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, Hartford, USA
†École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
‡Yale University, New Haven, USA

***Abstract*—Bias-resistant public randomness is a critical component required in many (distributed) protocols. Generating public randomness is hard, however, because active adversaries behave dishonestly in order 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. In response, we propose two large-scale, distributed protocols, RandHound and RandHerd, which provide publicly-verifiable, unpredictable, and unbiasable randomness against Byzantine adversaries. RandHound relies on an untrusted client to divide a set of randomness servers into groups for scalability, and it depends on the pigeonhole 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 are then repeatedly used to 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, whereas RandHerd, after a setup phase of 260 seconds, is able to generate fresh random output in intervals of approximately 6 seconds. For this configuration, our analysis indicates that 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 a high-entropy output, is a critical component in many protocols [10], [19]. 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.

More concretely, Tor hidden services [21] depend on the generation of a fresh random value each day for protection against popularity estimations and DoS attacks [28]. Anytrust-based systems, such as Herbivore [27], Dissent [51], and Vuvuzela [50], as well as sharded blockchains [20], use bias-resistant public randomness for scalability by sharding participants into smaller groups. TorPath [26] 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], [33], where adversaries should not be able to stir 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 [39], [14] for achieving scalability.

The process of generating public randomness is nontrivial though, because obtaining access to sources of good randomness (even only in terms of entropy) is difficult for regular users. One approach to public randomness is randomness beacons which were introduced by Rabin [42] in the context of contract signing, where a trusted third party regularly emits randomly chosen integers to the public. The NIST beacon [38] provides hardware-generated random output from quantum-mechanical effects, but assumes everyone trusts their centralized beacon—a problematic assumption, especially after the Dual EC DRBG debacle [46], [8].

In this work, we are concerned with the important complementary challenge of producing good sources of randomness in terms of trust. Having an approach to public randomness without a trusted party is very attractive, especially in a collaborative setting, where a significant number of users wishes to participate. This raises many interesting questions on making use of randomness beacons in a distributed setting, such as how to choose a subset of beacons or how to combine random outputs of the chosen subset in an unbiasable way in the presence of an active adversary. Proposals discussing approaches without trusted parties [41] make use of Bitcoin [12], [4], slow cryptographic hash functions [33], lotteries [2], or financial data [18] 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 [40] and Byzantine consensus protocols [14]. 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 $n \approx 10$ [14], [31], we address scalability challenges of using large values of $n \approx 100$ for enhanced security in real-world scenarios. For example, this is relevant for public cryptocurrencies [37], [32] 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, and are built upon existing cryptographic primitives to produce distributed randomness protocols that are efficient and scalable in practice. We introduce two scalable public-randomness generation protocols, RandHound and RandHerd, which provide *unbiasability*, *unpredictability*, *availability*, and *third-party verifiability* of random output.

RandHound is a client-server randomness scavenging protocol that enables a client to gather randomness from a potentially large set of servers. In every run, the client first splits the servers into balanced subgroups to achieve scalability, and each group uses publicly verifiable secret sharing (PVSS) [44] 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 account for potential server failures, the client subsequently selects only a subset of secret inputs from each group, relying on the pidgeonhole principle to ensure the integrity of RandHound's final output. The client then commits to his choice by requesting from all servers a CoSi signature [49] to ensure protection against equivocation attacks. After the servers release the selected secrets, the client combines and publishes them as RandHound's collective random output along with a third-party verifiable transcript of the protocol run that documents all relevant choices made within the protocol.

RandHerd is a scalable randomness cothority (collective authority) that uses RandHound and CoSi in a one-time setup to split a set of servers securely and verifiably into uniformly random groups, and to generate and certify an aggregate public key which is used to produce and verify the random output of RandHerd. In the operational mode, to enable the generation of collective randomness at frequent intervals using input from all groups, RandHerd uses CoSi in conjunction with a threshold variant of CoSi (TSS-CoSi) that we propose and outline in this work as well. Each collective random output produced by RandHerd is also a collective Schnorr signature [48], [49], that can be validated efficiently against the previously created aggregate group key.

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, after a one-time setup process using RandHound taking approximately 260 seconds. The randomness verification overhead of RandHerd is far below one second. In the same configuration setup, when we independently use RandHound to produce a 32-byte random output, it takes approximately 240 seconds to produce randomness and approximately 76 seconds to verify it using the produced 4 MBytes transcript. Our extensive analysis on failure probabilities also suggests that a Byzantine adversary can threaten availability, in either case, with a probability of at most 0.08%.

The remainder of the paper is organized as follows. In Section II we explore background and motivation for public randomness generation. In Sections III and IV we discuss the design and security properties of RandHound and RandHerd, respectively. In Section V we describe evaluation of the prototype implementations of both protocols. Finally, in Section VI we summarize related work and in Section VII we conclude the paper.

II. Background and Motivation

We first introduce notation and then recall different techniques for secret sharing and Schnorr signing, which we use as building blocks for RandHound and RandHerd. Then, we 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 $X_i = G^{x_i}$.

A. Publicly Verifiable Secret-Sharing

A (t, n)-secret sharing scheme [9], [45] 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 does not learn anything about s. Verifiable secret-sharing (VSS) [17], [22] extends secret sharing to account for a dishonest dealer who might intentionally send out bad shares and prevent honest trustees from recovering the same, correct secret.

