

# RANDCHAIN: Decentralised Randomness Beacon from Sequential Proof-of-Work

Runchao Han<sup>\*†</sup>, Jiangshan Yu<sup>\*¶</sup>, Haoyu Lin<sup>‡§</sup>

<sup>\*</sup>Monash University, {runchao.han, jiangshan.yu}@monash.edu

<sup>†</sup>CSIRO-Data61

<sup>‡</sup>Bytom Foundation, chris.haoyul@gmail.com

<sup>§</sup>ZenGo X

**Abstract**— Decentralised Randomness Beacon (DRB) is a service that periodically generates publicly verifiable randomness, which plays critical roles in various cryptographic protocols. However, constructing DRB protocols is challenging. Existing DRB protocols suffer from either strong network synchrony assumptions, high communication complexity or attacks.

In this paper, we propose RANDCHAIN, a new family of permissioned DRB protocols. RANDCHAIN is constructed from Sequential Proof-of-Work (SeqPoW) – a Proof-of-Work (PoW) variant that is sequential, i.e., non-parallelisable – and Nakamoto consensus. In RANDCHAIN, nodes jointly maintain a blockchain, and each block attaches a random output. Given the last block, each node deterministically derives a SeqPoW puzzle, and keeps solving SeqPoW (aka. mining) until finding a valid solution, of which the value is unpredictable. The first node solving SeqPoW includes a block to the blockchain. The SeqPoW solution’s hash is the random output in this block. RANDCHAIN applies Nakamoto consensus so that nodes agree on a unique blockchain.

We formalise SeqPoW, propose two constructions, prove their security and evaluate their performance. Our results show that SeqPoW is practical. We then define DRBs and analyse RANDCHAIN’s security. Our analysis shows that RANDCHAIN implements a DRB protocol and can produce strongly unpredictable randomness. In addition, while as simple and scalable as PoW-based consensus, RANDCHAIN achieves better energy efficiency and decentralisation than PoW-based consensus, thanks to the non-parallelisable mining.

## 1. INTRODUCTION

Randomness beacon (RB) is a service that periodically generates publicly verifiable randomness – an essential building block for various cryptographic primitives. RB is not always reliable: malicious RBs may generate biased randomness to compromise the security and/or fairness of randomness-based cryptographic primitives. To make RB trustworthy, a promising approach is the decentralised randomness beacon (DRB). In a DRB, a group of participants (aka. nodes) generate randomness that they agree on. No party can fully control the generated randomness, so biasing or predicting the randomness can be hard. DRBs have been integrated into various protocols, especially blockchains [1]–[4].

However, constructing secure and scalable DRBs is challenging, mainly due to four obstacles. First, DRBs usually execute in rounds, leading to the *lock-step synchrony* assumption, where all messages are delivered at the end of each round. *Lock-step synchrony* is a strong assumption, and is considered

unrealistic in real-world networks with delays. Second, DRBs usually suffer from high communication overhead – usually no less than  $O(n^2)$  where  $n$  is the number of nodes. Third, some DRBs can be vulnerable [2], [5]–[7], including being biasible, i.e., the adversary can bias the randomness to its preferred values, and being predictable, i.e., the adversary can predict randomness in the future. Last, some DRBs [6]–[10] rely on trustworthy components such as full-fledged blockchains and smart contracts, which are not always realistic.

In this paper, we propose RANDCHAIN – a new family of DRB protocols constructed from Sequential Proof-of-Work (SeqPoW) – a Proof-of-Work (PoW) [11] variant that is non-parallelisable. In RANDCHAIN, nodes jointly maintain a *blockchain*, i.e., a chain of blocks, and each block consists of a random output. Given the blockchain, each participant keeps solving a SeqPoW puzzle derived from the last block. The first participant solving the SeqPoW puzzle proposes a block with the next random output, of which the value is decided by the SeqPoW solution. RANDCHAIN employs Nakamoto consensus [12] so that nodes agree on the longest blockchain.

While inheriting PoW-based consensus’ simplicity, security and scalability, RANDCHAIN is also energy efficient and decentralised. As SeqPoW is non-parallelisable, each node can only use a single processor for SeqPoW mining. This prevents mining from marketisation and thus keeps RANDCHAIN energy efficient, unlike PoW-based blockchains where nodes invest high-end mining hardware with massively parallel processors. In addition, SeqPoW mining can only be accelerated using processors with high clock rate, which is hard to improve further, given the voltage limit of processors [13]. Thus, powerful nodes cannot obtain much speedup on mining, leading to high degree of mining power decentralisation. Concretely, we make the following contributions.

**Sequential Proof-of-Work (SeqPoW).** We propose SeqPoW, a PoW [11] variant that is *non-parallelisable*. To solve a PoW puzzle, the prover searches for a random string (aka. nonce) that satisfies a difficulty parameter. The search is done by enumerating all possible nonces, which can be parallelised using multiple processors. Meanwhile in a SeqPoW, the prover takes a predefined nonce, and executes an *iteratively sequential function* over this nonce. The prover keeps incrementing the iteratively sequential function until finding an output that

<sup>¶</sup> Corresponding author.

satisfies the difficulty parameter. In this way, one cannot accelerate solving SeqPoW by parallelism. Like VDFs, the only way of accelerating solving SeqPoW is to use a processor with high clock rate, which gives little optimisation space.

We formalise SeqPoW and propose two SeqPoW constructions, one is from Verifiable Delay Functions (VDFs) [14] and the other is from the Sloth hash function [15]. We formally analyse the security and efficiency of both SeqPoW constructions. We also implement them and evaluate their performance. Our results show that SeqPoW from Pietrzak’s VDF [16] exhibits the best candidate, given the efficient proof generation and verification and the acceptable proof size.

Of independent interest, SeqPoW is a powerful primitive and can be applied to various protocols. We discuss potential applications of SeqPoW, including leader election and Proof-of-Stake (PoS)-based consensus.

**RANDCHAIN– DRB from SeqPoW and Nakamoto consensus.** We then construct RANDCHAIN– a new family of DRBs that is simple, secure, scalable, energy efficient and decentralised in the meantime. In RANDCHAIN, nodes jointly maintain a *blockchain*, each of which includes a random output. Nodes continuously work on SeqPoW *mining*, where a participant can append a new block only after solving a SeqPoW puzzle. Each participant derives its SeqPoW input from the precedent block and its identity in the system, and cannot choose its own SeqPoW input freely. Thus, given the local view of the blockchain, the participant has only one valid SeqPoW input, and cannot accelerate SeqPoW mining by parallelism. RANDCHAIN implements Nakamoto consensus, i.e., the longest-chain rule where nodes agree on the longest blockchain, so that nodes have a consistent view on the blockchain.

We present the detailed construction of RANDCHAIN and analyse its security. We define DRBs with four properties, namely *consistency*, *liveness*, *uniform-distribution* and *unpredictability*. *Consistency* and *liveness* follow the definitions in Nakamoto consensus [12]. *Uniform-distribution* requires each output is pseudorandom, i.e., uniformly distributed. *Unpredictability* requires the adversary cannot predict random outputs either generated by other nodes or by itself. Our analysis shows that RANDCHAIN implements a DRB protocol with strong unpredictability guarantee. In addition, we show that while inheriting PoW-based consensus’ simplicity, security and scalability, RANDCHAIN is energy efficient and decentralised. We also analyse several practical considerations in RANDCHAIN.

