Scalable Bias-Resistant Distributed Randomness

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

*Trinity College, USA
†EPFL, Switzerland
‡University of Bonn, Germany
§Yale University, USA

Summer Research Institute, EPFL
June 19, 2017
Talk Outline

• Motivation
  ‣ The need for public randomness
  ‣ Strawman examples: Towards unbiasable randomness

• Two Randomness Protocols
  ‣ RandHound
  ‣ RandHerd

• Implementation and Experimental Results

• Demo and Conclusions
Talk Outline

• **Motivation**
  ‣ The need for public randomness
  ‣ Strawman examples: Towards unbiasable randomness

• Two Randomness Protocols
  ‣ RandHound
  ‣ RandHerd

• Implementation and Experimental Results

• Conclusions and Demo
Public Randomness

• **Different** from secret randomness
  ‣ Secret randomness used for cryptographic keys, for example

• **Collectively** used

• Unpredictable ahead of time

• Not secret past a certain point in time

• Entropy is not enough
Applications of Public Randomness

• Random selection
  ‣ lotteries, sweepstakes, jury selection, voting and election audits

• Games
  ‣ shuffled decks, team assignments

• Protocols
  ‣ parameters, IVs, nonces, sharding

• Crypto
  ‣ challenges for NZKP, authentication protocols, cut-and-choose methods, “nothing up my sleeves” numbers
Failed / Rigged Randomness

Vietnam War Lotteries (1969)

'European draws have been rigged': Ex-FIFA president Sepp Blatter claims to have seen hot and cold balls used to aid cheats

Man hacked random-number generator to rig lotteries, investigators say

New evidence shows lottery machines were rigged to produce predictable jackpot numbers on specific days of the year netting millions in winnings

Former FIFA president Sepp Blatter said he had witnessed rigged draws for European football competitions

Computer who rigged lottery number generator to produce predictable numbers a couple of times a year. Photograph: Brian Powers/MP
Public Randomness is not New

• 1955: Large table of random numbers published as a book by the Rand Corporation

• Today: Generating public random numbers is (still) hard

• Main issues: trust and scale
  ‣ Both, in generation and usage
Goals

1. Availability
   Successful protocol termination for up to \( f=t-1 \) malicious nodes.

2. Unpredictability
   Output not revealed prematurely.

3. Unbiasability
   Output distributed uniformly at random.

4. Verifiability
   Output correctness can be checked by third parties.

5. Scalability
   Executable with hundreds of participants.

Decentralized, public randomness in the \((t,n)\)-threshold security model

Assumptions: \( n=3f+1 \), Byzantine adversary and asynchronous network with eventual message delivery
Public Randomness Approaches

• **With** Trusted Third Party
  ‣ NIST Randomness Beacon

• **Without** TTP

  Unusual assumptions
  ‣ Bitcoin (Bonneau, 2015)
  ‣ Slow cryptographic hash functions (Lenstra, 2015)
  ‣ Lotteries (Baigneres, 2015)
  ‣ Financial data (Clark, 2010)

  $(t,n)$-threshold security model but not scalable
  ‣ Coin-flipping (Cachin, 2015)
  ‣ Distributed key generation (Kate, 2009)
## Public Randomness is Hard

### Strawman I
- **Idea:** Combine random inputs of all participants.
- **Problem:** Last node fully controls output.

### Strawman II
- **Idea:** Commit-then-reveal random inputs.
- **Problem:** Dishonest nodes can choose not to reveal.

### Strawman III
- **Idea:** Secret-share random inputs.
- **Problem:** Dishonest nodes can send bad shares.

<table>
<thead>
<tr>
<th>Availability</th>
<th>Unpredictability</th>
<th>Unbiasability</th>
<th>Verifiability</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawman I</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Strawman II</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Strawman III</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
### Public Randomness is Hard

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Unpredictability</th>
<th>Unbiasability</th>
<th>Verifiability</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawman I</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Strawman II</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Strawman III</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>RandShare</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

### RandShare

- **Idea:** Strawman III + *verifiable secret sharing* (Feldman, 1987)
- **Problems:**
  - Not publicly verifiable
  - Not scalable: $O(n^3)$ communication / computation complexity
Talk Outline

