Poster: RandGener: Distributed Randomness Beacon from Verifiable Delay Function

Arup Mondal
Ashoka University
Sonipat, Haryana, India
arup.mondal_phd19@ashoka.edu.in

Ruthu Hulikal Rooparaghunath
Vrije Universiteit
Amsterdam, The Netherlands
r.rooparaghunath@student.vu.nl

Debayan Gupta
Ashoka University
Sonipat, Haryana, India
debayan.gupta@ashoka.edu.in

***Abstract*—Buoyed by the excitement around secure decentralized applications, the last few decades have seen numerous constructions of distributed randomness beacons (DRB) along with use cases; however, a secure DRB (in many variations) remains an open problem. We further note that it is natural to want some kind of reward for participants who spend time and energy evaluating the randomness beacon value – this is already common in distributed protocols.**

In this work, we present RandGener, a *novel* n-party *commit-reveal-recover* (or *collaborative*) DRB protocol with a novel *reward and penalty mechanism* along with a set of realistic guarantees. We design our protocol using trapdoor watermarkable verifiable delay functions in the RSA group setting (without requiring a trusted dealer or distributed key generation).

***Index Terms*—Randomness Beacon, Verifiable Delay Function.**

I. Introduction

A *randomness beacon* [1] is an ideal functionality that continuously publishes independent random values which no party can predict or manipulate; critically, this value must be efficiently verifiable by anyone. A *Distributed Randomness Beacon* (DRB) protocol allows a set of participants to jointly compute a continuous stream of randomness beacon outputs. A secure DRB protocol should satisfy the following properties, outlined in [2], [3], [4]:

(1) *Liveness/availability*: participants should not be able to prevent the progress of random beacon computation,
(2) *Guaranteed output delivery*: adversaries should not be able to prevent honest participants in the protocol from obtaining a random beacon output,
(3) *Bias-resistance*: no participants should be able to influence future random beacon values to their advantage,
(4) *Public verifiability*: as soon as a random beacon value is generated, it can be verified by anyone independently using only public information, and
(5) *Unpredictability*: participants should not be able to predict the future random beacon values.

We introduce two new desirable properties for DRB protocols: (6) a ***reward mechanism***, which incentivizes participants who invest time and energy in evaluating the randomness beacon value by rewarding their effort, and (7) a ***penalty mechanism***, which discourages inadequate participation, incorrect information or cheating by applying penalties for participants who engage in those actions.

Our n-party distributed randomness beacon protocol, RandGener demonstrates a method of claiming "ownership" of a randomness beacon value evaluation in each round of the protocol's execution. This is done by attaching a "watermark" of computing participants to the result of the evaluation in order to reward corresponding participants for their contribution.

Our contributions are summarised as follows:

- We extend watermarkable VDF (wVDF) defined in [5] by formally defining a new type called ***trapdoor*** wVDF. Furthermore, we demonstrate a construction using Wesolowski [5] and Pietrzak's [6] scheme.
- We construct RandGener, an efficient n-party *commit-reveal-recover* (or *collaborative*) distributed randomness beacon protocol with a novel ***reward mechanism*** and ***penalty mechanism*** using a trapdoor wVDF. Our protocol does not require any trusted (or expensive) setup and proves that it provides the desired security properties.

Brief Relevant Work: A *commit-reveal* is a classic approach proposed in [1]. First, all participants publish a commitment $y_i = \mathsf{Commit}(x_i)$ to a random value x_i. Next, participants reveal their x_i values, resulting in $R = \mathsf{Combine}(x_1, \ldots, x_n)$ for some suitable combination function (such as an exclusive-or or a cryptographic hash).

However, the output can be biased by the last participant to open their commitment (referred to as a *last-revealer attack*), since the last participant, by knowing all other commitments x_i, can compute R early.