Compared to existing DRBs, RANDCHAIN makes three unique design choices. First, nodes in RANDCHAIN are *competitive*, i.e, each node tries to produce the next random output before other nodes, whereas nodes in existing DRBs are usually *collaborative*, i.e., nodes contribute their local random inputs and combine them to a unique one. Second, RANDCHAIN reuses random entropy from bootstrap, whereas nodes in existing DRBs usually sample new random entropy for every round. Last, RANDCHAIN uses uncontrollable entropy such as nodes’ Byzantine behaviours and network delay,

TABLE I: Summary of notations. The first section includes frequently used notations. The second and the last sections include notations for Sequential Proof-of-Work and RANDCHAIN, respectively.

Notation	Description
$H(\cdot), H'(\cdot)$	Hash functions mapping $\{0, 1\}^*$ to $\{0, 1\}^k$
$\mathcal{X}, \mathcal{Y}$	Domains
$pp$	Public parameter
$t$	Time parameter of VDFs ( $t \in \mathbb{N}^+$ )
$\pi$	Proof
$\psi$	Step parameter of SeqPoW ( $\psi \in \mathbb{N}^+$ )
$T$	Difficulty parameter of (Seq)PoW ( $T \in (1, \infty)$ )
$sk, pk$	Secret key and public key
$G, g$	Cyclic group and its generator
$H_G(\cdot)$	Hash function mapping $\{0, 1\}^*$ to an element on $G$
$S_i, b_i$	The $i$ -th SeqPoW solution and its validity
$\pi_i$	Proof of $i$ -th SeqPoW solution
$\{p_1, \dots, p_n\}$	Nodes in RANDCHAIN
$sk_k, pk_k$	Secret key and public key of node $p_k$
$C_k$	The local blockchain of node $p_k$
$MC_k$	The the main chain of node $p_k$
$B_\ell$	The $\ell$ -th block
$\alpha, \beta \in (0, 1)$	Mining power portion of Byzantine and correct nodes

whereas existing DRBs usually allow nodes to sample their own entropy.

**Paper organisation.** Section 2 describes the background. Section 3 introduces SeqPoW, including definitions, constructions and analysis. Section 4-5 describe the RANDCHAIN construction and security analysis, respectively. Section 6 discusses other concerns in RANDCHAIN. Section 7 compares RANDCHAIN with existing RBs. Section 8 concludes this paper and discusses future work. We defer some security proofs of our SeqPoW constructions to Appendix A.

## 2. PRELIMINARIES

**Notations.** Table I summarises notations in this paper.

**PoW and PoW-based consensus.** Proof-of-Work (PoW) can be seen as a computationally intensive puzzle. To solve a PoW puzzle, a prover should finish an amount of computation. Given a PoW’s solution, the verifier can verify whether the prover has finished that amount of computation with negligible overhead. PoW can be constructed from hash functions. As shown in Figure 1, the prover needs to find a nonce  $x$  which makes  $H(in||x) \leq \frac{2^c}{T}$ , where  $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$  is a cryptographic hash function,  $in$  is the puzzle, and  $T \in (1, \infty)$  is the difficulty parameter. As hash functions produce pseudorandom outputs, brute-force searching is the prover’s only strategy. Statistically, with a larger  $T$ , the prover needs to try more nonces.

PoW-based consensus – first introduced in Bitcoin [12] – has become a practical and scalable alternative of traditional Byzantine Fault Tolerant (BFT) consensus. In PoW-based consensus, nodes jointly maintain a blockchain. To append a block

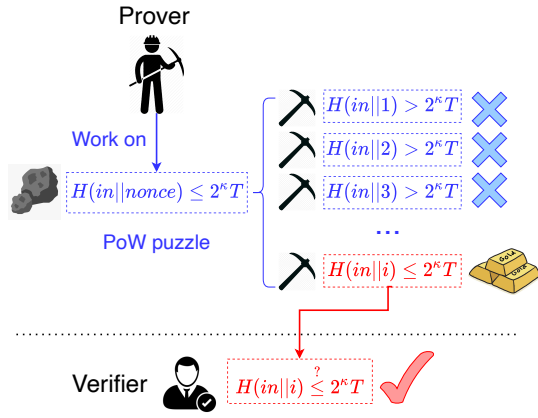


Fig. 1: Proof-of-Work.

to the blockchain, a node should solve a computationally hard PoW puzzle [17]. Given the latest block, each node derives its own PoW puzzle then keeps trying to solve it. Once solving the puzzle, the node appends its block to the blockchain. PoW specifies the amount of work needed by a difficulty parameter. By adaptively adjusting the PoW’s difficulty parameter, PoW-based consensus ensures only one node solves the puzzle for each time period with high probability. Nodes in PoW-based consensus are known as miners, the process of solving PoW puzzles is known as mining, and the computation power used for mining is known as mining power.

**Iteratively sequential functions.** A sequential function  $g(\cdot)$  is that, one cannot accelerate computing  $g(\cdot)$  by parallelism. An iteratively sequential function  $f(t, x)$  is implemented by composing a sequential function  $g(x)$  for a number  $t$  of times, denoted as  $g^t(x)$ . The iteratively sequential function  $f(\cdot)$  inherits the sequentiality of  $g(\cdot)$ : the fastest way of computing  $f(t, x)$  is to iterate  $g(x)$  for  $t$  times. In addition,  $f(\cdot)$  preserves a useful property called *self-composability*: for any  $x$  and  $(t_1, t_2)$ , let  $y \leftarrow f(x, t_1)$ , we have  $f(x, t_1 + t_2) = f(y, t_2)$ . Commonly used iteratively sequential functions include repeated squaring [16], [18] and repeated square rooting [15] on cyclic groups.

**Time-based cryptography.** Iteratively sequential functions are the key ingredient of time-based cryptographic primitives. Rivest, Shamir and Wagner first introduce the timelock puzzle and constructs it from repeated squaring on RSA groups [19]. Proofs of Sequential Work (PoSW) [20] allow to prove he has finished an amount of sequential computation with a succinct proof. Verifiable Delay Functions (VDFs) [14] can be seen as PoSW with *uniqueness*. After finishing the sequential computation, the prover in addition produces an output that is *unique*, i.e., it’s computationally hard to find two inputs that give the same output. Formally,

**Definition 1** (Verifiable Delay Function). A Verifiable Delay Function VDF is a tuple of four algorithms

$$\text{VDF} = (\text{Setup}, \text{Eval}, \text{Prove}, \text{Verify})$$

$\text{Setup}(\lambda) \rightarrow pp$  : On input security parameter  $\lambda$ , outputs public parameter  $pp$ . Public parameter  $pp$  specifies an input domain  $\mathcal{X}$  and an output domain  $\mathcal{Y}$ . We assume  $\mathcal{X}$  is efficiently sampleable.

$\text{Eval}(pp, x, t) \rightarrow y$  : On input public parameter  $pp$ , input  $x \in \mathcal{X}$ , and time parameter  $t \in \mathbb{N}^+$ , produces output  $y \in \mathcal{Y}$ .

$\text{Prove}(pp, x, y, t) \rightarrow \pi$  : On input public parameter  $pp$ , input  $x$ , and time parameter  $t$ , outputs proof  $\pi$ .

