r}$, nor go back to the previous value if the next one is not preferable, the fact that he can sacrifice a leadership role to try for an alternate outcome constitutes bias. This bias is limited, as the view-change schedule must eventually appoint an honest leader, at which point the adversary has no further bias opportunity. Section IV-D further addresses this issue with an improvement ensuring that an adversary can hope to hold leadership for at most $O(\log n)$ such events before permanently losing leadership and hence bias opportunity.

Verifiability. The random output $\widehat{r}$ generated in RandHerd is obtained from a TSS-CoSi Schnorr signature $(\widehat{c}, \widehat{r})$ on input w against a public key $\widehat{X}$. Any third-party can verify $\widehat{r}$ by simply checking the validity of $(\widehat{c}, \widehat{r})$ as a standard Schnorr signature on input w using $\widehat{X}$.

D. Addressing Leader Availability Issues

Each run of RandHerd is coordinated by a RandHerd leader who is responsible for ensuring a satisfactory progress of the protocol. Although a (honest or dishonest) leader might fail and cause the protocol failure, we are specifically concerned with intentional failures that benefit the adversary and enable him to affect the protocol's output.

As discussed above, once a dishonest RandHerd leader receives responses from group leaders in step 7, he is the first one to know $\widehat{r}$ and can act accordingly, including failing the protocol. However, the failure of the RandHerd leader does not necessarily have to cause the failure of the protocol. Even without the dishonest leader's participation, $f/m+1$ of honest nodes in each group are capable of recovering the protocol output. They need, however, a consistent view of the protocol and the output value that was committed to.

Instead of requiring a CoSi round to get $2f+1$ signatures on $\widehat{V}$, we use a Byzantine Fault Tolerance (BFT) protocol to reach consensus on $\widehat{V}$ and consequently on the global challenge $\widehat{c} = H(\widehat{V} \parallel w)$. Upon a successful completion of BFT, at least $f + 1$ honest nodes have witnessed that we have consensus on the $\widehat{V}$. Consequently, the $\widehat{c}$ that is required to produce each group's response $\widehat{r}_l = \widehat{v}_l - \widehat{c}\widehat{x}_l$ is "set in stone" at this point. If a leader fails, instead of restarting RandHerd, we can select a new leader, whose only allowed action is to continue the protocol from the existing commitment. This design removes the opportunity for a dishonest leader biasing the output even a few times before losing leadership.

Using a traditional BFT protocol (*e.g.*, PBFT [17]) would yield poor scalability for RandHerd because of the large num-

ber of servers that participate in the protcol. To overcome this challenge, we use BFT-CoSi from ByzCoin [39], a Byzantine consensus protocol that uses scalable collective signing, to agree on successfully delivering the commitment $\widehat{V}$. Due to the BFT guarantees RandHerd crosses the point-of-no return when consensus is reached. Even if the dishonest leader, tries to bias output by failing the protocol, the new (eventually honest) leader will be able to recover $\widehat{r}$, allowing all honest servers to successfully complete the protocol.

The downside of this BFT-commitment approach is that once consensus is reached and the point-of-no return is crossed, then in the rare event that an adversary controls two-thirds of any group, the attacker can halt the protocol forever by preventing honest nodes from recovering the committed secret. This risk may necessitate a more conservative choice of group size, such that the chance of an adversary ever controlling any group is not merely unlikely but truly negligible.

E. Extensions

1) Randomizing Temporary-Leader Election: The current set-up phase of RandHerd uses a simple leader election mechanism. Because the ticket generation uses only values known to all nodes, it is efficient as it does not require any communication between the nodes but makes the outcome of the election predicable as soon as the cothority configuration file C is available. We use this mechanism to elect a temporary RandHerd leader whose only responsibility is to run and provide the output of RandHound to other servers. Rand-Hound's unbiasibility property prevents the dishonest leader from biasing its output. However, an adversary can force f restarts of RandHound and can therefore delay the setup by compromising the first (or next) f successive leaders in a well-known schedule.

