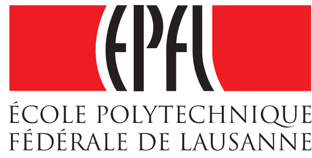


RandShare: Small-Scale Unbiasable Randomness Protocol

Mathilde Raynal

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Philipp Jovanovic
EPFL / DEDIS

Contents

Introduction **3**

Background and Motivations **5**
Insecure Approaches To Public Randomness 5
Lagrange Interpolation 6
Publicly Verifiable Secret Sharing 7
Terminology 7

Implementation **8**
RandShare 8
RandSharePVSS 10

Results and Evaluation **14**

Limitations **16**
Practical limitations 16
Theoretical limitations 16

Future Work **17**

Installation **18**

References **18**

Introduction

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." - *John von Neumann*

The process of generating public randomness is nontrivial, because obtaining access to sources of good randomness, even in terms of entropy alone, is often difficult and error prone [4], [5]. Producing and using randomness in a distributed setting presents many issues and challenges, such as how to choose a subset of available beacons, or how to combine random outputs from multiple beacons without permitting bias by an active adversary. Prior approaches to randomness without trusted parties employ Bitcoin [6], [7], slow cryptographic hash functions [8], lotteries [3], or financial data [15] as sources for public randomness.
A reliable source of randomness that provides high-entropy output is a critical component in many protocols [9], [10]. 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 provided [11].

In the following, we illustrate, through a series of strawman protocols, the key challenges in distributed randomness generation of commitment, selective aborts, and malicious secret shares.

In *Scalable Bias-Resistant Distributed Randomness* [1] the protocol RandShare is introduced as a way to provide bias-resistant public randomness in the familiar (t, n)-threshold security model already widely used both in threshold cryptography [12], [13] and Byzantine consensus protocols [14].

RandShare is an unbiasable randomness protocol that ensures unbiasability, unpredictability, and availability, but is practical only at small scale due to $O(n^3)$ communication overhead. To tolerate server failures, the client selects a subset of secret inputs from the nodes after a vote. Application of the pidgeonhole principle ensures the integrity of final output. After the servers release the selected secrets, the client combines and publishes the collective random output.

RandShare introduces key concepts that will be used in the more scalable RandHound protocol [1]. The main goal of RandShare is to remove the incentive for a Byzantine adversary (passively listening on or actively corrupting data in the channel) to misbehave. This is achieved by forcing the adversary to make a decision whether or not to follow the protocol early enough before he has any information about the output preventing him from devising a successful dishonest strategy. More concretely, RandShare extends the approach for distributed key generation in a synchronous model of Gennaro et al. [16] by adopting a point-of-no-return strategy and extending it to the asynchronous setting where the adversary can break timing assumptions [17], [15]. We use the concept of a *barrier*, a specific point in the protocol execution after which the protocol always completes successfully. Specifically, we define the barrier in the protocol as the point when the first honest member reveals the shares he is holding.

Before the barrier, the protocol output is fixed by all participants and no peer gets any information about the secrets of other honest peers. While it is of course not possible to prevent a malicious peer from refusing to participate, he does not obtain any information on the final output. Furthermore, assuming that all messages are eventually delivered, the protocol preserves liveness without the adversaries cooperation. Consequently, he has to randomly decide whether or not to participate which guarantees unbiasability and hence output integrity. After the barrier, the protocol output cannot be changed anymore and all honest peers eventually output the previously fixed value, regardless of the adversary's behavior.

RandShare performs verifiable secret sharing among all n nodes, i.e., verification of the output can be done only by an active participant of the protocol run. As the secret is computationally hidden, there is no way for a third-party to verify the output. It must be noted that the absence of public-verifiability of the randomness generation process is a major setback. To overcome it, we introduced PVSS to our project, creating RandSharePVSS. The random output is now produced along with a third-party verifiable transcript of the protocol run. Anyone can subsequently check this transcript to verify that the random output is trustworthy and unbiased, provided not too many servers were compromised.