• Motivation
  ‣ The need for public randomness
  ‣ Strawman examples: Towards unbiasable randomness

• Two Randomness Protocols
  ‣ RandHound
  ‣ RandHerd

• Implementation and Experimental Results

• Demo and Conclusions
RandHound

• **Goals**
  ‣ Verifiability: By third parties
  ‣ Scalability: Performance better than $O(n^3)$

• **Client/server randomness scavenging protocol**
  ‣ Untrusted client uses a large set of nearly-stateless servers
  ‣ On demand (via configuration file)
  ‣ One-shot approach
RandHound

• Scenario
  ‣ Lottery authority wants to pick a winner in a fair and verifiable process

• Setup
  ‣ **Run**: announced in advance, publicly available config
  ‣ **Client**: lottery authority
  ‣ **Servers**: a set of reputable and independent parties
  ‣ **Output**: randomness + third-party proof
RandHound

Achieving Public Verifiability

• Publicly-VSS (Schoenmakers, 1999)
  ‣ Shares are encrypted and publicly verifiable through zero-knowledge proofs
  ‣ No communication between servers

• CoSi Collective signing (Syta, 2016)
  ‣ Client publicly commits to their choices

• Create protocol transcript from all sent/received (signed) messages
RandHound

Achieving Scalability

• Shard participants into constant size groups
  ‣ Secret sharing with everyone too expensive!
  ‣ Run secret sharing (only) inside groups
  ‣ Collective randomness: combination of all group outputs

Chicken-and-Egg problem?

• How to securely assign participants to groups?
RandHound

Solving the Chicken-and-Egg Problem

• Client selects server grouping

• Availability might be affected (self-DoS)

• Security properties through
  ‣ *Pigeonhole principle*: at least one group is not controlled by the adversary
  ‣ *Collective signing*: prevents client equivocation by fixing the secrets that contribute to randomness
RandHound

1. Initialization (C)
Send session config, divide servers into PVSS groups

2. Share Distribution (S)
Send encrypted PVSS shares, CoSi commits

(E_{0j}, p_{0j}) = pvss_share(Z_0), \quad V_0
3. Secret Commitment (C)
Verify PVSS shares,
CoSi challenge: client commits to secrets

4. Secret Acknowledgement (S)
Verify commitment,
send (partial) CoSi responses
5. Decryption Request (C)
Request PVSS share decryption:
(aggregate) CoSi responses

6. Share Decryption (S)
Verify CoSi response,
If ok: decrypt valid PVSS shares

$r = \sum(r_i):$ aggregate CoSi resp.
7. Recover Randomness (C)
Verify decrypted PVSS shares, compute collective randomness

Verify Randomness (anyone)
- Use a protocol log (transcript) L to verify randomness Z
- Replay and check all steps
- Accept if all correct
Public Randomness is (not so) Hard

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Unpredictability</th>
<th>Unbiasability</th>
<th>Verifiability</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawman I</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Strawman II</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Strawman III</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RandShare</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>RandHound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Communication / computation complexity: $O(c^2n)$
RandHerd

• Goals
  ‣ Continuous, leader-coordinated randomness generation
  ‣ Small randomness proof size (a single Schnorr signature)
  ‣ Better performance than $O(n)$

• Decentralized randomness beacon
  ‣ Built as a collective authority or cothority
  ‣ Randomness on demand, at frequent intervals, or both
RandHerd

Achieving RandHerd’s Goals

• Idea
  ‣ Collective randomness = collective Schnorr signature
  ‣ Benefits: Small proofs, $O(\log n)$ complexity

• Problem
  ‣ Failing nodes influence output!
  ‣ If some nodes unavailable, then the signature not a function of everyone’s input
RandHerd

Achieving RandHerd’s Goals

• Solution
  ‣ Arrange nodes into $(t,n)$-threshold Schnorr signing (Stinson, 2001) groups (failure resistance)
  ‣ Collective randomness = aggregate group signatures
  ‣ Approach: Setup + round function
RandHerd Setup