$\text{Verify}(pp, x, y, \pi, t) \rightarrow \{0, 1\}$  : On input public parameter  $pp$ , input  $x$ , output  $y$ , proof  $\pi$  and time parameter  $t$ , produces 1 if correct, otherwise 0.

that satisfies the following properties

- *Completeness*: For all  $\lambda$ ,  $x$  and  $t$ ,

$$\Pr \left[ \begin{array}{c} \text{Verify}(pp, x, y, \\ \pi, t) = 1 \end{array} \middle| \begin{array}{c} pp \leftarrow \text{Setup}(\lambda) \\ y \leftarrow \text{Eval}(pp, x, t) \\ \pi \leftarrow \text{Prove}(pp, x, y, t) \end{array} \right] = 1$$

- *Soundness*: For all  $\lambda$  and adversary  $\mathcal{A}$ ,

$$\Pr \left[ \begin{array}{c} \text{Verify}(pp, x, y, \pi, t) = 1 \\ \wedge \text{Eval}(pp, x, t) \neq y \end{array} \middle| \begin{array}{c} pp \leftarrow \text{Setup}(\lambda) \\ (x, y, \pi, t) \leftarrow \mathcal{A}(pp) \end{array} \right] \leq \text{negl}(\lambda)$$

- *$\sigma$ -Sequentiality*: For any  $\lambda$ ,  $x$ ,  $t$ ,  $\mathcal{A}_0$  which runs in time  $O(\text{poly}(\lambda, t))$  and  $\mathcal{A}_1$  which runs in parallel time  $\sigma(t)$ ,

$$\Pr \left[ \begin{array}{c} \text{Eval}(x, y, t) = y \end{array} \middle| \begin{array}{c} pp \leftarrow \text{Setup}(\lambda) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_0(\lambda, t, pp) \\ y \leftarrow \mathcal{A}_1(x) \end{array} \right] \leq \text{negl}(\lambda)$$

VDFs can be constructed from two ingredients: 1) an iteratively sequential function  $f(\cdot)$ , and 2) a succinct argument on  $f(\cdot)$ ’s execution results. For example, two prevalent VDFs [16], [18] employ repeated squaring as the iteratively sequential function. Such VDFs inherit the *self-composability* from iteratively sequential functions, and are known as *self-composable VDFs* [21].

**Definition 2** (Self-Composability). A VDF ( $\text{Setup}, \text{Eval}, \text{Prove}, \text{Verify}$ ) satisfies self-composability if for all  $\lambda$ ,  $x$ ,  $(t_1, t_2)$ ,

$$\Pr \left[ \begin{array}{c} \text{Eval}(pp, x, t_1 + t_2) \\ = \text{Eval}(pp, y, t_2) \end{array} \middle| \begin{array}{c} pp \leftarrow \text{Setup}(\lambda) \\ y \leftarrow \text{Eval}(pp, x, t_1) \end{array} \right] = 1$$

**Lemma 1.** If a VDF ( $\text{Setup}, \text{Eval}, \text{Prove}, \text{Verify}$ ) satisfies self-composability, then for all  $\lambda$ ,  $x$ ,  $(t_1, t_2)$ ,

$$\Pr \left[ \begin{array}{c} \text{Verify}(pp, x, y', \\ \pi, t_1 + t_2) = 1 \end{array} \middle| \begin{array}{c} pp \leftarrow \text{Setup}(\lambda) \\ y \leftarrow \text{Eval}(pp, x, t_1) \\ y' \leftarrow \text{Eval}(pp, y, t_2) \\ \pi \leftarrow \text{Prove}(pp, x, y', t_1 + t_2) \end{array} \right] = 1$$

### 3. SEQUENTIAL PROOF-OF-WORK

In this section, we introduce Sequential Proof-of-Work (SeqPoW), a variant of PoW that is sequential and cannot be accelerated by parallelism. We formally define SeqPoW, provide two constructions with security proofs, evaluate their performance, and discuss its applications.

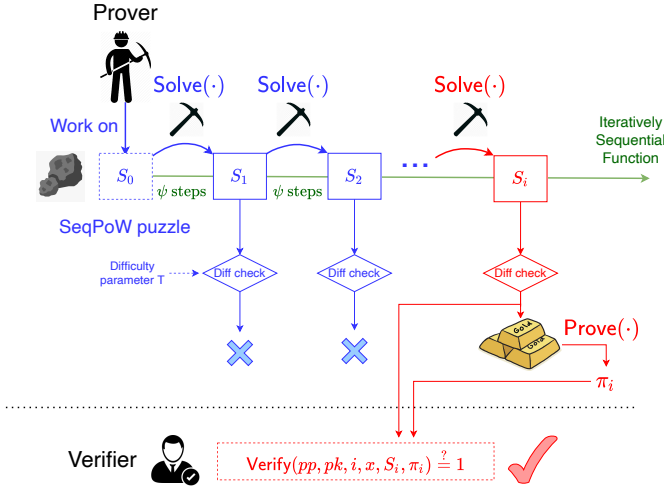


Fig. 2: Sequential Proof-of-Work.

### 1. Intuition and applications

Figure 2 gives the intuition of SeqPoW. Given an initial SeqPoW puzzle  $S_0$ , the solver keeps solving the puzzle by incrementing an iteratively sequential function. Each solving attempt takes the last output  $S_{i-1}$  as input to produce a new output  $S_i$ . For each output  $S_i$ , the prover checks whether  $S_i$  satisfies a difficulty parameter  $T$ . If yes, then  $S_i$  is a valid solution of this SeqPoW puzzle. The prover can generate a proof  $\pi_i$  on a valid solution  $S_i$ , and the verifier with  $S_i$  and  $\pi_i$  can check  $S_i$ 's correctness without solving the puzzle again. Due to the data dependency, the prover cannot solve a SeqPoW puzzle in parallel.

SeqPoW is the main ingredient of RANDCHAIN. Of independent interest, SeqPoW can also be applied to other protocols, such as leader election and PoS-based consensus.

**Leader election.** In PoW-based consensus, PoW mining can be seen as a way of electing leaders: given a set of nodes, the first node proposing a valid PoW solution becomes the leader and proposes a block. As a drop-in replacement of PoW, SeqPoW can be used for leader election. Compared to PoW-based leader election, based leader election is much fairer and energy efficient. Each node can only run a single processor, which prevents marketisation of mining and gives little optimisation space of mining speed. Later in §5-D we will discuss this in detail.

**PoS-based consensus.** In Proof-of-Stake (PoS)-based consensus [22], each node's chance of mining the next block is in proportion to its stake, which can be deposit or balance. In some PoS-based consensus designs [23]–[25], each node solves a VDF with time parameter inversely proportional to its stake, and the first node solving its VDF proposes the next block. However, such designs suffer from the “winner-takes-all” problem, where the node with most stake always solves VDFs faster than others. To avoid the “winner-takes-all” problem, one can replace VDF with SeqPoW, and specify the difficulty parameter in inverse proportion to each node's

stake. As the number of iterations taken to solve a SeqPoW is randomised, the node with most stake is not always the winner, but only with higher chance.

### 2. Definition

We formally define SeqPoW as follows.