Background and Motivation

Insecure Approaches To Public Randomness

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

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

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

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

Lagrange Interpolation

The Lagrange Interpolation is a method of data fitting that, given a set of constraints, i.e., data points, computes a polynomial of lowest possible degree that fits the data exactly, in other words, that passes through all the points of the set.

Given a set of $k+1$ data points: $(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$ where no two x_j are the same, the interpolation polynomial in the Lagrange form is a linear combination:

$$L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$$

of Lagrange basis polynomials:

$$\ell_j(x) := \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)},$$

where $0 \le j \le k$. Given the initial assumption that no two x_j are the same, $x_j - x_m \ne 0$, this expression is always well-defined.
For all $i \ne j$, $\ell_j(x)$ includes the term $(x - x_i)$ in the numerator, so the whole product will be zero at $x = x_i$:

$$\ell_{j \ne i}(x_i) = \prod_{m \ne j} \frac{x_i - x_m}{x_j - x_m} = \frac{(x_i - x_0)}{(x_j - x_0)} \cdots \frac{(x_i - x_i)}{(x_j - x_i)} \cdots \frac{(x_i - x_k)}{(x_j - x_k)} = 0.$$

On the other hand,

$$\ell_i(x_i) = \prod_{m \ne i} \frac{x_i - x_m}{x_i - x_m} = 1.$$

In other words, all basis polynomials are zero at $x = x_i$, except $\ell_i(x)$, for which it holds that $\ell_i(x_i) = 1$. It follows that $y_i \ell_i(x_i) = y_i$, so at each point x_i, $L(x_i) = y_i + 0 + 0 + \cdots + 0 = y_i$, showing that L interpolates the function exactly.

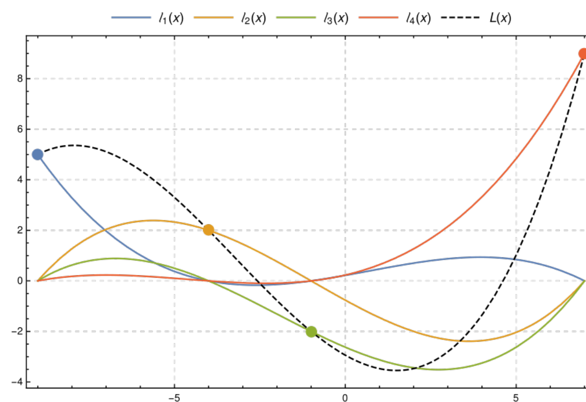


Figure 1: This image shows, for four points ((9, 5), (4, 2), (1, 2), (7, 9)), the interpolation polynomial L(x) (dashed, black), which is the sum of the scaled basis polynomials $y0\ell0(x)$, $y1\ell1(x)$, $y2\ell2(x)$ and $y3\ell3(x)$. The interpolation polynomial passes through all four points.

Publicly Verifiable Secret Sharing

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

A publicly verifiable secret sharing (PVSS) 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 of discrete log equivalence (DLEQ) that $E_i(s_i)$ correctly encrypts a valid share s_i of s. During the reconstruction phase, trustees recover s by pooling t properly-decrypted shares. They then publish s along with all shares and DLEQ proofs that show that the shares were properly decrypted.

PVSS runs in three steps:

1. The dealer chooses a degree $t-1$ secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ and creates, for each trustee $i \in \{1, \ldots, n\}$, an encrypted share $\hat{S}_i = X_i^{s(i)}$ 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 DLEQ encryption consistency proof $\hat{P}_i$. Afterwards, he publishes $\hat{S}_i$, $\hat{P}_i$, and A_j.
2. Each trustee i verifies his share $\hat{S}_i$ using $\hat{P}_i$ and A_j, and, if valid, publishes the decrypted share $S_i = (\hat{S}_i)^{x_i^{-1}}$ together with a DLEQ decryption consistency proof P_i.
3. The dealer checks the validity of S_i against P_i, discards invalid shares and, if there are at least t out of n decrypted shares left, recovers the shared secret S_0 through Lagrange interpolation.