To address this issue, we can use a lottery mechanism that depends on verifiable random functions (VRFs) [43], which ensures that each participant obtains an unpredictable "fair-share" chance of getting to be the leader in each round. Each node produces its lottery ticket as $t_i = \mathrm{H}(C \parallel j)^{x_i}$, where C is the group configuration, j is a round number, and x_i is node i's secret key, along with a NIZK consistency proof showing that t_i is well-formed. Since an adversary has at least a constant and unpredictable chance of losing the leadership to some honest node in each lottery, this refinement ensures with high probability that an adversary can induce at most $O(\log n)$ successive view changes before losing leadership.

2) BLS Signatures: Through the use of CoSi and TSS, RandHerd utilizes collective Schnorr signatures in a threshold setting. Other alternatives are possible. Specifically, Boneh-Lynn-Shacham (BLS) [12] signatures require pairing-based curves, but offer even shorter signatures (a single elliptic curve point) and a simpler signing protocol. In the simplified design using BLS signatures, there is no need to form a fresh Schnorr commitment collectively, and the process does not need to be coordinated by a group leader. Instead, a member of each subgroup, whenever it has decided that the next round has arrived, produces and releases its share for a BLS signature

of the message for the appropriate time (based on a hash of view information and the wall-clock time or sequence number). Each member of a given subgroup waits until a threshold number of BLS signature shares are available for that subgroup, and then forms the BLS signature for this subgroup. The first member to do so can then simply announce or gossip it with members of other subgroups, combining subgroup signatures until a global BLS signature is available (based on a simple combination of the signatures of all subgroups). This activity can be unstructured and leaderless, since no "arbitrary choices" need to be made per-transaction: the output of each time-step is completely deterministic but cryptographically random and unpredictable before the designated time.

V. Evaluation

This section experimentally evaluates of our prototype implementations of RandHound and RandHerd. The primary questions we wish to evaluate are whether architectures of the two protocols are practical and scalable to large numbers, *e.g.*, hundreds or thousands of servers, in realistic scenarios. Important secondary questions are what the important costs are, such as randomness generation latencies and computation costs. We start with some details on the implementation itself, followed by our experimental results, and finally describe our analysis of the failure probability for both protocols.

A. Implementation

We implemented PVSS, TSS, RandHound, and RandHerd in Go [33] and made these implementations available on GitHub as part of the EPFL DEDIS lab's Cothority framework.[2] We reused existing cothority framework code for CoSi and network communication, and built on the DEDIS advanced crypto library[3] for cryptographic operations such as Shamir secret sharing, zero-knowledge proofs, and optimized arithmetic on the popular Curve25519 elliptic curve [5]. As a rough indicator of implementation complexity, Table I shows approximate lines of code (LoC) of the new modules. Line counts were measured with GoLoC.[4]

TABLE I
LINES OF CODE PER MODULE

PVSS	TSS	RandHound	RandHerd
300	700	1300	1000

B. Performance Measurements

1) Experimental Setup: We ran all our experiments on DeterLab[5] using 32 physical machines, each equipped with an Intel Xeon E5-2650 v4 (24 cores at $2.2\,\mathrm{GHz}$), $64\,\mathrm{GBytes}$

---

[2]https://github.com/dedis/cothority
[3]https://github.com/dedis/crypto
[4]https://github.com/gengo/goloc
[5]http://isi.deterlab.net/

of RAM, and a 10 Gbps network link. To simulate a globally-distributed deployment realistically, we restricted the bandwidth of all intern-node connections to 100 Mbps and imposed 200 ms round-trip latencies on all communication links.