**Definition 3** (Sequential Proof-of-Work (SeqPoW)). A Sequential Proof-of-Work SeqPoW is a tuple of algorithms

$$\text{SeqPoW} = (\text{Setup}, \text{Gen}, \text{Init}, \text{Solve}, \text{Verify})$$

$\text{Setup}(\lambda, \psi, T) \rightarrow pp$  : On input security parameter  $\lambda$ , step  $\psi \in \mathbb{N}^+$  and difficulty  $T \in (1, \infty)$ , outputs public parameter  $pp$ . Public parameter  $pp$  specifies an input domain  $\mathcal{X}$ , an output domain  $\mathcal{Y}$ , and a cryptographically secure hash function  $H : \mathcal{Y} \rightarrow \mathcal{X}$ . We assume  $\mathcal{X}$  is efficiently sampleable.

$\text{Gen}(pp) \rightarrow (sk, pk)$  : Probabilistic. On input public parameter  $pp$ , produces a secret key  $sk \in \mathcal{X}$  and a public key  $pk \in \mathcal{X}$ .

$\text{Init}(pp, sk, x) \rightarrow (S_0, \pi_0)$  : On input public parameter  $pp$ , secret key  $sk$ , and input  $x \in \mathcal{X}$ , outputs initial solution  $S_0 \in \mathcal{Y}$  and proof  $\pi_0$ .

$\text{Solve}(pp, sk, S_i) \rightarrow (S_{i+1}, b_{i+1})$  : On input public parameter  $pp$ , secret key  $sk$ , and  $i$ -th solution  $S_i \in \mathcal{Y}$ , outputs  $(i+1)$ -th solution  $S_{i+1} \in \mathcal{Y}$  and result  $b_{i+1} \in \{0, 1\}$ .

$\text{Prove}(pp, sk, i, x, S_i) \rightarrow \pi_i$  : On input public parameter  $pp$ , secret key  $sk$ ,  $i$ , input  $x$ , and  $i$ -th solution  $S_i$ , outputs proof  $\pi_i$ .

$\text{Verify}(pp, pk, i, x, S_i, \pi_i) \rightarrow \{0, 1\}$  : On input  $pp$ , public key  $pk$ ,  $i$ , input  $x$ ,  $i$ -th solution  $S_i$ , and proof  $\pi_i$ , outputs result  $\{0, 1\}$ .

We define honest tuples and valid tuples as follows.

**Definition 4** (Honest tuple). A tuple  $(pp, sk, i, x, S_i, \pi_i)$  is  $(\lambda, \psi, T)$ -honest if and only if for all  $\lambda, \psi, T$  and  $pp \leftarrow \text{Setup}(\lambda, \psi, T)$ , the following holds:

- $i = 0$  and  $(S_0, \pi_0) \leftarrow \text{Init}(pp, sk, x)$ , and
- $i \in \mathbb{N}^+$ ,  $(S_i, b_i) \leftarrow \text{Solve}(pp, sk, S_{i-1})$  and  $\pi_i \leftarrow \text{Prove}(pp, sk, i, x, S_i)$ , where  $(pp, sk, i-1, x, S_{i-1}, \pi_{i-1})$  is  $(\lambda, \psi, T)$ -honest.

**Definition 5** (Valid tuple). For all  $\lambda, \psi, T$ , and  $pp \leftarrow \text{Setup}(\lambda, \psi, T)$ , a tuple  $(pp, sk, i, x, S_i, \pi_i)$  is  $(\lambda, \psi, T)$ -valid if

- $(pp, sk, i, x, S_i, \pi_i)$  is  $(\lambda, \psi, T)$ -honest, and
- $\text{Solve}(pp, sk, S_{i-1}) = (\cdot, 1)$

SeqPoW should satisfy *completeness*, *soundness* and *sequentiality*, *hardness*, plus an optional property *uniqueness*.

**Definition 6** (Completeness). A SeqPoW scheme satisfies completeness if for all  $\lambda, \psi, T$ ,

$$\Pr \left[ \text{Verify}(pp, pk, i, x, S_i, \pi_i) = 1 \mid \begin{array}{l} (sk, pk) \leftarrow \text{Gen}(pp) \\ (pp, pk, i, x, S_i, \pi_i) \\ \text{is } (\lambda, \psi, T)\text{-valid} \end{array} \right] = 1$$

**Definition 7 (Soundness).** A SeqPoW scheme satisfies soundness if for all  $\lambda, \psi, T$ ,

$$\Pr \left[ \begin{array}{l} \text{Verify}(pp, pk, i, \\ x, S_i, \pi_i) = 1 \end{array} \left| \begin{array}{l} (sk, pk) \leftarrow \text{Gen}(pp) \\ (pp, pk, i, x, S_i, \pi_i) \\ \text{is not } (\lambda, \psi, T)\text{-valid} \end{array} \right. \right] \leq \text{negl}(\lambda)$$

Then, we define *hardness*, which requires that each attempt of  $\text{Solve}(\cdot)$  has the success rate of  $\frac{1}{T}$ .

**Definition 8 (Hardness).** A SeqPoW scheme satisfies hardness if for all  $(\lambda, \psi, T)$ -honest tuple  $(pp, sk, i, x, S_i, \pi_i)$ ,

$$\Pr \left[ b_{i+1} = 1 \left| \begin{array}{l} (S_{i+1}, b_{i+1}) \leftarrow \\ \text{Solve}(pp, sk, S_i, \pi_i) \end{array} \right. \right] \leq \frac{1}{T} + \text{negl}(\lambda)$$

We also consider an optional property *uniqueness*. *Uniqueness* requires that, each SeqPoW puzzle only has a single valid solution  $S_i$ . Before finding  $S_i$  each  $\text{Solve}(\cdot)$  attempt follows the *hardness* definition, but after finding the solution  $S_i$  no  $\text{Solve}(\cdot)$  attempt leads to a valid solution. *Uniqueness* implies that, once finding a valid solution  $S_i$ , the prover stops incrementing  $\text{Solve}(\cdot)$ .

**Definition 9 (Uniqueness).** A SeqPoW scheme satisfies uniqueness if for any two  $(\lambda, \psi, T)$ -valid tuples  $(pp, sk, i, x, S_i, \pi_i)$  and  $(pp, sk, i, x, S_j, \pi_j)$ , it holds that  $i = j$ .

In addition, we define *sequentiality* that the fastest way of computing  $S_i$  is computing  $\text{Solve}(\cdot)$  for  $i$  times honestly. Similar to VDF, *sequentiality* in SeqPoW also captures the *unpredictability* that, the adversary  $\mathcal{A}_1$  cannot predict  $S_i$ 's value before time  $\sigma(\psi i)$ .

**Definition 10 ( $\sigma$ -Sequentiality).** A SeqPoW scheme satisfies  $\sigma$ -sequentiality if for all  $\lambda, \psi, T, i, x, \mathcal{A}_0$  which runs in less than time  $O(\text{poly}(\lambda, \psi, i))$  and  $\mathcal{A}_1$  which runs in less than time  $\sigma(\psi i)$ ,

$$\Pr \left[ \begin{array}{l} (pp, sk, i, x, S_i, \pi_i) \\ \text{is } (\lambda, \psi, T)\text{-honest} \end{array} \left| \begin{array}{l} pp \leftarrow \text{Setup}(\lambda, \psi, T) \\ (sk, pk) \leftarrow \text{Gen}(pp) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_0(pp, sk) \\ S_i \leftarrow \mathcal{A}_1(i, x) \\ \pi_i \leftarrow \text{Prove}(pp, sk, i, x, S_i) \end{array} \right. \right] \leq \text{negl}(\lambda)$$

### 3. Constructions

We formally propose two SeqPoW constructions and discuss other possible constructions. Let  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$  be a cryptographic hash function. Let  $G$  be a cyclic group. Let  $H_G(\cdot)$  be a hash function that takes an arbitrarily long string  $\{0, 1\}^*$  to a point on  $G$ . Let  $g$  be a generator of  $G$ .