- **Goal**: secure prep for RandHerd Round
- Executed once followed by many rounds of randomness
- Consists of 4 steps
1. Leader Election
Elect a temporary leader via lowest ticket
\[ t_i = VRF(config, key_i) \]

2. Sharding
Run RandHound to produce (Z,L) as sharding seed
RandHerd Setup

3. Group Setup
Create TSS groups using Z and generate group keys $X_i$

4. Collective RandHerd Key
Certify aggregate public key $X$ using CoSi

X = $X_0X_1X_2$
RandHerd Round

Randomness Generation

1. Cothority Leader (CL) broadcasts timestamp $v$

2. TSS-CoSi
   a. Produce group Schnorr signatures $(c, r_0)$ $(c, r_1)$ $(c, r_2)$ on $v$
   b. At least $2f+1$ nodes fix and certify challenge $c$ using CoSi
   c. Aggregate into collective Schnorr signature $(c, r = r_0 + r_1 + r_2)$
   d. Publish $(c, r)$ as collective randomness
RandHerd Round

Randomness Verification

1. RandHerd produces a simple Schnorr signature

2. Anyone can efficiently verify \((c, r)\) on \(v\) using the collective public key \(X = X_0X_1X_2\)

3. Single signature verification!
Public Randomness is (not so) Hard

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Unpredictability</th>
<th>Unbiasability</th>
<th>Verifiability</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawman I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strawman II</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strawman III</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RandShare</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RandHound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RandHerd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Communication / computation complexity: $O(c^2 \log(n))$
Talk Outline

• Motivation
  ‣ The need for public randomness
  ‣ Strawman examples: Towards unbiasable randomness

• Two Randomness Protocols
  ‣ RandHound
  ‣ RandHerd

• Implementation and Experimental Results

• Demo and Conclusions
Implementation & Experiments

**Implementation**

- Go versions of DLEQ-proofs, PVSS, TSS, CoSi-TSS, RandHound, RandHerd
- Based on DEDIS code
  - Crypto library
  - Network library
  - Cothority framework
- [https://github.com/dedis](https://github.com/dedis)

**DeterLab Setup**

- 32 physical machines
  - Intel Xeon E5-2650 v4 (24 cores @ 2.2 GHz)
  - 64 GB RAM
  - 10 Gbps network link
- Network restrictions
  - 100 Mbps bandwidth
  - 200 ms round-trip latency
Experimental Results – RandHound

Randomness verification and generation time
Experimental Results – RandHound

Take-away: In a RandHound run with 1024 nodes and group size 32, generation takes 290 sec and verification takes 160 sec.

Randomness verification and generation time
Experimental Results – RandHound

CPU cost for the client and the servers
Experimental Results – RandHound

Take-away: Total cost for 1 RandHound run is 10 CPU min (EC2: < $0.02) with 1024 nodes, group size 32.

CPU cost for the client and the servers
Experimental Results – RandHerd

Randomness generation time
Experimental Results – RandHerd

**Take-away:** Generation time for 1 RandHerd run with is 6 sec, after setup (10 mins) with 1024 nodes, group size 32.
Talk Outline

• Motivation
  ‣ The need for public randomness
  ‣ Strawman examples: Towards unbiasable randomness

• Two Randomness Protocols
  ‣ RandHound
  ‣ RandHerd

• Implementation and Experimental Results

• Demo and Conclusions
Demo

pulsar.dedis.ch
Conclusion

• Generation of public randomness: **trust** and **scale** issues

• Our solution: two protocols in the \((t,n)\)-threshold security model

<table>
<thead>
<tr>
<th></th>
<th>Availability</th>
<th>Unpredictability</th>
<th>Unbiasability</th>
<th>Verifiability</th>
<th>Scalability</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandHound</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>O(n)</td>
</tr>
<tr>
<td>RandHerd</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>O(log(n))</td>
</tr>
</tbody>
</table>

• Code: [https://github.com/dedis/cothority](https://github.com/dedis/cothority)
Thank you!

Questions?

Ewa Syta
ewa.syta@trincoll.edu