Terminology

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 $X_i = G^{x_i}$.

Implementation

The presented implementations are in Go, and use DEDIS code (Crypto library ; Network library ; Cothority framework) [1].

RandShare

The implementation of RandShare follows the description that can be found in *Scalable Bias-Resistant Distributed Randomness* [1]:

Let $N = \{1, \dots, n\}$ denote the list of peers that participate in RandShare and $n = 3f+1$, where f is the number of dishonest peers. Let $t = f + 1$ be the VSS threshold. We assume every peer has a copy of a public key X_j for all $j \in N$.

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

1. Share Distribution.

1. Select coefficients $a_{ik} \in_R \mathbb{Z}_q^*$ of a degree $t-1$ secret sharing polynomial

$$s_i(x) = \sum_{k=0}^{t-1} a_{ik} x^k.$$

   The secret to be shared is $s_i(0) = a_{i0}$.
2. Compute polynomial commitments $A_{ik} = G^{a_{ik}}$, for all $k \in \{0, \dots, t-1\}$, and calculate secret shares $s_i(j)$ for all $j \in N$.
3. Securely send $s_i(j)$ to peer $j \neq i$ and start a Byzantine agreement (BA) run on $s_i(0)$, by broadcasting $\widehat{A}_i = (A_{ik})_{k \in \{0,\dots,t-1\}}$.

2. Share Verification.

1. Setup a bit-vector $V_i = (v_{i1}, \dots, v_{in})$ 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 may be done by checking that $S_j(i) = G^{s_j(i)}$ where:

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

3. If verification succeeds, confirm $s_j(i)$ by broadcasting the prepare message $(\mathtt{p}, i, j, 1)$ 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.
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.

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

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

4. Randomness Recovery.

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

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

The storage complexity of the protocol is $O(n^2)$. Every peer needs to keep the messages exchanged, and to collect shares in a matrix-like structure, as shown in Figure 2. As we are using a (t, n)-threshold security model, a secret is considered recoverable with t shares, thus optimization can be done by storing only t shares per line. It also allows us to reduce the overall wall-clock time by waiting for fewer shares.

$s_1(1)$	$s_1(2)$	...	$s_1(n)$
$s_2(1)$	$s_2(2)$	...	$s_2(n)$
...	...	...	...
$s_n(1)$	$s_n(2)$	...	$s_n(n)$

Figure 2: Matrix representation of shares storage, where share $s_i(j)$ is the j-th piece of the i-th secret

RandSharePVSS

In this section, we introduce the improved protocol that uses a (t, n)-PVSS scheme which allows third parties (i.e., someone who did not participate in the protocol) to verify the correctness of the protocol run and therefore of the produced randomness. Given a transcript of all exchanged information, we can validate the output with a time stamp by calling the verify method.

The implementation brought challenges that required other modifications:
Firstly, to prevent malicious peers from sending multiples shares and flooding the servers, a tracker is used to ensure that only one message is received from each peer at each step.
Secondly, as the contribution of the secret of one honest peer suffices to provide unpredictability, the collective string can be computed from a subset of $n' \geq f+1$ secrets. Hence the protocol can tolerate server failures and should not fail when less than n secrets are recoverable. We can output a valid collective string computed with n' secrets, $f+1 \leq n' \leq n$. And to chose which secrets will be used, a voting process based on how many correct shares we receive from a peer allowed us to exclude malicious peers at an early stage.
At last but not least, a session ID was attached to messages to prevent replay attacks and any bias from an external input. As PVSS verifications are using a 2nd base point, we chose it to be the hash of this session ID.

The protocol RandSharePVSS is specified as follow:

Let $N = \{1, \ldots, n\}$ denote the list of peers that participate in RandSharePVSS and $n = 3f+1$, where f is the number of dishonest peers. Further, let t be the PVSS threshold such that $f < t \leq n - f$. Henceforth we simply assume that $t = f+1$.
We assume a Byzantine adversary and an asynchronous network where messages are eventually delivered. Private and public key of peer i are denoted by $x_i \in_R \mathbb{Z}_q^*$ and $X_i = G^{x_i}$, respectively. We assume that every peer i has a copy of public key X_j for all $j \in$ N. Moreover, each message from each peer includes a unique session identifier. Peers are assumed to only accept messages that include the correct identifier.