To scale our experiments up to 1024 participants given limited physical resources, we oversubscribed the DeterLab servers by up to a factor of 32, arranging the nodes such that most messages had to go through the network. To test the influence of oversubscription on our experiments, we reran the same simulations with 16 servers only. This resulted in an overhead increase of about 20%, indicating that our experiments are already CPU-bound and not network-bound at this scale. We therefore consider these simulation results to be pessimistic: real-world deployments on servers that are not oversubscribed in this way may yield better performance.

2) RandHound: Fig. 4 shows the CPU-usage costs of a complete RandHound run that generates a random value from N servers. We measured the total costs across all servers, plus the costs of the client that coordinates RandHound and generates the Transcript. With 1024 nodes divided into groups of 32 nodes, for example, the complete RandHound run to generate randomness requires less than 10 CPU minutes total, correspond to a cost of about $0.02 on Amazon EC2. This cost breaks down to about 0.3 CPU seconds per server, representing negligible per-transaction costs to the servers. The client that initiates RandHound spends about 3 CPU minutes, costing less than $0.01 on Amazon EC2. These results suggest that RandHound is quite economical on today's hardware.

Fig. 5. Total wall clock time of a RandHound protocol run

3) RandHerd: The RandHerd protocol requires a setup phase, which uses RandHound to form random groups and CoSi to sign the RandHerd collective key. The measured CPU usage of RandHerd setup is depicted in Fig. 6. For 1024 nodes and a group size of 32, RandHerd setup requires roughly 40 CPU-hours total (2.3 CPU-minutes per node), corresponding to a cost of $4.00 total on Amazon EC2 (0.3 cents per participant). The associated wall clock time we measured, not depicted in the graphs, amounts to about 10 minutes.

Fig. 4. Overall CPU cost of a RandHound protocol run

Fig. 6. Total CPU usage of RandHerd setup

Fig. 5 shows the wall clock time of a complete RandHound run for different configurations. This test measures total time elapsed from when the client initiates RandHound until the client has computed and verified the random output. Our measurements show that the wall clock time used by the servers to process client messages is negligible in comparison, and hence not depicted in Fig. 5. In the 1024-node configuration with groups of 32 nodes, randomness generation and verification take roughly 290 and 160 seconds, respectively.
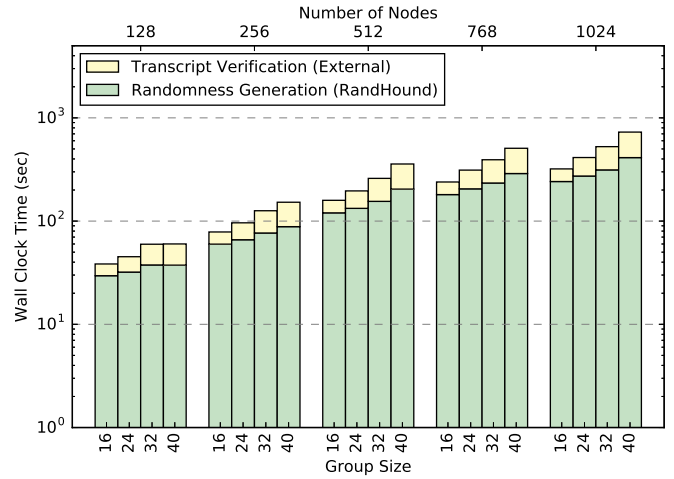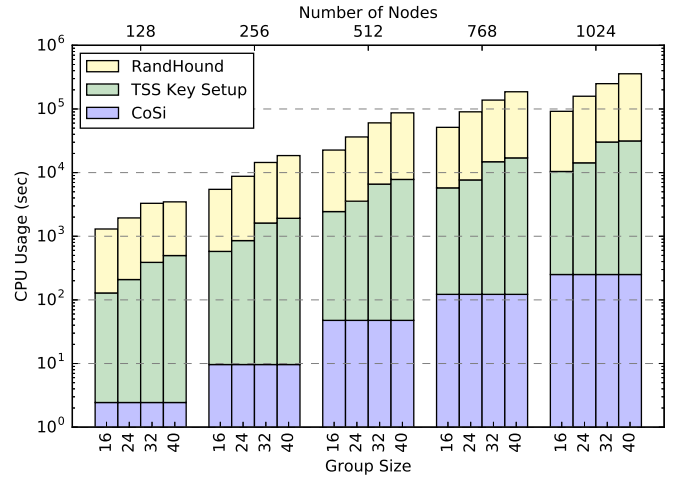
After this setup, RandHerd produces random numbers much more efficiently. Fig. 7 illustrates measured wall clock time for a single RandHerd round to generate a 32-byte random value. With 1024 nodes in groups of 32, RandHerd takes about 6 seconds per round. The corresponding CPU usage across the entire system, not shown in the graphs, amounts to roughly 30 seconds total (or about 29 CPU-milliseconds per node).