A very different approach to constructing DRBs uses time-sensitive cryptography (TSC), specifically using delay functions to prevent manipulation. The simplest example is Unicorn [4], a one-round protocol in which participants directly publish (within a fixed time window) a random input x_i. The result is computed as $R = \mathsf{TSC}(\mathsf{Combine}(x_1, \ldots, x_n))$. However, the downside of the Unicorn [4] is that a delay function must be computed for every run of the protocol. Recently, Choi et al. [3] introduced the Bicorn family of DRB protocols, which retain the advantages of Unicorn [4] while enabling efficient computation of the result (with no delay) if all participants act honestly. Yet, as stated in [3], all Bicorn variants come with a fundamental security caveat, i.e., the *last revealer prediction attack*: if participant P_i withholds their x_i value, but all others publish, then participant P_i will be able to simulate efficiently and learn R quickly (*optimistic case*), while honest participants will need to execute the force open and compute the delay function to complete before learning R (*pessimistic case*). Similarly, a coalition of malicious participants can share their x values and privately compute R. Nevertheless, *none* of the existing delay-cryptography-based commit-reveal-recover style DRB protocols provide a reward/penalty mechanism to regulate the behaviour of corrupted participants. In this work, we propose an efficient n-party commit-reveal-recover (or collaborative) DRB protocol with a novel reward and penalty mechanism based on the trapdoor wVDF.

TABLE I: Comparison collaborative DRB schemes.

Paper	Crypto Primit.	Comp. Cost	Comm. Cost	Fault Toler.	Crypto Model			Features								
					Network Model	Setup Assumption	Trusted Setup Req.	Adaptive Adversary	Liveness/Availability	Bias Resistance	Fairness	GOD	Scaleability	Unpredictability	**Reward**	**Penalty**
[4]	Sloth	$O(n)$	$O(1)$	$\frac{(n-1)}{n}$	■	◐	✓	×	✓	✓	✓	✓	×	✓	×	×
[2]	tVDF	$O(n^2)$	$O(1)$	$\frac{n}{2}$	□	◐	×	✓	✓	✓	✓	✓	✓	×	×	×
[3]	VDF	$O(n^2)$	$O(1)$	$n-1$	⊠	◐	×	✓	✓	✓	✓	✓	✓	✓	×	×
Ours	twVDF	$O(n^2)$	$O(1)$	$n-1$	⊠	◐	×	✓	✓	✓	✓	✓	✓	✓	✓	✓

□ denotes the "asynchronous" network model; ⊠ denotes the "partial synchronous" network model; ■ denotes the "synchronous" network model. ◐ denotes the "Common Reference String" setup assumption; ✓ denotes provide the property; × denotes does not provide the property. GOD – Guaranteed Output Delivery.

II. Technical Preliminaries

Basic Notation: Given a set $\mathcal{X}$, we denote $x \xleftarrow{\$} \mathcal{X}$ as the process of sampling a value x from the uniform distribution on $\mathcal{X}$. Supp($\mathcal{X}$) denotes the support of the distribution $\mathcal{X}$.

We denote the security parameter by $\lambda \in \mathbb{N}$. A function $\texttt{negl}\colon \mathbb{N} \to \mathbb{R}$ is negligible if it is asymptotically smaller than any inverse-polynomial function. Namely, for every constant $\epsilon > 0$ there exists an integer N_ϵ for all $\lambda > N_\epsilon$ such that $\texttt{negl}(\lambda) \leq \lambda^{-\epsilon}$.

Number Theory: We assume that $N = p \cdot q$ is the product of two large secret and *safe* primes and $p \neq q$. We say that N is a strong composite integer if $p = 2p' + 1$ and $q = 2q' + 1$ are safe primes, where p' and q' are also prime. We say that $\mathbb{Z}_N$ consists of all integers in $[N]$ that are relatively prime to N (i.e., $\mathbb{Z}_N = \{x \in \mathbb{Z}_N : \texttt{gcd}(x, N) = 1\}$).