A publicly verifiable secret sharing (PVSS) [44] 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) [16], [23], [24] 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 and 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 $i \in \{1, \dots, n\}$, 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 $\widehat{\pi}_i$. Afterwards, he publishes $\widehat{S}_i$, $\widehat{\pi}_i$, and A_j.
2) Each trustee i verifies his share $\widehat{S}_i$ using $\widehat{\pi}_i$ 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 π_i.
3) The dealer checks the validity of S_i against π_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

Both RandHound and RandHerd use, as important building blocks, different variations of the well-known Schnorr signatures [43] and multisignatures [35], [3].

1) Threshold Signing: TSS [48] is a distributed (t, n)-threshold Schnorr signature scheme, that 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 $c = \mathrm{H}(V \parallel S)$. 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 [49] 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 $c = \mathrm{H}(\widehat{V} \parallel S)$, each server i responds with a partial response $r_i = v_i - cx_i$, and all responses are aggregated into $r = \sum_{i=0}^{n-1} r_i$ in a final bottom-up process.

C. Public Randomness

For expositional clarity, we now summarize a series of inadequate strawman designs: (I) a naive, insecure design, (II) one that uses *commit-then-reveal* to ensure unpredictability but fails to be unbiasable and (III) one that uses secret sharing to ensure unbisability 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 wrong 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. 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 [34].

Strawman III. In order to address this issue, we want to ensure that a dishonest peer either cannot force the protocol to abort by refusing to participate or does not benefit from that. By using a (t, n)-secret sharing scheme, we can force the adversary to commit to his action *before* he can decide which action is favorable for 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 can blindly choose to abort the protocol or to distribute his share and allow honest peers to complete the protocol even if he stops participating upon seeing the recovered inputs. Unfortunately, a dishonest peer can still misbehave by sending out some bad shares to selectively chosen peers. The fact that each peer receives enough shares does not guarantee a successful recovery of the same secret by all peers.

The above protocol is a skeleton of RandShare, a small-scale unbiasable randomness protocol, that resists active bias attacks but is unable to scale to hundreds of participants.

D. RandShare: Small-Scale Unbiasable Randomness Protocol

RandShare is a small-scale unbiasable randomness protocol that provides our desired security properties: unbiasability, unpredictability, and availability. Because of its $O(n^3)$ communication complexity, it serves as an example which introduces central concepts that we re-use in the design of RandHound (Section III), where we additionally achieve scalability.

RandShare extends the approach for distributed key-generation in a synchronous model of Gennaro et al. [25] 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 [13], [14].

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 $\mathrm{N} = \{1, \ldots, n\}$ denote the list of peers that participate in RandShare and $n = 3f + 1$, where f is the number of dishonest peers. Further, let $t = f + 1$ be the VSS threshold. We assume that every peer has a copy of a public key X_j for all $j \neq i$ and only properly signed messages with valid session identifiers are accepted.

To run RandShare, every peer $i \in \mathrm{N}$ 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 $j \in \mathrm{N}$.
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) Setup a bit-vector $V_i = (v_{i1}, \ldots, v_{in})$ to keep track of valid secrets $s_j(0)$ and initialize it to all-zero. 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 can 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 the verification succeeds, confirm $s_j(i)$ by broadcasting the prepare message $(\mathtt{p}, i, j, 1)$ as a positive vote on the BA instance of $s_j(0)$. Otherwise, broadcast $(\mathtt{p}, i, j, s_j(i))$ 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 $(\mathtt{c}, i, j, 1)$ as a positive commitment. If there are at least $f+1$ negative votes for secret $s_j(0)$, broadcast $(\mathtt{c}, i, j, 0)$ as a negative commitment.
5) If there are at least $2f+1$ commits $(\mathtt{c}, i, j, x)$ 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 all Byzantine agreement instances finish 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 the collective random string as
$$\mathrm{R} = \bigoplus_{j=1}^{n'} s_j(0)$$
and publish it.

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 hence 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 R 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 client/server protocol for producing public, verifiable and unbiasable randomness. RandHound enables a client to "scavenge" public randomness from an arbitrary collection of servers. RandHound uses a commit-then-reveal approach for the randomness generation, implemented through publicly verifiable secret sharing (PVSS) [44], and it uses CoSi [49] as a witnessing mechanism to fix the protocol output and avoid client equivocation. Below, we first provide an overview of RandHound then introduce the notation and threat model; we describe in detail the randomness generation and verification phases of the protocol, provide and discuss security properties, and conclude with some extensions of the protocols.

A. Overview

RandHound switches to a client/server model, where a client uses a set of cooperating 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 the protocol run in the honest client scenario and to bias the output in the malicious client scenario. We assume that the client gets one attempt at running RandHound. In principle, the client could run the protocol arbitrarily many times until he obtains a random output that is favorable to him. However, each protocol run uses a session configuration file C that identifies a protocol run, and defines and binds it to the intended purpose of the random output. As a scenario illustrating RandHound's deployment model, we envision a lottery, in which the client is the authority running the lottery and must commit ahead of time to all lottery parameters, including the time and date of the lottery. A cryptographically unique hash of those configuration parameters included in C, identifies the specific run of RandHound to everyone. If that one and only run of the protocol failed to produce a drawing, it would set off an alarm triggering an investigative procedure, ensuring that the lottery authority does not get to covertly re-run the protocol.