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

1. **Share-Distribution.**
   (a) 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 $S_i = G^{s_i(0)} = G^{a_{i0}}$.
   (b) For all $j \in N$ compute share commitments $H^{s_i(j)}$, encrypted shares $\hat{S}_i(j) = X_i^{s_i(j)} = G^{s_i(j)x_i}$, and polynomial commitments $A_{ij} = H^{a_{ij}}$. Note that $H^{s_i(k)}$ can be recovered from $(A_{ij})_{j \in N}$ as follows:
   $$\prod_{j=0}^{t-1} A_{ij}^{(k^j)} = H^{\sum_{j=0}^{t-1} a_{ij} k^j} = H^{s_i(k)} \ .$$
   (c) Create encryption consistency proofs $\hat{P}_{ij}$.
   (d) Publish $(\hat{S}_i(j))_{j \in N}$, $(\hat{P}_{ij})_{j \in N}$, and $(A_{ij})_{j \in N}$.
2. **Voting Process.** To verify the received encrypted shares $(\hat{S}_j(k))_{k \in N}$ received from node j, and select a subgroup of peers participating in the next step, each trustee i executes the following steps:

(a) Setup a bit-vector $V_i = (v_{i1}, ..., v_{in})$ to keep track of valid messages and initialize it to all-zero.
(b) For all $k \in N$, verify $\hat{S}_j(k)$ against $\hat{P}_{jk}$ by reconstructing $H^{s_j(k)}$ from $(A_{jk'})_{k' \in N}$ and by checking that

$$\log_H H^{s_j(k)} = \log_{X_j} \hat{S}_j(k).$$

(c) If more than $f + t$ encrypted shares are valid, put v_{ij} to 1.
(d) Wait until a decision has been taken for all entries and publish V_i.

3. **Share-Decryption.** To decrypt his share, trustee i executes the following steps:
(a) Sum all the votes received in a vector $V = (v_1 = \sum_{j \in N} v_{j1}, ..., v_n = \sum_{j \in N} v_{jn})$. Determine the number n' of entries in V such that $v_k > f$. If $n' \leq f$, abort. Otherwise, Let $N' = \{1, \dots, n'\}$ denote the list of peers whose vote was greater than f.
(b) Decrypt $\hat{S}_j(i)$ using x_i and obtain $S_j(i) = \hat{S}_j(i)^{x_i^{-1}}$.
(c) Create a decryption consistency proof P_{ji}.
(d) Publish $(S_j(i))_{j \in N'}$, $(P_{ji})_{j \in N'}$.

4. **Secret-Recovery.** Wait until at least t decrypted shares for each $j \in N'$ have arrived. To reconstruct every secret S_j, each peer i executes the following steps:
(a) Verify $S_j(k)$ against P_{jk} by checking that

$$\log_G X_j = \log_{S_j(k)} \hat{S}_j(k).$$

and discard $S_j(k)$ if the verification fails.
(b) Suppose w.l.o.g, for $1 \leq k \leq t$, that shares $S_j(k)$ are valid. Reconstruct secret S_j by Lagrange interpolation

$$\prod_{k=1}^{t} (S_j(k))^{\lambda_k} = \prod_{k=1}^{t} (G^{s_j(k)})^{\lambda_k} = G^{\sum_{k=1}^{t} s_j(k)\lambda_k} = G^{s_j(0)} = S_j$$

where $\lambda_k = \prod_{j \neq k} \frac{j}{j-k}$ is a Lagrange coefficient.
(c) Compute and publish the collective random string as:

$$Z = \bigoplus_{j=1}^{n'} S_j,$$

along with a third-party verifiable transcript of the protocol run.

