Poster: Scalable Bias-Resistant Distributed Randomness

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

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

I. Introduction

Bias-resistant public randomness is a critical component required in many (distributed) protocols. Tor hidden services depend on a fresh random value generated each day for protection against popularity estimations and DoS attacks. Anytrust-based systems such as Herbivore, Dissent, and Vuvuzela use bias-resistant public randomness for scalability by sharding participants into smaller groups. TorPath critically depends on public randomness for setting up consensus groups. Public randomness can be used to select parameters for cryptographic protocols or standards transparently, like in the generation of elliptic curves where adversaries should not be able to stir the process to select curves with weak security parameters Voting systems, elections, polls, lotteries, and some Byzantine agreement algorithms also critically depend on public randomness and illustrate its importance.

The process of generating public random values, however, is not trivial because sources of good randomness (in terms of entropy) are difficult to obtain for regular users. NIST deployed a publicly-accessible randomness beacon that provides hardware-generated random output and addresses the challenge of producing good randomness in terms of entropy. The beacon's design assumes that everyone trusts NIST which is not a reasonable assumption, though, especially given issues such as the Dual EC DRBG debacle.

II. Goals

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 attractive, especially in a collaborative setting, where a significant number of users wishes to participate. Proposals discussing such approaches utilize Bitcoin, slow cryptographic hash functions, lotteries, or financial data as sources for public randomness.

Our goal is to provide *unpredictable* and *unbiasable* public randomness in the familiar t-of-n threshold security model already widely-used in threshold cryptography and Byzantine consensus protocols. Generating public randomness is hard, however, because active adversaries may 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. While dealing with those issues is relatively straightforward [2] for small values of $n \approx 10$, we address scalability challenges of using large values of $n \approx 1000$ for enhanced security in real-world scenarios.

III. Design

In this section we describe three solutions to bias-resistant public randomness: RandShare, a motivational example that lays the foundation for our other proposals, RandHound, a client/server protocol that addresses RandShare's scalability issues, and RandHerd, an efficient, distributed randomness beacon that utilizes RandHound for setup and reconfiguration.

At first, designing a protocol to produce a public random value seems straightforward among a group of n clients, one of which is honest and can contribute a "good" random input. We could collect random inputs $r_1, \cdots, r_n$ from n clients and XOR them together, but the last client to provide his input can first look at everyone's inputs and then completely decide the output. We can require clients to send commitments first, then reveal the corresponding random inputs. Again, however, if the last client to reveal does not like the output of the protocol, it can either send a bad opening or refuse to reveal, forcing everyone to abort and re-run the protocol, until the output is favorable to the obstinate client. Applying a secret sharing scheme [4] to each client's input such that it can be recovered so long t out of n clients cooperate, brings us one step closer to addressing this active attack but it assumes a trusted dealer. To eliminate the trusted dealer, RandShare uses a verifiable secret sharing scheme [1]. This results in a scheme that is secure against active disruptions and has been used in various ways before [2] but has a $O(n^3)$ complexity.

A. RandHound

For the rest of the paper we assume that out of n servers fewer than one-third are maliciously colluding.

RandHound switches to a client/server model, in which a client invokes the services of a set of RandHound servers to produce a random value (Figure 1). RandHound addresses the scalability issues of RandShare by sharing secrets not directly among all other servers but only within smaller groups of servers. The client first arranges the servers into disjoint, balanced groups for scalability. Application of the pidgeonhole principle ensures the unpredictability and unbiasability of RandHound's final output even if some subgroups are compromised, *e.g.*, due to biased grouping. Then, each server pre-shares its random input among only the members of its group instead of the full group of n servers (out of which $t \leq k$ are sufficient to reconstruct the secret). This reduces the communication and computational overhead from $O(n^3)$ to $O(nc^2)$, where c is the average (constant) size of a group.

RandHound consists of *randomness generation* and *randomness verification* phases. During randomness generation,

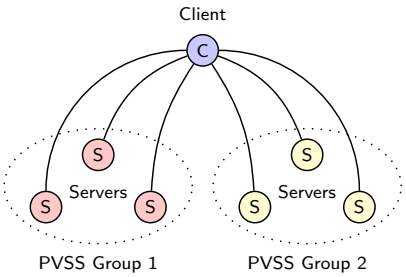

Fig. 1. An overview of the RandHound design.