A clear sign of the server-oversubscription with regard to the network-traffic can be seen in Fig. 7, where the wall clock

time for 1024 nodes and a group size of 32 is lower than the one for a group size of 24. This is due to the fact that nodes running on the same server do not have any network-delay. We did a verification run without server oversubscription for up to 512 nodes and could verify that the wall clock time increases with higher group-size.
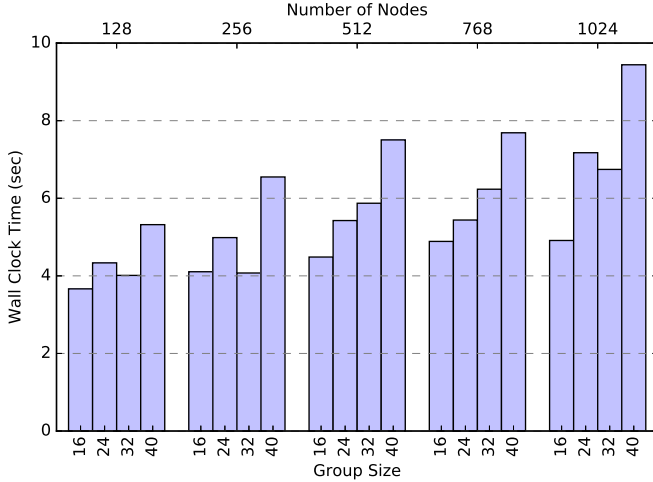

Fig. 7. Wall clock time per randomness creation round in RandHerd

Fig. 8 compares communication bandwidth costs for CoSi, RandHound, and RandHerd, with varying number of participants and a fixed group size of 32 nodes. The straight lines depict total costs, while the dashed lines depict average cost per participating server. For the case of 1024 nodes, CoSi and RandHound require about 15 and 25 MB, respectively. After the initial setup, one round of RandHerd among 1024 nodes requires about 400 MB (excluding any setup costs) due to the higher in-group communication. These values correspond to the sum of the communication costs of the entire system and, considering the number of servers involved, are still fairly moderate. This can be also seen as the average per server cost is less than 300 KB for RandHerd and around 20 KB for CoSi and RandHound.

Finally, Fig. 9 compares RandHerd, configured to use only one group, against a non-scalable baseline protocol similar to RandShare. Because RandShare performs verifiable secret sharing among all n nodes, it has computation and communication complexity of $O(n^3)$. In comparison, RandHerd has sublinear per-round complexity of $O(\log n)$ when the group size is constant.