To face race conditions, a situation where two or more threads access the same memory location concurrently and at least one of the accesses is for writing, we introduced a mutex that locks and unlocks the data, ensuring that simultaneous updates to the same part cannot occur, nor any loss of information.

The storage complexity stays $O(n^2)$. With a PVSS scheme, we need to collect the encrypted shares and the decrypted shares. We kept the same matrix-like structure as Figure 2. We have to be cautious with indices as a server i encrypts the shares $(\hat{S}_i(j))_{j \in N}$, which is, in our matrix representation, a row, but decrypts $(S_j(i))_{j \in N}$, which represents a column, as shown in Figure 3.

Optimization had to be done carefully. To recover the secret S_i, we need t verified decrypted shares $S_i(j)$, and the verification of $S_i(j)$ uses $\hat{S}_i(j)$, thus, we need t tuples $(\hat{S}_i(j)$, $S_i(j))$. Storing t encrypted shares per line was not enough to prevent failure due to malicious peers not sending their decrypted shares. See Figure 4 for an example of a setting where the

$\hat{S}_1(1)$	$\hat{S}_1(2)$	...	$\hat{S}_1(n)$
$\hat{S}_2(1)$	$\hat{S}_2(2)$	...	$\hat{S}_2(n)$
...	...	...	...
$\hat{S}_i(1)$	$\hat{S}_i(2)$	...	$\hat{S}_i(n)$
...	...	...	...
$\hat{S}_n(1)$	$\hat{S}_n(2)$	...	$\hat{S}_n(n)$

(a) Highlight of shares encrypted by server i, where $\hat{S}_i(j)$ is the j-th encrypted share of secret S_i.

$\hat{S}_1(1)$	$\hat{S}_1(2)$	...	$\hat{S}_1(i)$	...	$\hat{S}_1(n)$
$\hat{S}_2(1)$	$\hat{S}_2(2)$	...	$\hat{S}_2(i)$	...	$\hat{S}_2(n)$
...	...	...	...	...	...
$\hat{S}_n(1)$	$\hat{S}_n(2)$	...	$\hat{S}_n(i)$	...	$\hat{S}_n(n)$

(b) Highlight of shares decrypted by server i, where $S_i(j)$ is the j-th decrypted share of secret S_i.

Figure 3: Share encryption and decryption

protocol fails when the malicious node 1 does not send its decrypted shares. To make sure that we would end up with t tuples $(\hat{S}_i(j), S_i(j))$, we have to store $f + t = 2 \cdot f + 1$, assuming that $t = f + 1$, encrypted shares per line to be able to verify and keep at least t of the $n - f$ decrypted shares received without the participation of malicious nodes. As mentioned on the improved protocol storage optimization paragraph, not only we lower the storage cost, which, as $O(n^2)$, can be a problem when the number of nodes becomes large, but we also have to wait for fewer shares to arrive, and thus, the running time of the protocol improves.

$\hat{S}_1(1)$	$\hat{S}_1(2)$	$\hat{S}_1(3)$
$\hat{S}_2(1)$	$\hat{S}_2(2)$	$\hat{S}_2(3)$
$\hat{S}_3(1)$	$\hat{S}_3(2)$	$\hat{S}_3(3)$

(a) Received encrypted shares during 1st step

$\hat{S}_1(1)$	$\hat{S}_1(2)$	X
$\hat{S}_2(1)$	$\hat{S}_2(2)$	X
$\hat{S}_3(1)$	$\hat{S}_3(2)$	X

(b) t verified encrypted shares are stored per line

X	$S_1(2)$	$S_1(3)$
X	$S_2(2)$	$S_2(3)$
X	$S_3(2)$	$S_3(3)$

(c) Received decrypted shares during 3rd step, malicious node 1 does not send its own shares

X	$S_1(2)$	$S_1(3)$
X	$S_2(2)$	$S_2(3)$
X	$S_3(2)$	$S_3(3)$

(d) As we need $\hat{S}_i(j)$ to verify $S_i(j)$, only shares in green can be verified. In red shares that can not be verified and will be discarded

X	$S_1(2)$	X
X	$S_2(2)$	X
X	$S_3(2)$	X

(e) Results in less than t decrypted shares per line, thus less than t tuples ($\hat{S}_i(j)$, $S_i(j)$)

Figure 4: Example of a failed protocol when storing t encrypted shares per line with a setup of 3 peers and a threshold of 2

Results and Evaluation

Security Properties

The RandSharePVSS protocol provides the following security properties: unbiasability, unpredictability, availability and third party verifiability.