**SeqPoW from self-composable VDFs.** We first present a SeqPoW construction  $\text{SeqPoW}_{\text{VDF}}$  based on self-composable VDFs. Let  $\psi$  be a step parameter,  $x$  be the input, and  $T$  be the difficulty parameter. Let the initial solution  $S_0 = H_G(pk||x)$ . We take each of  $S_i = f(i\psi, S_0) = \text{VDF.Eval}^i(pp, S_0, \psi)$  as an intermediate output. The prover keeps calculating each  $S_i$ , and checks if  $S_i$  satisfies the difficulty  $T$

by calculating  $H(pk||S_i) \stackrel{?}{\leq} \frac{2^\kappa}{T}$ . If true, then  $S_i$  is a valid solution. The prover can then compute a succinct argument of the statement  $S_i = \text{VDF.Eval}^i(pp, S_0, \psi)$  by running  $\text{VDF.Prove}(pp_{\text{VDF}}, S_0, S_i, i\psi)$ . Note that by *self-composability*,  $\text{VDF.Eval}^i(pp, S_0, \psi) = \text{VDF.Eval}(pp, S_0, i\psi)$ . With the  $pk, i, S_i$  and  $\pi_i$ , the verifier then can check whether 1)  $S_i$  satisfies the difficulty  $T$ , and 2)  $S_i = \text{Eval}^i(pp, S_0, \psi)$ . Figure 3 describes our  $\text{SeqPoW}_{\text{VDF}}$  construction in detail.

**Unique SeqPoW from Sloth.**  $\text{SeqPoW}_{\text{VDF}}$  does not provide *uniqueness*: the prover can keep incrementing the iteratively sequential function to find as many valid solutions as possible. We present  $\text{SeqPoW}_{\text{Sloth}}$  – a SeqPoW construction with *uniqueness*.  $\text{SeqPoW}_{\text{Sloth}}$  is based on Sloth [15], a hash function that supports hashchains with fast verification. Sloth employs repeated square rooting on a cyclic group as the iteratively sequential function. The prover square roots (on a cyclic group  $G$ ) the input for a predefined number  $t$  of times to get the output. To verify the output, the verifier squares – which is  $O(\log |G|)$  times faster than square rooting – the output for  $t$  times to recover the input. This means Sloth is *reversible* [26]: given the output, the verifier can recover intermediate results and the input. Same as in  $\text{SeqPoW}_{\text{VDF}}$ ,  $\text{SeqPoW}_{\text{Sloth}}$  takes each of  $S_i = f(i\psi, S_0)$  as an intermediate output and checks if  $H(pk||S_i) \leq \frac{2^\kappa}{T}$ . In addition,  $\text{SeqPoW}_{\text{Sloth}}$  only treats the first solution satisfying the difficulty as valid, which makes the solution unique. Figure 4 describes the detailed construction of  $\text{SeqPoW}_{\text{Sloth}}$ .

**Other possible constructions.** The main challenge of constructing SeqPoW is the succinct argument of iteratively sequential functions. In addition to VDFs and Sloth, Incremental Verifiable Computation (IVC) [27] can also provide such arguments. IVC is a powerful primitive that, for every step of a computation, a prover can produce a succinct proof by updating the last step's proof rather than computing the proof from scratch. Given an input, an output, a series of computation and the proof, one can verify current output is produced from the series of computation on the input.

The advantage of IVC-based SeqPoW is that it supports any iteratively sequential functions. This means IVC-based SeqPoW can be more egalitarian by using iteratively sequential functions that are more hard to parallelise and optimise. However, IVC is usually constructed from complicated cryptographic primitives, such as SNARKs [27]–[31]. This makes the construction inefficient and the implementation challenging. In addition, when generating proofs in IVC takes non-negligible time, IVC-based SeqPoW may not be fair, as miners with powerful hardware can take advantage of mining by accelerating  $\text{SeqPoW.Prove}(\cdot)$ .

### 4. Security and efficiency analysis

**Security.** We defer full security proofs to Appendix A. Proofs of *completeness* and *soundness* directly follow the completeness, soundness and self-composability of Sloth and VDFs. By pseudorandomness of hash functions and sequentiality of Sloth

<p><b>SeqPoW<sub>VDF</sub>.Setup(<math>\lambda, \psi, T</math>)</b></p> <pre> 1: <math>pp_{VDF} = (\{0, 1\}^*, G, g)</math>    <math>\leftarrow \text{VDF.Setup}(\lambda)</math> 2: <math>pp \leftarrow (pp_{VDF}, \psi, T)</math> 3: <b>return</b> <math>pp</math> </pre>	<p><b>SeqPoW<sub>VDF</sub>.Init(<math>pp, sk, x</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <math>pk \leftarrow g^{sk} \in G</math> 3: <math>S_0 \leftarrow H_G(pk  x)</math> 4: <b>return</b> <math>S_0</math> </pre>	<p><b>SeqPoW<sub>VDF</sub>.Prove(<math>pp, sk, i, x, S_i</math>)</b></p> <pre> 1: <math>(pp_{VDF}, \psi, T) \leftarrow pp</math> 2: <math>(\{0, 1\}^*, G, g) \leftarrow pp_{VDF}</math> 3: <math>pk \leftarrow g^{sk} \in G</math> 4: <math>S_0 \leftarrow H_G(pk  x)</math> 5: <math>\pi_{VDF} \leftarrow \text{VDF.Prove}(pp_{VDF}, S_0, S_i, i\psi)</math> 6: <b>return</b> <math>\pi_{VDF}</math> </pre>
<p><b>SeqPoW<sub>VDF</sub>.Gen(<math>pp</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <b>Sample random</b> <math>sk \in \mathbb{N}</math> 3: <math>pk \leftarrow g^{sk} \in G</math> 4: <b>return</b> <math>(sk, pk)</math> </pre>	<p><b>SeqPoW<sub>VDF</sub>.Solve(<math>pp, sk, S_i</math>)</b></p> <pre> 1: <math>(pp_{VDF}, \psi, T) \leftarrow pp</math> 2: <math>(\{0, 1\}^*, G, g) \leftarrow pp_{VDF}</math> 3: <math>pk \leftarrow g^{sk} \in G</math> 4: <math>S_{i+1} \leftarrow \text{VDF.Eval}(pp_{VDF}, S_i, \psi)</math> 5: <math>b_{i+1} \leftarrow H(pk  S_{i+1}) \leq \frac{2^\kappa}{T} ? 1 : 0</math> 6: <b>return</b> <math>(S_{i+1}, b_{i+1})</math> </pre>	<p><b>SeqPoW<sub>VDF</sub>.Verify(<math>pp, pk, i, x, S_i, \pi_i</math>)</b></p> <pre> 1: <math>(pp_{VDF}, \psi, T) \leftarrow pp</math> 2: <math>(\{0, 1\}^*, G, g) \leftarrow pp_{VDF}</math> 3: <math>S_0 \leftarrow H_G(pk  x)</math> 4: <b>if</b> <math>\text{VDF.Verify}(pp_{VDF}, S_0, S_i, \pi_i, i\psi) = 0</math> 5:   <b>return</b> 0 6: <b>if</b> <math>H(pk  S_i) &gt; \frac{2^\kappa}{T}</math> <b>then return</b> 0 7: <b>return</b> 1 </pre>

Fig. 3: Construction of SeqPoW<sub>VDF</sub>.