RandHound improves upon RandShare by addressing scalability and availability concerns. RandShare, to ensure unbiasability, produces the random output as a function of all members' committed and globally agreed-upon inputs. This approach, however, results in a high computational and communication overhead as initially all members need to exchange shares of their input values with all other members to ensure that each secret input is recoverable. This limits the use of RandShare to small sets of highly-available servers.

In RandHound, instead of requiring that every server directly contributes its own secret input towards the random output, we use group secrets, generated only from inputs of the respective group members. The servers share their secrets only with their respective group members, decreasing the number of created and transmitted shares. First, the client arranges the servers into disjoint groups. Based on the pigeonhole principle, the resulting grouping is guaranteed to be secure, even for non-random, adversarial group choices. Each server chooses its random input value and creates shares only for the group members using PVSS, making the random input recoverable by a much smaller set of servers. Then 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 that, in turn, combines the recovered group secrets to produce the final random output R. The client documents the run of the protocol in a transcript T, 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.

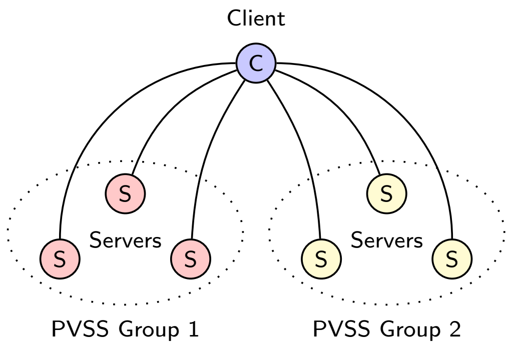


Fig. 1. An overview of the RandHound design.

B. Description

Let $\mathcal{G}$ be a group of large prime order q with generator G. Let $\mathrm{N} = \{0, \ldots, n-1\}$ denote the list of nodes, let $\mathrm{S} = \mathrm{N} \setminus \{0\}$ 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 $T_l \subset \mathrm{S}$, 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.

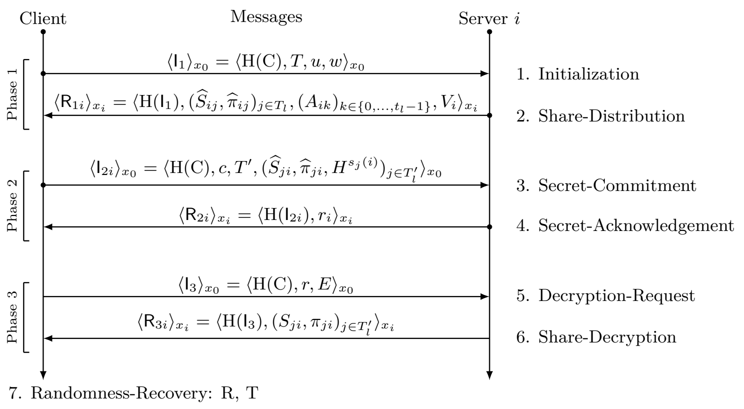
The publicly available session configuration is denoted by $C = (X, T, f, u, w)$, where $X = (X_0, \ldots, X_{n-1})$ is the list of public keys, $T = (T_0, \ldots, T_{m-1})$ is the server grouping, u is a purpose string, and w is a timestamp. We call $\mathrm{H}(C)$ the session identifier. The session 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 R which is publicly verifiable through a transcript T.

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:
   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 T.
   b) Prepare the message
   $$\langle \mathsf{I}_1 \rangle_{x_0} = \langle \mathrm{H}(C), T, u, w \rangle_{x_0},$$
   record it in T, 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 $\mathrm{H}(C)$ 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 $\widehat{\pi}_{ij}$ 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
   $$\langle \mathsf{R}_{1i} \rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_1), (\widehat{S}_{ij}, \widehat{\pi}_{ij})_{j \in T_l}, (A_{ik})_{k \in \{0,\ldots,t_l-1\}}, V_i \rangle_{x_i}$$



Client
Messages
Server i
Phase 1
Phase 2
Phase 3
$\langle \mathsf{I}_1\rangle_{x_0} = \langle \mathrm{H(C)}, T, u, w\rangle_{x_0}$
1. Initialization
$\langle \mathsf{R}_{1i}\rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_1), (\widehat{S}_{ij}, \widehat{\pi}_{ij})_{j\in T_l}, (A_{ik})_{k\in\{0,\dots,t_l-1\}}, V_i\rangle_{x_i}$
2. Share-Distribution
$\langle \mathsf{I}_{2i}\rangle_{x_0} = \langle \mathrm{H(C)}, c, T', (\widehat{S}_{ji}, \widehat{\pi}_{ji}, H^{s_j(i)})_{j\in T'_l}\rangle_{x_0}$
3. Secret-Commitment
$\langle \mathsf{R}_{2i}\rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_{2i}), r_i\rangle_{x_i}$
4. Secret-Acknowledgement
$\langle \mathsf{I}_3\rangle_{x_0} = \langle \mathrm{H(C)}, r, E\rangle_{x_0}$
5. Decryption-Request
$\langle \mathsf{R}_{3i}\rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_3), (S_{ji}, \pi_{ji})_{j\in T'_l}\rangle_{x_i}$
6. Share-Decryption
7. Randomness-Recovery: R, T

(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
$\widehat{\pi}_{ij}$ / π_{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
R	Collective randomness
T	Transcript

Fig. 2. An overview of RandHound randomness generation.

and send it back to the client.