In the discussion below, we assume that honest peers follow the protocol and that all used cryptographic primitives provide their intended security properties. In particular the (n, t)-public-verifiable secret sharing (PVSS) scheme ensures that a secret can only be recovered using a minimum of t shares and that the shares do not leak information about the secret.

Unbiasability. The final random output Z represents an unbiased, uniformly random value, except with negligible probability. To prevent dishonest peers from recovering the honest peers' secrets prematurely, i.e., before the barrier, and therefore be able to bias the output by deciding whether or not to fail the protocol, we require that the secret sharing threshold is $t = f + 1$. This also means that the scheme can tolerate up to f dishonest peers. The Byzantine agreement procedures moreover ensure that all honest peers have the consistent copies for their vector V_i and therefore know which $n' > f$ secrets will be recovered after the barrier or if the protocol run has already failed as $n' \leq f$.

Unpredictability. No party learns anything about the final random output Z, except with negligible probability, until the secret shares are revealed i.e., before barrier. Since there are at most f malicious peers, and no honest peer will release his shares before the barrier, the attacker knows the values of at most f secrets. As the final random string Z contains $n' \geq f + 1$ secrets, and therefore at least one secret from an honest peer, Z is unpredictable except with negligible probability.

Availability. After the barrier, honest participants are able to complete the protocol run and produce the random output Z with high probability. The secret sharing threshold of $t = f + 1$ ensures that the $f + 1$ honest nodes out of the total $2f + 1$ positive voters, are able to recover the secrets corresponding to the 1-entries in V_i, and therefore Z, without the collaboration of the dishonest nodes.

Experimental Results

To test our implementation, we deployed it on DETERLAB's clusters with 10 machines, each equipped with an Intel(R) Xeon(R) E3-1260L quad-core processor running at 2.4 GHz, 16GB of RAM, and imposed 200 ms round-trip latencies on all communication links.

Figure 5 shows the CPU-usage costs of a complete RandShare run that generates a random value from N servers.

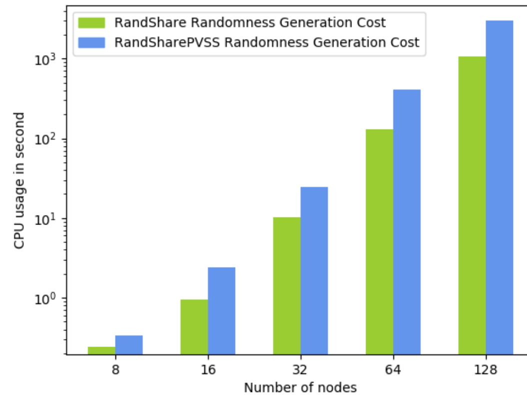


Figure 5: Overall CPU usage of a protocol run

Figure 6 shows the wall-clock time of a complete protocol run. This test measures total time elapsed from the start until the production of a random output for RandShare, and until the verification of a random output for RandSharePVSS.

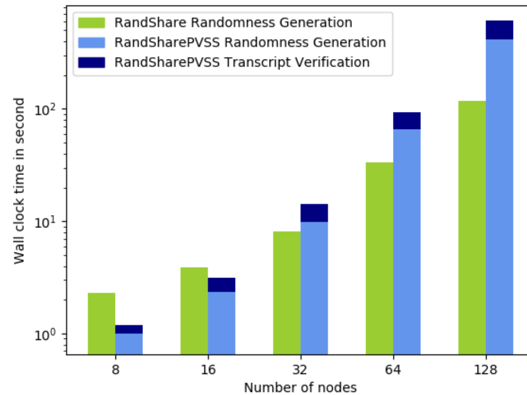


Figure 6: Total wall-clock time of a protocol run

While we lower the number of round trips between the servers from 4 for the basic protocol, to 3 by using PVSS, we see that the incurred computational tasks make the random generation process longer.