<p><b>SeqPoW<sub>VDF</sub>.Setup(<math>\lambda, \psi, T</math>)</b></p> <pre> 1: <math>pp \leftarrow (\{0, 1\}^*, G, g, \psi, T)</math> 2: <b>return</b> <math>pp</math> </pre>	<p><b>SeqPoW<sub>Sloth</sub>.Init(<math>pp, sk, x</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <math>pk \leftarrow g^{sk} \in G</math> 3: <math>S_0 \leftarrow H_G(pk  x)</math> 4: <b>return</b> <math>S_0</math> </pre>	<p><b>SeqPoW<sub>Sloth</sub>.Prove(<math>pp, sk, i, x, S_i</math>)</b></p> <pre> 1: <b>return</b> <math>\perp</math> </pre>
<p><b>SeqPoW<sub>VDF</sub>.Gen(<math>pp</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <b>Sample random</b> <math>sk \in \mathbb{N}</math> 3: <math>pk \leftarrow g^{sk} \in G</math> 4: <b>return</b> <math>(sk, pk)</math> </pre>	<p><b>SeqPoW<sub>Sloth</sub>.Solve(<math>pp, sk, S_i</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <math>pk \leftarrow g^{sk} \in G</math> 3: <math>S_{i+1} \leftarrow S_i^{2^\psi}</math> 4: <math>b_{i+1} \leftarrow H(pk  S_{i+1}) \leq \frac{2^\kappa}{T} ? 1 : 0</math> 5: <b>return</b> <math>(S_{i+1}, b_{i+1})</math> </pre>	<p><b>SeqPoW<sub>Sloth</sub>.Verify(<math>pp, pk, i, x, S_i, \pi_i</math>)</b></p> <pre> 1: <math>(\{0, 1\}^*, G, g, \psi, T) \leftarrow pp</math> 2: <math>y \leftarrow S_i</math> 3: <b>if</b> <math>H(pk  y) &gt; \frac{2^\kappa}{T}</math>, <b>then return</b> 0 4: <b>repeat</b> <math>i</math> <b>times</b> 5:   <math>y \leftarrow y^{2^\psi}</math> 6:   <b>if</b> <math>H(pk  y) \leq \frac{2^\kappa}{T}</math> <b>then return</b> 0 7: <math>g \leftarrow H_G(pk  x)</math> 8: <b>if</b> <math>g \neq y</math> <b>then return</b> 0 9: <b>return</b> 1 </pre>

Fig. 4: Construction of SeqPoW<sub>Sloth</sub>.

TABLE II: Efficiency of two SeqPoW constructions. ISF stands for iteratively sequential function.

	Construction		Efficiency			
	ISF	Proof	Solve( $\cdot$ )	Prove( $\cdot$ )	Verify( $\cdot$ )	Proof size (Bytes)
SeqPoW <sub>VDF</sub>	Repeated SQ.	Wes19 [18]	$O(\psi)$	$O(\psi T)$	$O(\log \psi T)$	$s$
		Pie19 [16]	$O(\psi)$	$O(\sqrt{\psi T} \log \psi T)$	$O(\log \psi T)$	$s \log_2 \psi T$
SeqPoW <sub>Sloth</sub>	Repeated SQRT.	Repeated SQ.	$O(\psi)$	0	$O(\psi T)$	0

and VDFs, Solve( $\cdot$ ) outputs unpredictable solutions and the probability that the solution satisfies the difficulty is  $\frac{1}{T}$ , leading to *hardness*. *Sequentiality* directly follows the sequentiality and self-composability of Sloth and VDFs.

**Efficiency.** Table II summarises the efficiency analysis of two SeqPoW constructions. SeqPoW<sub>VDF</sub> and SeqPoW<sub>Sloth</sub> employ

repeated squaring on an RSA group and repeated square rooting on a group of prime order as the iteratively sequential function, respectively. Let  $s$  be the size (in Bytes) of an element on the group, and  $\psi$  be the step parameter. Each Solve( $\cdot$ ) executes  $\psi$  steps of the iteratively sequential function. By *hardness* and *uniqueness*, a node attempts Solve( $\cdot$ ) for  $T$  times to find a valid solution on average. Thus, Prove( $\cdot$ ) generates proofs on  $\psi T$  steps, and Verify( $\cdot$ ) verifies proofs of  $\psi T$  steps. SeqPoW<sub>VDF</sub> employs either Wesolowski's VDF (Wes19) [18] or Pietrzak's VDF (Pie19) [16] for succinct arguments of repeated squaring. According to the existing analysis [32], the proving complexity, verification complexity and proof size of Wes19's succinct argument are  $O(\psi T)$ ,  $O(\log \psi T)$  and  $s$  Bytes, respectively; and the proving complexity, verification complexity and proof size of Pie19's succinct argument are  $O(\sqrt{\psi T} \log \psi T)$ ,  $O(\log \psi T)$  and  $s \log_2 \psi T$ , respectively.

There have been techniques for optimising and parallelising  $\text{VDF.Prove}(\cdot)$  for both VDFs [16], [18], [33]. The proof sizes of both  $\text{SeqPoW}_{\text{VDF}}$  constructions are acceptable: when  $\psi T = 2^{40}$  and  $s = 32$  Bytes, the proof sizes of  $\text{SeqPoW}_{\text{VDF}}$  with Wes [18] and with Pie19 [16] are 32 and 1280 Bytes, respectively.  $\text{SeqPoW}_{\text{Sloth}}$  does not have proofs; instead, the verifier keeps squaring the solution to recover the input and check whether the recovered input equals to the real one. This leads to verification complexity of  $O(\psi T)$ .

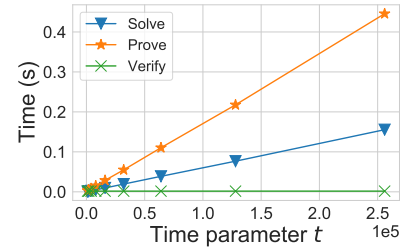
### 5. Performance evaluation

We evaluate the performance of our two  $\text{SeqPoW}$  constructions. The code is available at Github [34]. Our results show that, without optimisation,  $\text{SeqPoW}_{\text{VDF}}$  with Pie19 [16] provides better performance than  $\text{SeqPoW}_{\text{VDF}}$  with Wes19 [18], as generating and verifying proofs is efficient and the proof size is acceptable. In addition,  $\text{SeqPoW.Sloth}$  achieves acceptable verification efficiency, although far from the theoretically optimal value.