3) **Secret Commitment (Client).** The client commits in this step to the shared secrets that contribute to the final random string, and he requests servers to co-sign his choice:
   a) Record each received $\langle \mathsf{R}_{1i}\rangle_{x_i}$ message in T.
   b) Verify all $\widehat{S}_{ij}$ against $\widehat{\pi}_{ij}$ 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 $(\widehat{S}_{ij}, \widehat{\pi}_{ij}, H^{s_i(j)})$ to the respective trustee.
   c) Create the commitment to the final list of secrets by randomly selecting $T'_l \subset T_l$ such that $|T'_l| = t_l$ for all $l \in \{0, \dots, m-1\}$.
   d) Compute the aggregate Schnorr commit $V = \prod_i V_i$ and the Schnorr challenge $c = \mathrm{H}(V \parallel \mathrm{H}(C) \parallel T')$.
   e) Prepare the message

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

   record it in T, 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 $f+1 \le \sum_{l=0}^{m-1} t_l$. Abort if any of those conditions does not hold.
   b) Compute the Schnorr response $r_i = v_i - cx_i$.
   c) Prepare the message

$$\langle \mathsf{R}_{2i}\rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_{2i}), r_i\rangle_{x_i}$$

   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 $\langle \mathsf{R}_{2i}\rangle_{x_i}$ message in T.
   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

$$\langle \mathsf{I}_3\rangle_{x_0} = \langle \mathrm{H}(C), r, E\rangle_{x_0},$$

   record it in T, 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 $\widehat{\pi}_{ji}$ 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 π_{ji}.
   d) Prepare the message

$$\langle \mathsf{R}_{3i}\rangle_{x_i} = \langle \mathrm{H}(\mathsf{I}_3), (S_{ji}, \pi_{ji})_{j\in T'_l}\rangle_{x_i}$$

   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 T.
   b) Check each share S_{ji} against π_{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

$$\mathrm{R} = \prod_{i \in \bigcup T'_l} S_{i0} \ ,$$

   and publish R and T.

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

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

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 R, 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 $\widehat{\pi}_{ij}$ 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 π_{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 R by recovering R' from the recovered individual secrets S_{i0} and by checking that $\mathrm{R} = \mathrm{R}'$. If the values are equal, then the collective randomness R is valid. Otherwise, reject R.

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 R with high probability.
2) **Unpredictability.** No party learns anything about the final random output R, except with negligible probability, until the secret shares are revealed.
3) **Unbiasability.** The final random output R represents an unbiased, uniformly random value, except with negligible probability.
4) **Verifiability.** The collective randomness R is third-party verifiable against the transcript T, that serves as an unforgeable attestation that the documented set of participants ran the protocol to produce the one-and-only random output R, 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 only be recovered 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 an active adversary who can arbitrarily misbehave, including refusing to participate in the protocol. A dishonest client can always choose to abort the protocol, which is equivalent to a self-DoS attack, and therefore, we do not consider it as 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 requiring the collaboration of malicious nodes. This ensures, with high probability, availability for an honest client. We refer to Section V-C for an analysis of the failure probability of a RandHound run for different parameter configurations.

Unpredictability. We want to ensure that output R 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 R 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 honest and dishonest nodes into 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. This means that there will be at least one group with at least an average number of honest nodes. Therefore, the threshold for secret recovery for each group l must be set 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 R is a function of all m group secrets, the adversary will not be able to reconstruct R 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 R.

In order to prevent the adversary from controlling the output R, 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 R that is 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 R.

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 R. 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 R. As a result, the honest member's input randomizes the group's secret and R, 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 R and generate multiple collective random values with valid transcripts from which he could choose and release the one that is most beneficial to him.

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

T 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 T 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 forges the transcript, that is, produces the transcript entirely on its own without an actual run of the protocol, then the adversary must be in possession of all secret keys of the participant listed in C, which violates 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 $(H^{s_i(j)})^a = H^{a s_i(j)} = G^{s_i(j)} = S_{ij}$, 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 R.

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 T. 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,

IV. RandHerd: A Scalable Randomness Cothority

In this section we introduce RandHerd, a randomness cothority, that extends the notion of a collective authority or a cothority [49] to unbiasable and verifiable randomness generation. Our objective is to produce a decentralized randomness beacon [42], [38], which efficiently generates not just one but a frequent, regular stream of random values. RandHerd retains many of the major RandHound's design choices, but it significantly improves upon RandHound in terms of repeat-execution performance.

We first give an overview of RandHerd followed by a detailed protocol description, then describe its security properties, and we end with some protocol extensions.

A. Overview

RandHerd is a continually-running decentralized service that generates fresh, publicly verifiable and unbiasable randomness on demand, in regular intervals, or both. It is implemented as a randomness cothority, a collective authority that consists of hundreds of diverse participants that partake in the randomness generation process. 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 gives an overview on the RandHerd design.

The RandHerd's design uses RandHound, CoSi [49] (collective witness cosigning) that implements a cothority, and a (t, n)-threshold Schnorr signature (TSS) scheme [48], as described in Section II, that implements threshold-based CoSi (TSS-CoSi).