Repeated Squaring Assumption: The repeated squaring assumption [6] roughly states that there is no parallel algorithm that can perform T squarings modulo an integer N significantly faster than just doing so sequentially, assuming that N cannot be factored efficiently, or in other words $\texttt{RSW}$ assumption implies that factoring is hard. More formally, no adversary can factor an integer $N = p \cdot q$ where p and q are large secret and "safe" primes [6]. A repeated squaring $\texttt{RSW} = (\texttt{Setup}, \texttt{Sample}, \texttt{Eval})$ is defined below. Moreover, we define a trapdoor evaluation $\texttt{RSW.tdEval}$ (to enable *fast* repeated squaring evaluation), from which we can derive an actual output using trapdoor in $\mathsf{poly}(\lambda)$ time.

• $N \leftarrow \texttt{RSW.Setup}(\lambda)$: Output $\mathsf{pp} = (N)$ where $N = p \cdot q$ as the product of two large (λ-bit) randomly chosen secret and safe primes p and q.
• $x \leftarrow \texttt{RSW.Sample}(\mathsf{pp})$: Sample a random instance x.
• $y \leftarrow \texttt{RSW.Eval}(\mathsf{pp}, T, x)$: Output $y = x^{2^T} \mod N$ by computing the T sequential repeated squaring from x.
• $y \leftarrow \texttt{RSW.tdEval}(\mathsf{pp}, \mathsf{sp} = \phi(N), x)$: To compute $y = x^{2^T} \mod N$ *efficiently* using the trapdoor as follows: ***(i)*** Compute $v = 2^T \mod \phi(N)$. **Note:** $(2^T \mod \phi(N)) \ll 2^T$ for large T. ***(ii)*** Compute $y = x^v \mod N$. **Note:** $x^{2^T} \equiv x^{(2^T \mod \phi(N))} \equiv x^v \pmod N$.

III. Verifiable Delay Function

A *verifiable delay function* (VDF), introduced by Boneh et al. [7], is a special type of delay function f characterized by a time-bound parameter T and the following three properties: ***(i)*** *T-sequential function*: The function f can be evaluated in sequential time T, but it should not be possible to evaluate f significantly faster than T even with parallel processing. ***(ii)*** *Unique output*: The function f produces a unique output, which is efficiently and publicly ***(iii)*** *Verifiable* (in time that is essentially independent of T) - meaning that the function f should produce a proof π which convinces a verifier that the function output has been correctly computed.

Wesolowski [5] first describes a trapdoor VDF (tVDF) as a modified and extended version of traditional VDFs [7] such that the $\texttt{Setup}$ algorithm, in addition to the public parameters pp, outputs a trapdoor or secret parameter sp to the party invoking the $\texttt{Setup}$ algorithm. This parameter sp is kept secret by the invoker, whereas pp is published. Furthermore, using the trapdoor evaluation $\texttt{tdEval}$ and trapdoor proof generation $\texttt{tdProve}$ (by enabling *fast* computations), the secret parameter-holding participants can derive an actual output and the proof of correctness in $\mathsf{poly}(\lambda)$ time. Parties without knowledge of the trapdoor, as in the traditional VDF case, can still compute the output and proof of correctness by executing $\texttt{Eval}$ and $\texttt{Prove}$. However, it requires T-sequential steps to do so. For the purpose of our distributed random beacon protocol, we require and define a trapdoor watermarkable VDF (twVDF). In this case, we use the same trapdoor evaluation $\texttt{tdEval}$, but we generate a watermarked proof of correctness using $\texttt{tdProve}$ by embedding a watermark of the evaluator.

In Algorithm 1, we provide details for the formal construction of watermarkable verifiable delay function $\texttt{VDF}$ using Wesolowski [5] and Pietrzak's [6] scheme, consisting of algorithms ($\texttt{Setup}$, $\texttt{Sample}$, $\texttt{Eval}$, $\texttt{Prove}$, $\texttt{Verify}$) with a trapdoor watermarkable VDF evaluation $\texttt{tdEval}$ and proof generation $\texttt{tdProve}$.