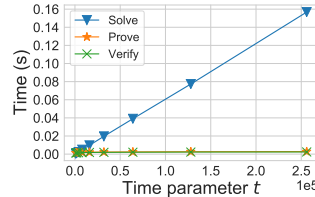
**Implementation.** We implement the two  $\text{SeqPoW}$  constructions in Rust programming language. We use the `rug` [35] crate for big integer arithmetic. We implement the RSA group with 1024-bit keys and the group of prime order. We implement the two  $\text{SeqPoW}_{\text{VDF}}$  constructions based on the RSA group, and  $\text{SeqPoW.Sloth}$  based on the group of prime order. Our implementations strictly follow their original papers [15], [16], [18] without any optimisation.

**Experimental setup.** We benchmark  $\text{Solve}(\cdot)$ ,  $\text{Prove}(\cdot)$  and  $\text{Verify}(\cdot)$  for each  $\text{SeqPoW}$  construction. We test  $\psi \in [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000]$  and assume  $i = 1$ . Note that  $\text{SeqPoW}_{\text{Sloth}}$  does not have  $\text{Prove}(\cdot)$ . We benchmark the performance using Rust’s native benchmarking suite `cargo-bench` [36] and `criterion` [37]. We sample ten executions for each configuration, i.e., a function with a unique group of parameters. We specify O3-level optimisation when compiling. All experiments were conducted on a MacBook Pro with a 2.2 GHz 6-Core Intel Core i7 Processor and a 16 GB 2400 MHz DDR4 RAM.

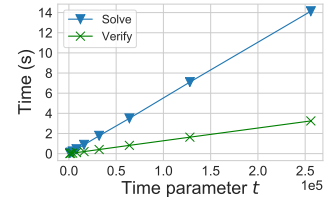
**Results (Figure 5).** For all  $\text{SeqPoW}$  constructions, the running time of  $\text{Solve}(\cdot)$  increases linearly with time parameter  $t$ . This is as expected as  $\text{Solve}(\cdot)$  is dominated by the iteratively sequential function. For  $\text{SeqPoW}_{\text{VDF}}$  with Wes19,  $\text{Prove}(\cdot)$  takes more time than  $\text{Solve}(\cdot)$ , as our implementation follows the original algorithm rather than considering other optimisations (such as [33]).  $\text{SeqPoW}_{\text{VDF}}$  with Pie19 achieves ideal performance:  $\text{Prove}(\cdot)$  and  $\text{Verify}(\cdot)$  takes negligible time compared to  $\text{Solve}(\cdot)$ . In addition, according to the efficiency analysis, the proof size of Pie19 is also acceptable. Thus, without optimisation,  $\text{SeqPoW}_{\text{VDF}}$  with Pie19 is more practical than with Wes19. For  $\text{SeqPoW}_{\text{Sloth}}$ ,  $\text{Solve}(\cdot)$  is approximately five times slower than  $\text{Verify}(\cdot)$ . Although this is far from theoretically optimal value, i.e.,  $\log_2 |G| = 2048$  in our case [26], the verification overhead is acceptable when random outputs are not generated frequently.



(a)  $\text{SeqPoW}_{\text{VDF}}$  + Wes19 [18].



(b)  $\text{SeqPoW}_{\text{VDF}}$  + Pie19 [16].



(c)  $\text{SeqPoW}_{\text{Sloth}}$ .

Fig. 5: Evaluation of  $\text{SeqPoW}$  constructions.

### 4. RANDCHAIN: DRB FROM SEQPOW

Based on Sequential Proof-of-Work ( $\text{SeqPoW}$ ), we construct  $\text{RANDCHAIN}$ , a new family of Decentralised Randomness Beacons (DRBs). Figure 6 describes the full construction of  $\text{RANDCHAIN}$ .

**System setting.** Let  $H, H' : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$  be two different hash functions.  $\text{RANDCHAIN}$  works in a permissioned system. Nodes  $\mathcal{P} = \{p_1, \dots, p_n\}$  register themselves into the system through an identity management service such as Public Key Infrastructure. The identity management service assigns each registered node  $p_k \in \mathcal{P}$  with a pair of secret key  $sk_k$  and public key  $pk_k$ . Each node is identified by its public key in the system. Given a public key, any node in the system can verify whether the node is in the system by querying the PKI. There can be unlimited number of registered nodes. Each node does not know the exact number of nodes in the system, and is only directly connected to a subset of peers.

**Structure (Figure 7a).** All nodes jointly maintain a *blockchain*. Each block  $B$  is of the format  $(h^-, h, i, S, pk, \pi)$ , where  $h^-$  is the previous block ID,  $h$  is the current block ID,  $i$  is the  $\text{SeqPoW}$  solution index,  $S$  is the  $\text{SeqPoW}$  solution,  $pk$  is the public key of this block’s creator, and  $\pi$  is the proof that  $S$  is a valid  $\text{SeqPoW}$  solution on input  $h^-$ . The block ID  $B.h$  is calculated as  $B.h = H(pk \| S)$ . The block’s random output  $B.rand$  is calculated as  $B.rand = H'(pk \| S)$ .

Each node  $p_k$  has its local view  $\mathcal{C}_k$  of the blockchain. The local view  $\mathcal{C}_k$  may have *forks*: there may be multiple blocks following the same block.  $\text{RANDCHAIN}$  applies Nakamoto consensus that, nodes considers the longest chain among all forks as valid. As described in  $\text{mainChain}(\cdot)$  of Figure 6, node  $p_k$  considers the longest fork of  $\mathcal{C}_k$  as the main chain  $\mathcal{MC}_k$ .

**Process (Figure 7b).** We describe the process of  $\text{RANDCHAIN}$  from a node  $p_k$ ’s perspective. Node  $p_k$  runs two routines: the synchronisation routine  $\text{SyncRoutine}(\cdot)$  and the