The starting point for RandHerd is an existing cothority defined by the cothority configuration C, that among other parameters, lists the public keys of all participating servers and the collective public key X of the cothority. RandHerd consists of RandHerd-Setup, a one-time setup mode and RandHerd-Round, an operational mode that produces randomness.

In the setup mode, we use the output of RandHound to randomly select a RandHerd leader 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 $\widehat{X}$, that is endorsed by individual nodes in a run of CoSi.

In the operational mode of RandHerd, we produce a collective Schnorr signature $(\widehat{c}, \widehat{r})$ on some input w using TSS-CoSi and output $\widehat{r}$ as the randomness. We modify CoSi's original design to achieve bias-resistance in the presence of node failures. CoSi requires that either all nodes participate in the signing process or that an exception mechanism is used to account for missing nodes. Although sufficient for many application, the exceptions mechanism in the context of randomness would enable to bias the output by selectively preventing certain nodes from participating and including their input. In RandHerd, all m groups contribute towards RandHerd's output, however, each group's contribution requires the participation of only t_l group members. The selection of each group's contributions towards group secrets is certified by individual nodes in a run of CoSi. This enables a reasonable fraction of all nodes to be absent without biasing $\widehat{r}$. The key improvement over RandHound is the fact that whereas randomness is still produced based on secrets contributed by all m groups, instead of PVSS, we use a much more efficient TSS scheme to construct group secrets in the threshold setting. Further, the properties of CoSi ensure that the random output of RandHerd can be easily verified using $\widehat{X}$.

B. Description

Let $\mathrm{N} = \{0, \ldots, n-1\}$ denote the list of available nodes and let f denote the maximum number of permitted Byzantine nodes. We assume that $n = 3f + 1$. Private and public key of node $i \in \mathrm{N}$ are identified with x_i and $X_i = G^{x_i}$, respectively. Further, let C denote the cothority configuration file that includes the public keys of all nodes, the collective public key of the cothority X, 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.

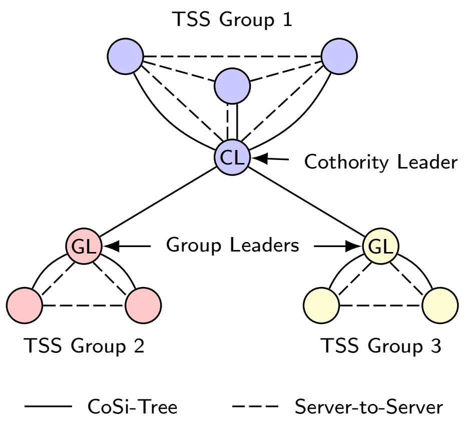


Fig. 3. An overview on the RandHerd design

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

1) **Leader Election.** Upon the occurrence of a predefined event, which might be specified in C, each node generates a lottery ticket $t_i = \mathrm{H}(C \parallel X_i)$ for every $i \in \mathrm{N}$ 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 (temporary) RandHerd leader. If the leader is unavailable, the node in the ascending order steps in to take the place of the leader. This process is repeated until an available node has been found. Section IV-E1 further discusses leader election and proposes a randomized lottery mechanism.
2) **Seed Generation.** The leader assumes the role of the RandHound client and runs the protocol together with all other nodes. A given leader has one chance to execute this step. If he fails, the next node, as specified by the outcome of the lottery, steps in and attempts to execute RandHound. After a successful run of RandHound, the leader obtains the tuple (R, T), where R is a collective random string and T is the publicly verifiable transcript that proves the validity of R. Lastly, the current leader broadcasts (R, T) to all nodes.
3) **Group Setup.** Once the nodes receive (R, T), they use T to verify R, and then use R 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 C.

After a successful setup, RandHerd switches to the operational randomness generation mode.

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 multiple times to produce the desired number of random outputs:

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 $\widehat{r}_l = \widehat{v}_l - \widehat{c}x_l$ 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 R with high probability.
2) **Unpredictability.** No party learns anything about the final random output R, except with negligible probability, until the group responses are revealed.
3) **Unbiasability.** The final random output R represents an unbiased, uniformly random value, except with negligible probability.
4) **Verifiability.** The collective randomness R is third-party verifiable as a collective Schnorr signature under $\widehat{X}$.

In the discussion below, 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" [15], [32] 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.

We note that the high-level design of RandHerd closely resembles the one 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, based on the analysis of the unpredicability property of RandHound, we know that in RandHerd, there will also be at least one 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 $\widehat{r}$.

Applying the arguments used in the discussion of unbiasability of 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 (if the next selected leader is honest) or more (if the next selected leader is dishonest or the adversary can DoS the selected honest leader) alternative values of $\widehat{r}$. Although the adversary cannot freely choose the next value of $\widehat{r}$ and cannot go back to the previous value if the next one is not more advantageous, he nonetheless has an option to try for another value of $\widehat{r}$ which constitutes bias. This bias, however, is limited as the view-change schedule must eventually appoint an honest leader, at which point the adversary no longer has any bias opportunity. Section IV-D further discusses this issue and proposes alternative designs of RandHerd that address this limited, "one-per-view-change" bias opportunity in the case of a dishonest leader.