C. Availability Failure Analysis

An adversary who controls too many nodes in any group can compromise the availability of both RandHound and RandHerd. We can analyze the probability of availability failure assuming that nodes are assigned randomly to groups, which is the case in RandHound when the client assigns groups honestly, and is always the case in RandHerd. As discussed in Section III-C, dishonest grouping in RandHound amounts to self-DoS by the client and is thus outside the threat model.
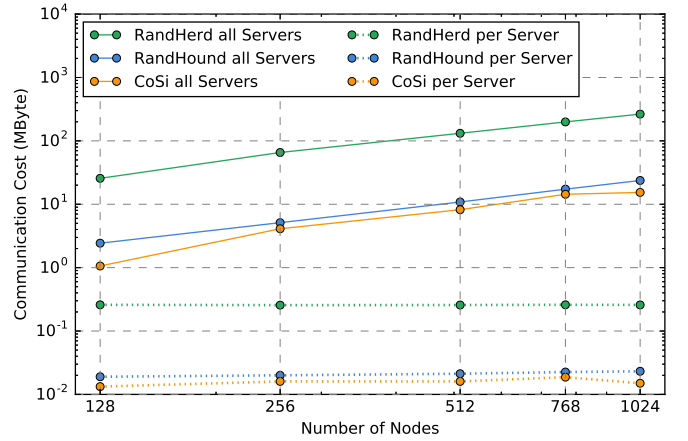

Fig. 8. Comparison of communication bandwidth costs between RandHerd, RandHound, and CoSi for fixed group size $c = 32$
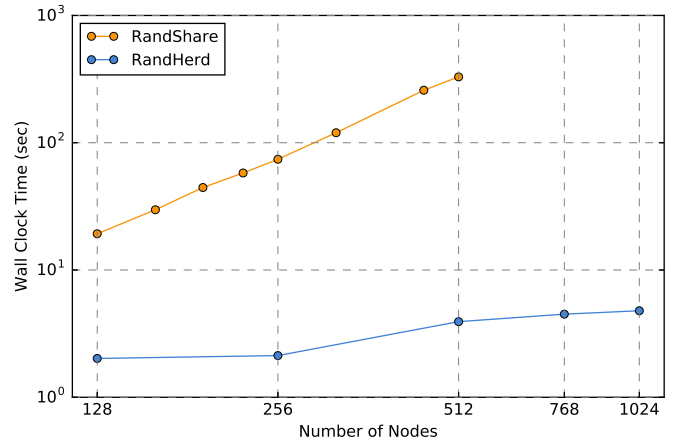

Fig. 9. Comparison of randomness generation times for RandShare and RandHerd (group size $c = 32$ for RandHerd and $c = n$ for RandShare)

To get an upper bound for the failure probability of the entire system, we first bound the failure probability of a single group, that can be modeled as a random variable X that follows the hypergeometric distribution, followed by the application of Boole's inequality, also known as the union bound. For a single group we start with Chvátal's formula [55]

$$P[X \geq E[X] + cd] \leq e^{-2cd^2}$$

where $d \geq 0$ is a constant and c is the number of draws or in our case the group size. The event of having a disproportionate number of malicious nodes in a given group is modeled by $X \geq c - t + 1$, where t is the secret sharing threshold. In our case we use $t = cp + 1$ since $E[X] = cp$, where $p \leq 0.33$ is the adversaries' power. Plugging everything into Chvátal's formula and doing some simplifications, we obtain

$$P[X \geq c(1-p)] \leq e^{-2c(1-2p)^2}$$

Applying the union bound on this result, we obtain Figs. 10 and 11, which show average system failure probabilities q for
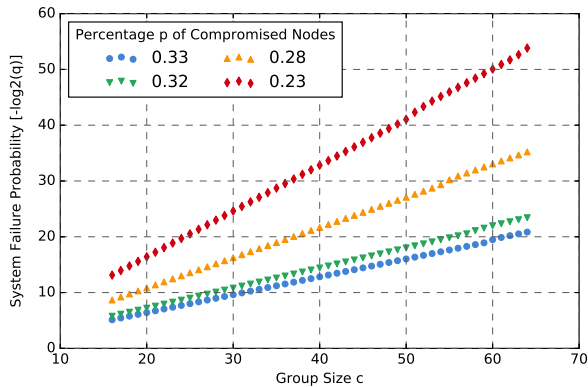
Fig. 10. System failure probability for varying group sizes
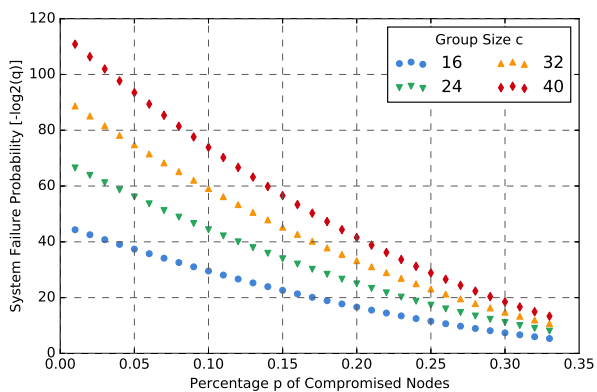