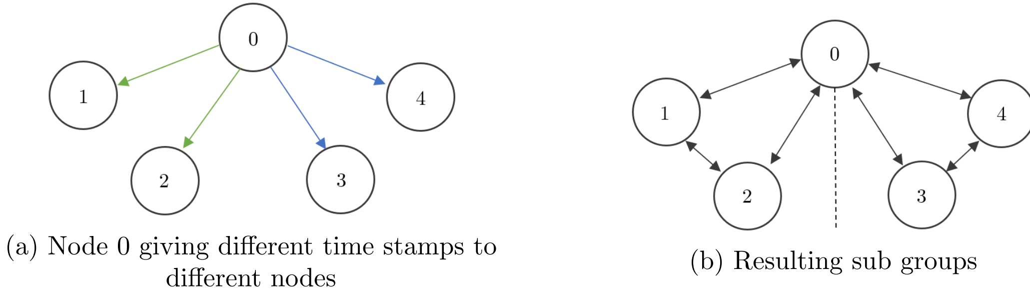
Limitations

Practical limitations

The main limitation of RandShare randomness generation protocol is that it is not scalable as it incurs $O(n^3)$ communication and computation costs on each of n participants when using a (t, n)-PVSS scheme. As we saw in the experimental results, with 128 nodes, a random string is produced after 6 minutes.

Theoretical limitations

In contrast of RandHound which uses a server-client model, in RandShare every node ends up with a copy of the collective string along with the transcript, and the initiator is a non-predefined node among the participants. Thus we have to think what would happen if the node starting the protocol, node 0 in Figure 7, would be malicious. Peers are assumed to only accept messages that include the correct identifier. Hence, by giving a different time stamp to two different subgroups of nodes, he could do a network splitting attack. For a network with n nodes, if the malicious node splits the group of $2/3 \cdot n$ honest nodes into to chunks of $1/3 \cdot n$ each. It would take 1/3 of its secrets (from the malicious nodes) plus 1/3 of the first honest group of nodes, creating a randomness $r1$ and then by doing the same process again, 1/3 of malicious secrets plus 1/3 of secrets from the second group of nodes, it would create a randomness $r2$. Since the adversary managed to compute two different legit random values $r1$ and $r2$, the output is not unique anymore.



(a) Node 0 giving different time stamps to different nodes

(b) Resulting sub groups

Figure 7: Example of a network splitting attack with 5 nodes, 1 faulty node (0) and a threshold of 2, creating two subgroups that do not accept messages from each other, resulting in the output of two valid random strings $r1 = S_1 \oplus S_2$ and $r2 = S_3 \oplus S_4$.

An other attack that can be realized by malicious nodes is impersonation, i.e., sending a bad share where the sender field would be set as the ID of a honest node. If that malicious node were to send it before the honest node sends its own correct message, it would discredit the honest node for the rest of the protocol as we accept only one message per node per step, and a vote excludes some nodes based on the number of correct shares they sent. If too many nodes are considered as seen as malicious, the protocol aborts. This attacks is thus a threat to availability.

Future work

Timeout: If malicious peers do not send their decrypted shares, we saw that we can still move on to the final step and output the collective string with high probability. We have now to think of what would happen if a malicious peer does not send its encrypted share. The protocol would actually wait until it finally timeouts.
An output is considered unbiased if the collective string is computed with at least one secret from an honest peer. We could thus set up a timeout for the initial step, where the peer would be evicted from the protocol if we didn't receive its shares after a certain time. If we consider that at least $f + 1$ secrets are recoverable we could move on the the next step, and abort otherwise.