Verifiability. The random output $\widehat{r}$ generated in RandHerd is obtained from a collective 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 CoSi 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 in the previous section, 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 reach Byzantine Fault Tolerant (BFT) consensus on $\widehat{V}$ and consequently the global challenge $\widehat{c} = \mathrm{H}(\widehat{V} \parallel w)$. 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 to stone. Therefore, if a leader fails, instead of restarting RandHerd, we can select a new leader, (for example by using the group ordering from the setup phase) whose only allowed action is to continue this protocol's execution. This removes the possibility of biasing the output by a dishonest leader and the incentive to fail a protocol execution.

However, using a traditional BFT protocol (*e.g.*, PBFT [15]), like we did in RandShare, yields prohibiting performance for RandHerd because of the large number of servers that participate in the protcol. In order to overcome this obstacle, we use BFT-CoSi from ByzCoin [32], 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. This means that even if the dishonest leader, in an attempt to bias the output, chooses to fail the protocol, the new (eventually honest) leader will be able to recover $\widehat{r}$ allowing all honest servers to successfully complete the protocol.

E. Extensions

1) Randomizing Temporary-Leader Election: The current set-up phase of RandHerd uses a very 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. RandHound'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) [36], 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.

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) [11] 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 collectively form a fresh Schnorr commitment, 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 them 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, because 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

In this section we discuss the evaluation of our prototype implementations for 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 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 and made it publicly available on GitHub[1]. Table I shows rounded up numbers for the lines of code (LoC) of our implementations. We relied on the cothority-framework (≈ 27000 LoC) for CoSi and for handling network communication, and used the corresponding crypto-library[2] (≈ 20000 LoC) for basic cryptographic operations (such as Curve25519 [5] arithmetic, etc.).

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 using 32 physical machines, each equipped with an Intel Xeon E5-2650 v4 (24 cores at 2.2 GHz), 64 GBytes of RAM, and a 10 Gbps network link. To obtain more realistic simulation results we restricted the connections to 100 Mbps and imposed an 200 ms round-trip latency.

To run the experiments with up to 1024 nodes, we oversubscribed the servers by a factor of up to 32, and arranged the nodes in such a way that most messages had to go through the network. In order to test the influence of the oversubscription on our experiments, we ran the same simulations also with 16 servers only. This resulted in an overhead increase of about 20%, indicating that we are CPU-bound and not network-bound. In real-world deployments of our systems we can therefore expect slightly better results than the values from our simulations.

2) RandHound: Fig. 4 shows the CPU-usage costs of a complete RandHound run that first generates a random value that is validated subsequently via the transcript. For the randomness-generation part, we measured the total costs across all servers, plus the costs of the client; whereas, for the verification, we only measured the costs for the client or equivalently a single node. In a setup with 1024 nodes and a group size of 32, for example, the complete RandHound run (generation + verification) requires about 10 CPU minutes, which is equivalent to costs of about \$0.02 on Amazon EC2.

Fig. 5 shows the wall-clock time of a complete RandHound run for different configurations. This corresponds to the duration of a RandHound execution, from the point when the protocol is started until the client has computed the collective randomness and verified it. Our measurements show that the time required by the servers is negligible, hence not depicted

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

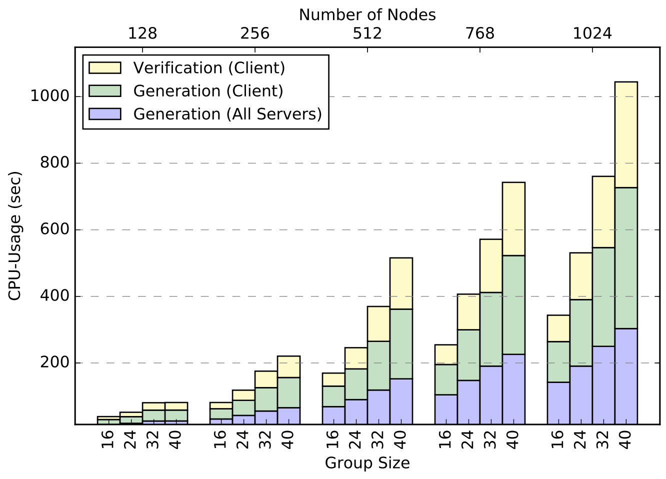


Fig. 4. System CPU-usage of RandHound

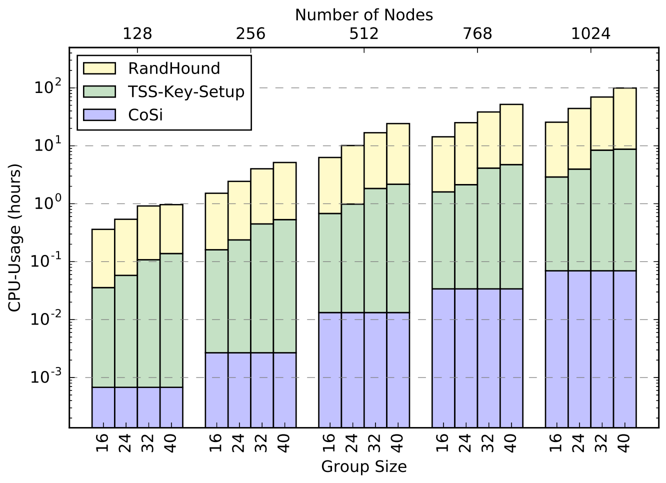


Fig. 6. System CPU-usage of RandHerd setup

in Fig. 5. In the above configuration RandHound randomness generation and verification consume roughly 290 and 160 seconds, respectively.