Algorithm 1: Trapdoor Watermarkable VDF using Wesolowski and Pietrzak

```
• VDF.Setup($\lambda$)
    1) Call and generate $N \leftarrow \texttt{RSW.Setup}(\lambda)$
    2) A cryptographically secure $\lambda$-bit hash function $\mathcal{H}_{\mathsf{prime}}$ or $\mathcal{H}_{\mathsf{random}}$.
    3) Generate a time-bound parameter $T$.
    4) return $\mathsf{pp} = (N, T, \mathcal{H})$.
  • VDF.Sample($\mathsf{pp}$)
    1) Generate an input $x \in \mathbb{Z}_N^* \leftarrow \texttt{RSW.Sample}(\mathsf{pp})$
  • VDF.Eval($\mathsf{pp}, x$)
    1) Compute $y = x^{2^T} \mod N \in \mathbb{Z}_N^*$ using $\texttt{RSW.Eval}(\mathsf{pp}, T, x)$
    2) Generate an advice string $\alpha$.
```

```
3) **return** $(y, \alpha)$.
• VDF.tdEval(pp, $x$)
1) Compute group order $\phi(N) = (p-1) \cdot (q-1)$ using trapdoor $(p, q)$.
2) Compute $y = x^{2^T} \mod N$ using RSW.tdEval(pp, sp $= \phi(N), x$).
3) Generate an advice string $\alpha$.
4) **return** $(y, \alpha)$.

.................... Using Wesolowski's [5] Scheme ....................
• VDF.Prove(pp, $x, \mu, y, \alpha, T$)
1) Generate a prime $l = \mathcal{H}_{\text{prime}}(x \parallel y \parallel \mu)$ [a]
2) Compute the proof $\pi_\mu = x^{\lfloor 2^T / l \rfloor} \pmod N$ [b] and **return** $\pi_\mu$.
• VDF.tdProve(pp, $x, \mu, y, \alpha, T$)
1) Generate a prime $l = \mathcal{H}_{\text{prime}}(x \parallel y \parallel \mu)$
2) Compute group order $\phi(N) = (p-1) \cdot (q-1)$ using trapdoor $(p, q)$.
3) Compute proof $\pi_\mu = x^{(\lfloor 2^T / l \rfloor \mod \phi(N))} \pmod N$ and **return** $\pi_\mu$.
• VDF.Verify(pp, $x, \mu, y, \pi_\mu, T$)
1) Generate a prime $l = \mathcal{H}_{\text{prime}}(x \parallel y \parallel \mu)$
2) $r = 2^T \mod l$
3) **return accept if** $(\pi_\mu^l \cdot x^r) \mod N = y$, otherwise reject

...................... Using Pietrzak's [6] Scheme ......................
• VDF.Prove(pp, $x, \mu, y, \alpha, T$)
1) Compute $u = x^{2^{T/2}} \mod N$
2) Generate a random $r = \mathcal{H}_{\text{random}}(x \parallel T/2 \parallel y \parallel u \parallel \mu)$ [c]
3) Compute $x = x^r \cdot u \mod N$ and $y = u^r \cdot y \mod N$
4) Proof $\pi_\mu = u \cup$ VDF.Prove(pp, $x, \mu, y, \alpha, T/2$) and **return** $\pi_\mu$.
• VDF.tdProve(pp, $x, \mu, y, \alpha, T$)
1) Compute group order $\phi(N) = (p-1) \cdot (q-1)$ using trapdoor $(p, q)$.
2) Compute $u = x^{(2^{T/2} \mod \phi(N))} \mod N$
3) Generate a random $r = \mathcal{H}_{\text{random}}(x \parallel T/2 \parallel y \parallel u \parallel \mu)$
4) Compute $x = x^{(r \mod \phi(N))} \cdot u \mod N$ and $y = u^{(r \mod \phi(N))} \cdot y \mod N$
5) Proof $\pi_\mu = u \cup$ VDF.tdProve(pp, $x, \mu, y, \alpha, T/2$) and **return** $\pi_\mu$.
• VDF.Verify(pp, $x, \mu, y, \pi_\mu, T$)
1) Generate a random $r = \mathcal{H}_{\text{random}}(x \parallel T/2 \parallel y \parallel u \parallel \mu)$
2) Compute $x = x^r \cdot u \mod N$ and $y = u^r \cdot y \mod N$
3) Call VDF.Verify(pp, $x, \mu, y, \pi_\mu, T/2$)
4) **return** accept if $T = 1$ check $y = x^2 \mod N$, otherwise reject.
```