Fig. 11. System failure probability for varying adversarial power

varying group sizes ($c = 16, \ldots, 64$) and varying adversarial power ($p = 0.01, \ldots, 0.33$), respectively. Note that q on the y-axis is plotted in "security parameter" form as $-\log_2(q)$: thus, higher points in the graph indicate exponentially lower failure probability. Finally, Table II lists failure probabilities for some concrete configurations. There we see, for example, that both RandHound and RandHerd have a failure probability of at most $2^{-10.25} \approx 0.08\%$ for $p = 0.33$ and $c = 32$. Moreover, assuming $p = 0.33$, we identified the point where the system's failure probability falls below 1% for a group size of $c = 21$.

TABLE II
SYSTEM FAILURE PROBABILITIES q (GIVEN AS $-\log_2(q)$) FOR CONCRETE
CONFIGURATIONS OF ADVERSARIAL POWER p AND GROUP SIZE c

$p \mid c$	16	24	32	40
0.23	13.13	19.69	26.26	32.82
0.28	8.66	15.17	17.33	21.67
0.32	5.76	8.64	11.52	14.40
0.33	5.12	7.69	10.25	12.82

VI. RELATED WORK

Generation of public randomness has been studied in various contexts. In 1981, Blum proposed the first coin flipping proto-

col [10]. Rabin introduced the notion of cryptographic randomness beacons in 1983 [49]. NIST later launched such a beacon to generate randomness from high-entropy sources [45]. Centralized randomness servers have seen limited adoption, however, in part because users must rely on the trustworthiness of the party that runs the service.

Other approaches attempt to avoid trusted parties [48], [13], [2], [21]. Bonneau et al. [13] use Bitcoin to collect entropy, focusing on analyzing the financial cost of a given amount of bias rather than preventing bias outright. Lenstra et al. [40] propose a new cryptographic primitive, a *slow hash*, to prevent a client from biasing the output. This approach is promising but relies on new and untested cryptographic hardness assumptions, and assumes that everyone observes the commitment before the slow hash produces its output. If an adversary can delay the commitment messages and/or accelerate the slow hash sufficiently, he can see the hash function's output before committing, leaving the difficult question of how slow is "slow enough" in practice. Other approached use lotteries [2], or financial data [21] as public randomness sources.

An important observation by Gennaro et al. [29] is that in many distributed key generation protocols [47] an attacker can observe public values of honest participants. To mitigate this attack, the authors propose to delay the disclosure of the protocol's public values after a "point-of-no-return" at which point the attacker cannot influence the output anymore. We also use the concept of a "point-of-no-return" to prevent an adversary from biasing the output. However, their assumption of a fully synchronous network is unrealistic for real-world scenarios. Cachin et al., propose an asynchronous distributed coin tossing scheme for public randomness generation [15], which relies on a trusted setup dealer.

We improve on that by letting multiple nodes deal secrets and combine them for randomness generation in our protocols. Finally, Kate et al. [38], introduced an approach to solve distributed key-generation in large-scale asynchronous networks, such as the Internet. The communication complexity of their solution, similar to Gennaro's and Cachin's prevents scalability to large numbers of nodes. Our protocols use sharding to limit communication overheads to linear increases, which enables RandHound and RandHerd to scale to hundreds of nodes.