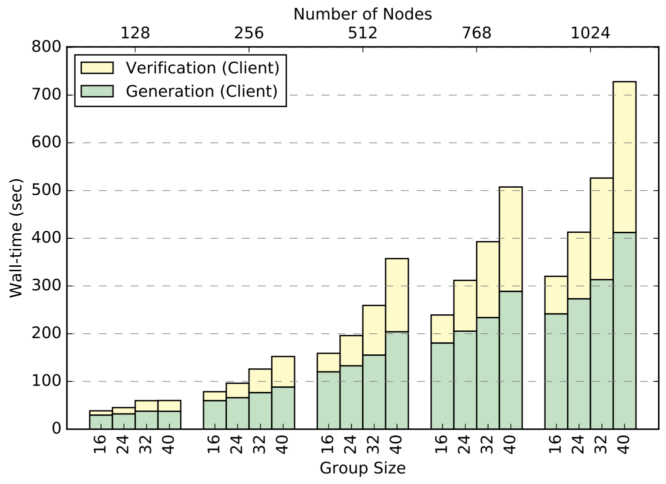


Fig. 5. Wall-clock time of RandHound

3) RandHerd: Before RandHerd is able to produce randomness it needs to go through a setup phase, using RandHound for the intial randomness, and CoSi for siging the RandHerd collective key. The results of our measurements for the CPU-usage are depicted in Fig. 6. These indicate that for 1024 nodes and a group size of 32, the RandHerd setup requires roughly 40 CPU hours which corresponds to \$4.00 on Amazon EC2. The associated wall-clock time we measured (not depicted in the graphs) amounts to about 10 minutes.

After the setup, RandHerd efficiently produces random numbers. Fig. 7 illustrates the wall-clock time results of our measurements in order for a single execution of the RandHerd round function to generate a 32-byte random value. For our example with 1024 nodes and a group size of 32, RandHerd needs slightly more than 6 seconds to finish one run. The corresponding CPU-usage (not shown in the graphs), across the entire system, amounts to roughly 30 seconds.

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.

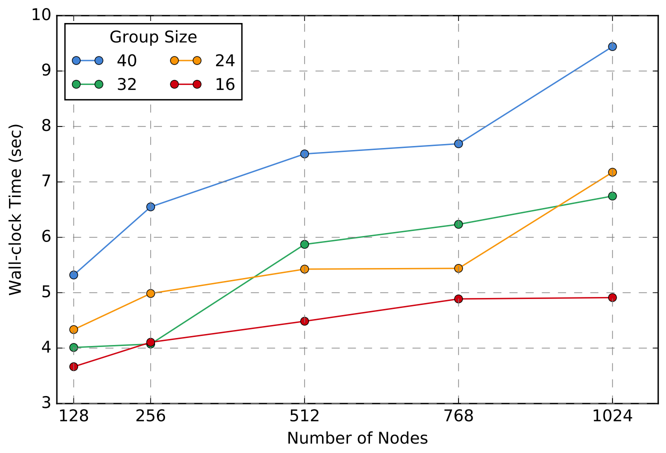


Fig. 7. Randomness creation time in RandHerd

In Fig. 8 we compare the overall communication costs for CoSi, RandHound, and RandHerd, with varying server numbers and a group size of 32 nodes. For the example of 1024 nodes, CoSi and RandHound require about 15 and 25 MB, respectively, whereas RandHerd needs about 400 MB. This is expected, due to the higher in-group communication of RandHerd. Note that these values correspond to the sum of the communication costs of the entire system and, considering the number of servers involved, are thus still fairly moderate.

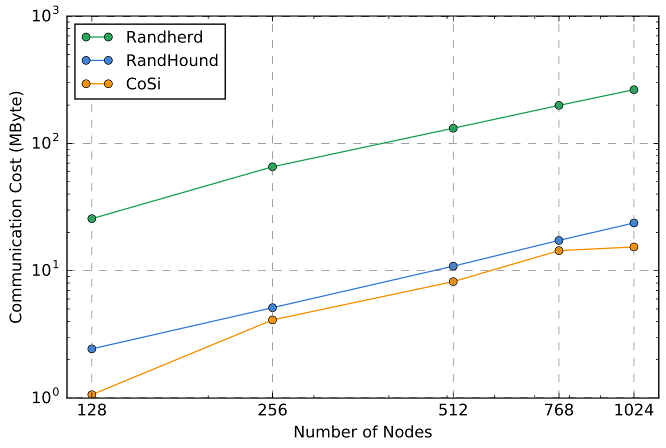


Fig. 8. Comparison of communication costs for RandHerd, RandHound, and CoSi (group size $c = 32$)

Finally, in Fig. 9 we compare RandHound and RandHerd against a non-scalable version of RandHerd that uses only one group. This variant is basically equivalent to RandShare, see Section II-D, and has communication complexity of $O(n^3)$. In comparison, RandHound has communication and computation complexity of $O(nc)$ and thus scales linearly, given a constant group size c. RandHerd has communication complexity $O(n + c^2)$ and scales linearly as well but the overhead increase is lower.