[a]Sampled uniformly from Prime(λ)
[b]μ is an evaluator's watermark
[c]Sampled uniformly from $\{1, 2, \ldots, 2^\lambda\}$

IV. RandGener Protocol Design

In this section, we present our RandGener protocol, a n-party distributed randomness beacon protocol DRB = (Setup, VerifySetup, Gen). The construction details are in Algorithm 2 using our trapdoor watermarkable VDF.

Algorithm 2: RandGener: Distributed Randomness Beacon Protocol

Input: A globally agreed security parameter λ, a set of participants $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, a set of public parameters $\mathcal{PP} = \{\mathsf{pp}_1, \mathsf{pp}_2, \ldots, \mathsf{pp}_n\}$, a time-bound parameter T, an initial random beacon value R_0 (it becomes available to all parties running the protocol after the setup is completed at approximately the same time), and two cryptographically secure λ-bit hash functions: (i) $\mathcal{H}_{\text{randToinput}}$ – mapping a random value to the input space of the VDF, and (ii) $\mathcal{H}_{\text{inputTorand}}$ – mapping a VDF output to a random value.
Output: The randomness beacon value $R_1, R_2, \ldots, R_\infty$ for that round of the protocol.

```
• DRB.Setup($\lambda$)
1) $\forall i$ $P_i \in \mathcal{P}$ locally generate a public parameter $\mathsf{pp}_i = (N_i, T, \mathcal{H}) =$ VDF.Setup($\lambda$).
2) $\forall i$ $P_i \in \mathcal{P}$ run the zero-knowledge protocol for proving that a known $N_i$ is
   the product of two safe primes and the protocol "proving the knowledge of a
   discrete logarithm that lies in a given range" to show that the prime factors
   $p_i$ and $q_i$ are $\lambda$-bits each. Let $\pi_{N_i}$ denote the resulting proof obtained by
   running both protocols non-interactively using the Fiat-Shamir heuristic.
3) **Broadcast** ($\mathcal{PP} = \{\mathsf{pp}_1, \ldots, \mathsf{pp}_n\}$, $\Pi = \{\pi_{N_1}, \ldots, \pi_{N_n}\}$).
• DRB.VerifySetup($\mathcal{PP}, \Pi$)
1) For each public parameter $\mathsf{pp}_i \in \mathcal{PP}$ and a corresponding proof $\pi_{N_i} \in \Pi$,
   **return** accept if the validity of $\mathsf{pp}_i$ can be successfully checked by using
   the verification procedures corresponding to the proof techniques used in
   DRB.Setup algorithm as specified in [2], otherwise **return** reject.
• DRB.Gen($\mathcal{PP}, T, R_0$)
1) Set $r \leftarrow 1$.
   ...................... Commit ...................... **deadline** $T_0$
2) Compute $x_r \leftarrow \mathcal{H}_{\text{randToinput}}(R_{r-1})$.
3) Generate a random input $x'_{r,i}$
4) Compute and publish $x_{r,i} \leftarrow \mathcal{H}(x'_{r,i} || x'_r)$          ▷ Broadcast
   ...................... Reveal ...................... **deadline** $T_1$
5) For each participant $P_i \in \mathcal{P}$ in parallel
   a) Compute $(y_{r,i}, \alpha_{r,i}) \leftarrow$ VDF.tdEval($\mathsf{pp}_i, x_{r,i}, T$).
   b) Compute $\pi_{r,i} \leftarrow$ VDF.tdProve($\mathsf{pp}_i, x_{r,i}, \mu_i, y_{r,i}, \alpha_{r,i}, T$).
   c) Publish $(y_{r,i}, \pi_{r,i})$.          ▷ Broadcast
   ............................ Finalize ............................
6) For each participant $P_i \in \mathcal{P}$, verify $(x_r, y_{r,i}, \pi_{r,i})$
   a) If VDF.Verify($\mathsf{pp}_i, x_{r,i}, y_{r,i}, \pi_{r,i}, T$) = reject or $x_{r,i}$ was not
      published by $T_1$, then remove participant $P_i$ and add $\tilde{\mathcal{P}} \leftarrow \tilde{\mathcal{P}} \cup P_i$.
7) For **all** $P_i \in \mathcal{P}$, If VDF.Verify($\mathsf{pp}_i, x_{r,i}, y_{r,i}, \pi_{r,i}, T$) = accept
   a) Compute $y_r = \prod_{P_i \in \mathcal{P}} y_{r,i}$          ▷ **Optimistic case**
   b) Optionally, a proof $\pi_{y_r}$ can be compute to enable verification of $y_r$.
   ............................ Recover ............................
8) For each participant $P_j \in \tilde{\mathcal{P}}$ in parallel          ▷ **Pessimistic case**
   a) Compute $(y_{r,j}, \alpha_{r,j}) \leftarrow$ VDF.Eval($\mathsf{pp}_j, x_{r,j}, T$).
   b) Compute $\pi_{r,j} \leftarrow$ VDF.Prove($\mathsf{pp}_j, x_{r,j}, \mu_r, y_{r,j}, \alpha_{r,j}, T$).
   c) Compute $y_r = \prod_{P_i \in \mathcal{P}} y_{r,i} \cdot \prod_{P_j \in \tilde{\mathcal{P}}} y_{r,j}$
   d) Optionally, a proof $\pi_{y_r}$ can be computed to enable verification of $y_r$.
   ..................................................................
9) Output the $r$-th round's randomness beacon $R_r \leftarrow \mathcal{H}_{\text{inputTorand}}(y_r)$
10) **Reward the participants computing the $r$-th round's randomness beacon
    value $\mathcal{P} \setminus \tilde{\mathcal{P}}$ and apply a Penalty to participants $\tilde{\mathcal{P}}$.**
11) Set $r \leftarrow r + 1$.
12) Repeat from step 2 to step 11 – to generate the next round's randomness.
```