Applications of public randomness are manifold and include the protection of hidden services in the Tor network [34], selection of elliptic curve parameters [2], [40], Byzantine consensus [46], electronic voting [1], random sharding of nodes into groups [35], and non-interactive client-puzzles [37]. In all of these cases, both RandHound and RandHerd may be useful for generating bias-resistant, third-party verifiable randomness. For example, RandHound could be integrated into the Tor consensus mechanism to help the directory authorities generate their daily random values in order to protect hidden services against DoS or popularity estimation attacks.

VII. CONCLUSIONS

Although many distributed protocols critically depend on public bias-resistant randomness for security, current solutions that are secure against active adversaries only work for small ($n \approx 10$) numbers of participants [15], [38]. In this paper, we have focused on the important issue of scalability and addressed this challenge by adapting well-known cryptographic primitives. We have proposed two different approaches to generating public randomness in a secure manner in the presence of a Byzantine adversary. RandHound uses PVSS and depends on the pigeonhole principle for output integrity. RandHerd relies on RandHound for secure setup and then uses TSS and CoSi to produce random output as a Schnorr signature verifiable under a collective RandHerd key. RandHound and RandHerd provide *unbiasability*, *unpredictability*, *availability* and *third-party verifiability* while retaining good performance and low failure probabilities. Our working prototype demonstrates that both protocols, in principle, can scale even to thousands of participants. By carefully choosing protocols parameters, however, we achieve a balance of performance, security, and availability. While retaining a failure probability of at most 0.08% against a Byzantine adversary, a set of 512 nodes divided into groups of 32 can produce fresh random output every 240 seconds in RandHound, and every 6 seconds in RandHerd after an initial setup.

ACKNOWLEDGMENTS

REFERENCES

[1] B. Adida. Helios: Web-based Open-audit Voting. In *17th USENIX Security Symposium*, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[2] T. Baignères, C. Delerablée, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain. Trap Me If You Can – Million Dollar Curve. Cryptology ePrint Archive, Report 2015/1249, 2015.

[3] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[4] I. Bentov, A. Gabizon, and D. Zuckerman. Bitcoin Beacon. https://arxiv.org/abs/1605.04559, 2016.

[5] D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, T. Lange, R. Niederhagen, and C. van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive, Report 2014/571, 2014.

[7] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 967–980. ACM, 2013.

[8] D. J. Bernstein, T. Lange, and R. Niederhagen. Dual EC: A Standardized Back Door. Cryptology ePrint Archive, Report 2015/767, 2015.

[9] G. R. Blakley. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on*, 00:313, 1979.

[10] M. Blum. Coin Flipping by Telephone: A Protocol for Solving Impossible Problems. In *Advances in Cryptology (CRYPTO)*, 1981.

[11] C. Blundo, A. De Santis, and U. Vaccaro. Randomness in distribution protocols. In S. Abiteboul and E. Shamir, editors, *Automata, Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 568–579. Springer Berlin Heidelberg, 1994.

[12] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *ASIACRYPT*, Dec. 2001.

[13] J. Bonneau, J. Clark, and S. Goldfeder. On Bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015.

[14] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology (CRYPTO)*, Aug. 2001.

[15] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18:219–246, July 2005.

[16] I. Cascudo and B. David. SCRAPE: Scalable randomness attested by public entities. Cryptology ePrint Archive, Report 2017/216, 2017. https://eprint.iacr.org/2017/216.pdf.

[17] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[18] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *IACR International Cryptology Conference (CRYPTO)*, 1992.

[19] R. Chirgwin. iOS 7's weak random number generator stuns kernel security. *The Register*, Mar. 2014.

[20] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Symposium on Foundations of Computer Science (SFCS)*, SFCS '85, pages 383–395, Washington, DC, USA, 1985. IEEE Computer Society.