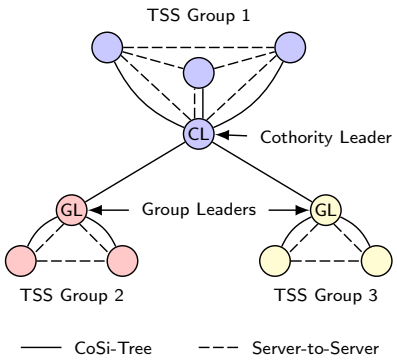

Fig. 2. An overview on the RandHerd design

the client works with RandHound servers to produce a random string Z which is publicly verifiable through a log L, or *transcript*, where the client documents the run of the protocol by recording the messages he sends and receives. The transcript serves as a third party verifiable proof of the produced randomness. During randomness verification, any external verifier can check the validity of Z against the transcript L. RandHound's randomness generation consists of three *inquiry-response* phases between the client and the servers followed by the client's randomness recovery.

1) **Initialization (Client).** The client broadcasts a session configuration C consisting of all participants' public keys, server groupings, a timestamp and a session identifier.
2) **Share Distribution (Server).** Each server chooses its random input and creates shares for members of the same group using step 1 of PVSS [3], a publicly verifiable secret sharing scheme, and sends encrypted shares to the client together with the NIZK proofs of correctness.
3) **Secret Commitment (Client).** To tolerate server failures, the client selects and commits to a subset of secret inputs from each group that contribute to the final random string Z, and asks servers to co-sign his choice using CoSi [6], an efficient co-signing protocol that produces a collective Schnorr signature under an aggregate public key.
4) **Secret Acknowledgment (Server).** Each server acknowledges the client's commitment by participating in CoSi preventing client's equivocation.
5) **Decryption Request (Client).** After a successful CoSi run, the client requests the decryption of the secrets.
6) **Share Decryption (Server).** Each server decrypts the shares received from the client using step 2 of PVSS.
7) **Randomness Recovery (Client).** The client combines the recovered secrets to produce the final random output Z by performing step 3 of PVSS.

B. RandHerd

RandHound's design offers a scalability and performance improvements but it is still not efficient enough to run frequently, e.g., once a minute, a rate comparable to NIST's randomness beacon. RandHerd is a collective randomness authority or *cothority* [6] that uses RandHound for bootstrapping or occasional reconfiguration, but then produces randomness at frequent intervals much more efficiently. RandHerd reduces communication and computational overhead from RandHound's $O(c^2 n)$ to $O(c^2 \log n)$ given a group size c.

RandHerd runs continually and need not be initiated by any client, but requires stateful servers (Figure 2). No single or sub-threshold group of failing or malicious servers can halt the protocol, or predict or significantly bias its output. Clients can check the trustworthiness of any published beacon output with a single, efficient check of one collective signature [6].

The RandHerd protocol consists of RandHerd-Setup, which performs one-time setup, and RandHerd-Round, which produces successive random outputs.

1) **RandHerd-Setup.** RandHerd first invokes RandHound to divide the set of servers securely into uniformly random groups. Then, it uses TSS-CoSi, a threshold-based witness cosigning protocol [5], [6] , to generate and endorse a short-term aggregate public key used to produce and verify individual random outputs.
2) **RandHerd-Round.** To produce each random output at regular intervals, RandHerd generates a collective Schnorr signature $(\widehat{c}, \widehat{r})$ on some input w using TSS-CoSi and outputs $\widehat{r}$ as randomness. All RandHerd groups contribute to each output ensuring honest members' participation, but each group's contribution requires the participation of only a threshold of members, which improves availability. Each of RandHerd's random outputs doubles as a collective Schnorr signature [5], [6], which clients can validate efficiently against the group's aggregate public key.

REFERENCES

[1] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, 1987.
[2] T. P. Pedersen. A threshold cryptosystem without a trusted party. In *Eurocrypt*, 1991.
[3] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *CRYPTO*.
[4] A. Shamir. How to share a secret. 1979.
[5] D. R. Stinson et al. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *ACISP*, 2001.
[6] E. Syta, et al. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *IEEE Symposium on Security and Privacy*, 2016.