Theorem IV.1. *Assuming that $\mathcal{H}_{\text{randToinput}}$ and $\mathcal{H}_{\text{inputTorand}}$ is the random oracle and* VDF *is a trapdoor watermarkable VDF, then it holds that* Algorithm 2 *is a DRB scheme.*

The proof of Theorem IV.1 is deferred to the full version.

V. Future Work

Existing collaborative DRB protocols experience challenges in inefficient communication complexity, which limits their scalability.

In the near future, we hope to construct a complexity-efficient collaborative DRB protocol.

References

[1] M. Blum, "Coin flipping by telephone a protocol for solving impossible problems," *SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983. [Online]. Available: https://doi.org/10.1145/1008908.1008911

[2] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. R. Weippl, "Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness," 2020, p. 942. [Online]. Available: https://eprint.iacr.org/2020/942

[3] K. Choi, A. Arun, N. Tyagi, and J. Bonneau, "Bicorn: An optimistically efficient distributed randomness beacon," *IACR Cryptol. ePrint Arch.*, p. 221, 2023. [Online]. Available: https://eprint.iacr.org/2023/221

[4] A. K. Lenstra and B. Wesolowski, "A random zoo: sloth, unicorn, and trx," *IACR Cryptol. ePrint Arch.*, p. 366, 2015. [Online]. Available: http://eprint.iacr.org/2015/366

[5] B. Wesolowski, "Efficient verifiable delay functions," 2018, p. 623. [Online]. Available: https://eprint.iacr.org/2018/623

[6] K. Pietrzak, "Simple verifiable delay functions," *IACR Cryptol. ePrint Arch.*, 2018. [Online]. Available: https://eprint.iacr.org/2018/627

[7] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 601, 2018. [Online]. Available: https://eprint.iacr.org/2018/601