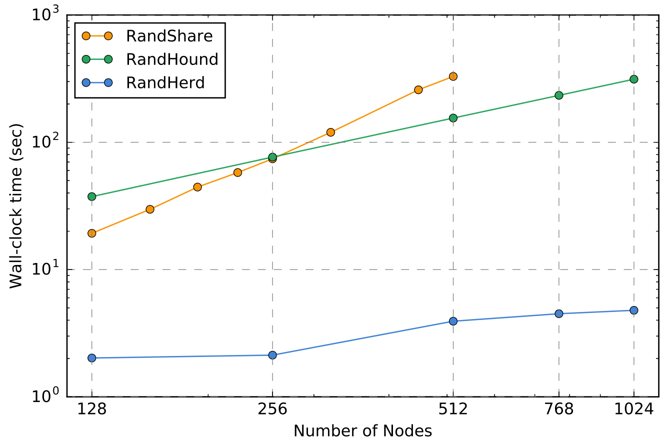


Fig. 9. Comparison of randomness generation times for RandShare, RandHound, and RandHerd (group size $c = 32$)

C. System Failure Probability

An adversary who controls a disproportional number of nodes in a single group can threaten the availability in both RandHound and RandHerd. Assuming that nodes are assigned randomly to groups, we can model analogously the system failure probability for both protocols. Note that this assumption excludes the scenario where the adversary controls the client in RandHound, as his goal is there to bias the output and not disrupt availability which would be equivalent to a self-DoS.

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 [47]

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

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

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

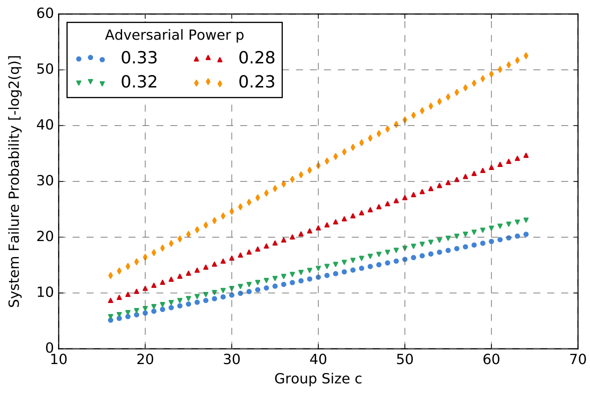


Fig. 10. System failure probability for varying group sizes

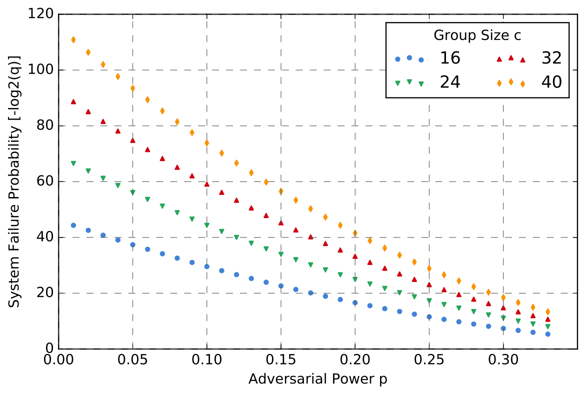


Fig. 11. System failure probability for varying adversarial power

Applying the union bound on this result, we obtain Figs. 10 and 11 that show the average system failure probabilities q for 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 given as $-\log_2(q)$, meaning that the higher the points in the graph are, the lower the failure probability of the system is. 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 already studied in various contexts: Rabin introduced the notion of randomness beacons in cryptography [42] in 1983, that was later adopted by NIST who launched their own online service to provide public randomness [38] from high-entropy sources. However, these centralized beacons lack transparency and potential users have to rely on the trustworthiness of the party that controls the service. There are some approaches, though, that increase transparency and sometimes even work completely without trusted third parties [41]. Bonneau et al. [12] show how to use Bitcoin for collecting entropy, however this work focuses on the *financial costs* of a given amount of bias instead of preventing it. Another approach from Lenstra et al. [33] implements a new cryptographic primitive, namely a slow hash function, in order to prevent a client from biasing the output. This approach assumes that the network is loosely synchronous so that everyone gets the commitment before the hash function produces the output. Nevertheless, if an adversary manages delay the commitment messages enough, he can still see the output of the hash-function before committing. Then the unbiasability depends upon whether the client accepts this delayed commitment as valid or not, and how slow is *slow-enough*, in order to be sure that all commits have been delivered. Other approached use lotteries [2], or financial data [18] as a source for public randomness.

An important observation by Gennaro et al. [25] is that in many distributed key generation protocols [40] 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, on the other hand proposed an asychronous distributed coin tossing scheme [14] for public randomness-generation that relies on a single trusted dealer for share distribution. We improve on that by letting multiple nodes deal secrets and combine them for randomness generation in our protocols. Finally, Kate et al. [31], 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 Genaro'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 [28], the selection of elliptic-curve parameters [2], [33], Byzantine consensus [39], electronic voting [1], the random sharding of nodes into groups [29], and non-interactive client-puzzles [30]. In all of these cases, both RandHound and RandHerd can be valuable assets for generating bias-resistant and 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 [14], [31]. 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.

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] 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.

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

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

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

[38] NIST Randomness Beacon.

[39] 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.

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

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

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

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

[44] 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.

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

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.