Signing: Impersonation can be avoid by signing each message, so that the sender is easily identified. A variation of Schnorr (multi-)signature could be easly introduced into the protocol, for example the Threshold Signing [19] scheme, a distributed (t, n)-Threshold Schnorr Signature. TSS allows any subset of t signers to produce a valid signature. During setup, all n trustees use VSS to create a long-term shared secret key x and a public key $X = G^x$. To sign a statement S, the n trustees first use VSS to create a short-term shared secret v and a commitment $V = G^v$ and then compute the challenge $c = H(V \parallel S)$. Afterwards, each trustee i uses his shares v_i and x_i of v and x, respectively, to create a partial response $r_i = v_i - cx_i$. Finally, when t out of n trustees collaborate they can reconstruct the response r through Lagrange interpolation. The tuple (c, r) forms a regular Schnorr signature on S, which can be verified against the public key X.

Network splitting attacks: If we enforce that at least $t' = 2f + 1$ secrets go into the final randomness then a network splitting is prevented because at least one of the good peers has to be in both groups but would participate in the generation of only one secret and thereby enforce uniqueness. Changing the number of secrets in the collective string from $n/3 + 1$ to $2/3 \cdot n + 1$ comes with a significant loss of efficiency, making the protocol even less scalable.

Scale: A core component of RandShare is public verifiable secret sharing (PVSS) scheme, producing secret inputs such that an honest threshold of participants can later recover them and form a third-party verifiable proof of their validity. But as we saw in the experimental results, it comes with a huge computational cost on each of the participants. SCRAPE [18] proposes an alternative verification technique where the cost goes from $O(n^3)$ to $O(n^2)$.
We could also model RandHound's protocol [1], whom setup arranges participants into verifiably unbiased random secret-sharing groups [1], and allowed them to drop the complexity from $O(n^3)$ to $O(cn^2)$, where c is the average (constant) size of a group.

Installation

To run the protocol, install Golang v1.8+, set your GOPATH and execute:

```
go get -u github.com/dedis/student_17_randomness
cd $GOPATH/src/github.com/dedis/student_17_randomness
randshare
```

References

[1] E. Syta, P. Jovanovic, E. Kokoris Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, B. Ford. *Scalable Bias-Resistant Distributed Randomness.* 2017

[2] B. Schoenmakers. *A simple publicly verifiable secret sharing scheme and its application to electronic voting.* 1999

[3] T. Baigneres, C. Delerablee, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain. *Trap Me If You Can−Million Dollar Curve.* 2015

[4] Z. Gutterman, B. Pinkas, and T. Reinman. *Analysis of the Linux random number generator.* 2006

[5] R. Chirgwin. *iOS 7s weak random number generator stuns kernel security.* 2014

[6] I. Bentov, A. Gabizon, and D. Zuckerman. *Bitcoin Beacon.* 2016

[7] J. Bonneau, J. Clark, and S. Goldfeder. *On Bitcoin as a public randomness source.* 2015

[8] A. K. Lenstra and B. Wesolowski. *A random zoo: sloth, unicorn, and trx.* 2015

[9] C. Blundo, A. De Santis, and U. Vaccaro. *Randomness in distribution protocols.* 1994

[10] H. Corrigan-Gibbs, W. Mu, D. Boneh, and B. Ford. *Ensuring highquality randomness in cryptographic key generation.* 2013

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

[12] Y. G. Desmedt and Y. Frankel. *Threshold cryptosystems.* 1989

[13] T. P. Pedersen. *A threshold cryptosystem without a trusted party.* 1991

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

[15] J. Clark and U. Hengartner. *On the Use of Financial Data as a Random Beacon.* 2010

[16] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. *Secure distributed key generation for discrete-log based cryptosystems.* Journal of Cryptology, 2007

[17] C. Cachin, K. Kursawe, F. Petzold, V. Shoup. *Secure and efficient asynchronous broadcast protocols.* 2001

[18] I. Cascudo and B. David. *SCRAPE: Scalable randomness attested by public entities.* 2017

[19] D. R. Stinson and R. Strobl. *Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates.* 2001