[21] J. Clark and U. Hengartner. On the Use of Financial Data as a Random Beacon. Cryptology ePrint Archive, Report 2010/361, 2010.

[22] H. Corrigan-Gibbs, W. Mu, D. Boneh, and B. Ford. Ensuring high-quality randomness in cryptographic key generation. In *20th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013.

[23] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, and E. Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[24] Y. G. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology (CRYPTO)*, 1989.

[25] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *13th USENIX Security Symposium*, Aug. 2004.

[26] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 427–438, Washington, DC, USA, 1987. IEEE Computer Society.

[27] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *IACR International Cryptology Conference (CRYPTO)*, pages 186–194, 1987.

[28] M. Franklin and H. Zhang. Unique ring signatures: A practical construction. In A.-R. Sadeghi, editor, *Financial Cryptography and Data Security 2013*, pages 162–170. Springer Berlin Heidelberg, 2013.

[29] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[30] M. Ghosh, M. Richardson, B. Ford, and R. Jansen. A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays. In *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2014.

[31] S. Gibbs. Man hacked random-number generator to rig lotteries, investigators say. *The Guardian*, Apr. 2016.

[32] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report 2003-1890, Cornell University, February 2003.

[33] The Go programming language, Jan. 2015. http://golang.org/.

[34] D. Goulet and G. Kadianakis. Random Number Generation During Tor Voting, 2015.

[35] R. Guerraoui, F. Huc, and A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 176–183, New York, NY, USA, 2013. ACM.

[36] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy*, pages 371–385, 2006.

[37] J. A. Halderman and B. Waters. Harvesting Verifiable Challenges from Oblivious Online Sources. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 330–341, New York, NY, USA, 2007. ACM.

[38] A. Kate and I. Goldberg. Distributed key generation for the internet. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 119–128. IEEE, 2009.

[39] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Conference on Security Symposium*, 2016.

[40] A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366, 2015.

[41] C. Lesniewski-Lass and M. F. Kaashoek. Whanau: A sybil-proof distributed hash table. NSDI, 2010.

[42] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures. In *ACM Conference on Computer and Communications Security (CCS)*, 2001.

[43] S. Micali, S. Vadhan, and M. Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–130. IEEE Computer Society, 1999.

[44] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Oct. 2008.

[45] NIST Randomness Beacon.

[46] O. Oluwasanmi and J. Saia. Scalable Byzantine Agreement with a Random Beacon. In A. W. Richa and C. Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 7596 of *Lecture Notes in Computer Science*, pages 253–265. Springer Berlin Heidelberg, 2012.

[47] T. P. Pedersen. A threshold cryptosystem without a trusted party. In *EUROCRYPT (EUROCRYPT)*. Springer, 1991.

[48] S. Popov. On a Decentralized Trustless Pseudo-Random Number Generation Algorithm. Cryptology ePrint Archive, Report 2016/228, 2016.

[49] M. O. Rabin. Transaction Protection by Beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[50] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *ACM Symposium on Theory of Computing (STOC)*, 1989.

[51] C.-P. Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology (CRYPTO)*, 1990.

[52] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *IACR International Cryptology Conference (CRYPTO)*, pages 784–784, 1999.

[53] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[54] D. Shumow and N. Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual EC PRNG. CRYPTO 2007 Rump Session, 2007.

[55] M. Skala. Hypergeometric Tail Inequalities: Ending the Insanity. *CoRR*, abs/1311.5939, 2013.

[56] M. Stadler. Publicly Verifiable Secret Sharing. In *15th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 190–199, Berlin, Heidelberg, 1996. Springer.

[57] D. R. Stinson and R. Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In V. Varadharajan and Y. Mu, editors, *Australasian Conference on Information Security and Privacy (ACISP)*, pages 417–434, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[58] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy*, May 2016.

[59] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152, New York, NY, USA, 2015. ACM.

[60] D. I. Wolinsky, H. Corrigan-Gibbs, A. Johnson, and B. Ford. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.