<pre> mainChain(<math>\mathcal{C}_k</math>) 1: <math>tmp \leftarrow \epsilon</math> 2: <b>for</b> <math>\mathcal{C}_k^t \in \mathcal{C}_k</math> 3:   <math>tmp \leftarrow \max(\mathcal{C}_k^t, tmp)</math> 4: <b>return</b> <math>\mathcal{MC}_k = tmp</math>  MainProcedure(<math>pp, sk_k, pk_k</math>) 1: Synchronise chain as <math>\mathcal{C}_k</math> 2: MineRoutine(<math>pp, sk_k, pk_k, \mathcal{C}_k</math>) in a thread 3: SyncRoutine(<math>pp, \mathcal{C}_k</math>) in a thread  SyncRoutine(<math>pp, \mathcal{C}</math>) 1: <b>while</b> <i>True</i> 2:   Wait for a new block as <math>B</math> 3:   <math>(h^-, h, i, S, pk, \pi) \leftarrow B</math> 4:   <b>if</b> <math>h^- \notin \mathcal{C}_k</math> <b>then</b> Discard <math>B</math> 5:   <b>if</b> <math>h \neq H(pk  S)</math> <b>then</b> Discard <math>B</math> 6:   <b>if</b> SeqPoW.Verify(<math>pp, pk, i, h^-, S, \pi</math>) = 0 7:     Discard <math>B</math> 8:     Append <math>B</math> to <math>\mathcal{C}_k</math> after block with hash <math>h^-</math> 9:     Propagate <math>B</math> </pre>	<pre> MineRoutine(<math>pp, sk_k, pk_k, \mathcal{C}_k</math>) 1: <b>while</b> <i>True</i> 2:   <math>\mathcal{MC}_k \leftarrow \text{mainChain}(\mathcal{C}_k)</math> 3:   <math>B^- \leftarrow \mathcal{MC}_k[-1]</math> 4:   <math>i \leftarrow 0</math> 5:   <math>S \leftarrow \text{SeqPoW.Init}(pp, sk_k, B^-.h)</math> 6:   <b>while</b> <i>True</i> 7:     <math>S, b \leftarrow \text{SeqPoW.Solve}(pp, sk_k, S)</math> 8:     <b>if</b> <math>b = 1</math> <b>then</b> Break 9:     <math>i+ = 1</math> 10:    <math>\mathcal{MC}'_k \leftarrow \text{mainChain}(\mathcal{C}_k)</math> 11:    <b>if</b> <math>\mathcal{MC}_k \neq \mathcal{MC}'_k</math> 12:      <math>\mathcal{MC}_k \leftarrow \mathcal{MC}'_k</math> 13:      Repeat line3-6 14:    <math>h \leftarrow H(pk_k  S)</math> 15:    <math>\pi \leftarrow \text{SeqPoW}_{\text{VDF}}.\text{Prove}(pp, sk, i, B^-.h, S)</math> 16:    <math>B \leftarrow (B^-.h, h, i, S, pk_k, \pi)</math> 17:    <b>New random output</b> <math>B.rand \leftarrow H'(pk_k  S)</math> 18:    Append <math>B</math> to <math>\mathcal{MC}_k</math> after <math>B^-</math> 19:    Propagate <math>B</math> </pre>
--	--

Fig. 6: Construction of RANDCHAIN.

mining routine MineRoutine( $\cdot$ ). In SyncRoutine( $\cdot$ ), node  $p_k$  synchronises its local blockchain  $\mathcal{C}_k$  with other nodes. The synchronisation process is the same as in other blockchains: node  $p_k$  keeps receiving blocks from other nodes, verifying them, and adding valid blocks to its local blockchain  $\mathcal{C}_k$ .

In MineRoutine( $\cdot$ ), node  $p_k$  keeps *mining*, i.e., adding new blocks, on the main chain  $\mathcal{MC}_k$ . To append a block to the blockchain, node  $p_k$  should solve a SeqPoW puzzle. In particular, node  $p_k$  finds its latest main chain  $\mathcal{MC}_k$ , and derives a SeqPoW puzzle from  $\mathcal{MC}_k$ 's last block. Then, node  $p_k$  executes SeqPoW.Init( $\cdot$ ) to find the starting point of mining. Node  $p_k$  keeps solving SeqPoW by iterating SeqPoW.Solve( $\cdot$ ) until finding a solution  $S$  satisfying the difficulty. As SeqPoW.Init( $\cdot$ ) takes each node's secret key as input, nodes' SeqPoW puzzles start from different points and take different steps to solve. With a valid solution  $S$ , node  $p_k$  constructs a block  $B$  consisting of a random output, and appends  $B$  to  $\mathcal{MC}_k$ .

Similar to PoW-based blockchains, RANDCHAIN's blockchain may have forks, and it's possible that  $B$  is committed then reverted by a longer chain. By applying a high difficulty parameter  $T$ , the average time of mining a new block can be set much longer than network delay. This gives nodes sufficient time to propagate new blocks, and thus the fork rate can be reduced. Meanwhile, by Nakamoto consensus, if the majority of nodes honestly mine the longest chain and a sufficient number of valid blocks are appended after  $B$ , then  $B$  cannot be reverted, i.e., is *stable*, except

for negligible probability [38]. We analyse RANDCHAIN's consistency guarantee in §5, and discuss how to achieve *finality*, i.e., make blocks irreversible in §6-B.

## 5. SECURITY AND EFFICIENCY OF RANDCHAIN

In this section, we analyse the security and efficiency of RANDCHAIN. We define the notion of Decentralised Randomness Beacon (DRB), and prove that RANDCHAIN implements a DRB. In addition, we show that although constructed from PoW variants and Nakamoto consensus, RANDCHAIN is energy efficient and decentralised.

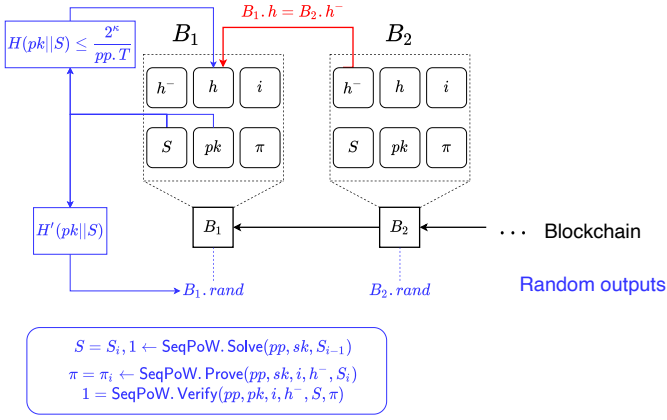
### 1. Security model

We consider the network is *synchronous*: all messages are delivered within a known time bound  $\Delta$ . We do not assume rounds or *lock-step synchrony* [38]. The network consists of  $n$  nodes, each of which controls the same amount of mining power. We consider an adaptive Byzantine adversary who can corrupt any of  $\alpha n$  nodes at any time, where  $\alpha \in [0, 1]$ . The adversary can coordinate its corrupted nodes in real-time without delay, and can arbitrarily delay, drop, forge and modify messages sent from its corrupted nodes. Let  $\beta = 1 - \alpha$  be the percentage of correct nodes.

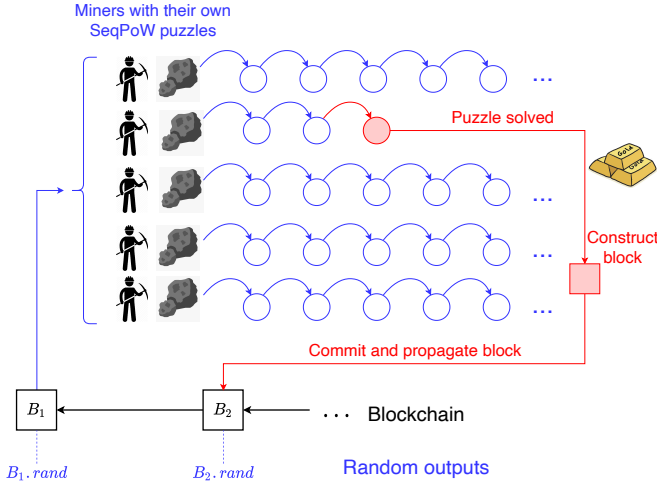
### 2. Defining Decentralised Randomness Beacon

We start from defining security properties of DRB. First, similar to Nakamoto consensus, DRB should satisfy *consistency* and *liveness*. We follow existing papers [39]–[42] for defining *consistency* and *liveness*. *Consistency* requires





(a) Beacon structure.



(b) Process of mining.

Fig. 7: Illustration of RANDCHAIN beacon.

nodes to have a consistent view on the blockchain. Without *consistency*, nodes may revert random outputs arbitrarily.

**Definition 11** (Consistency). Parametrised by  $\ell \in \mathbb{N}$ . A DRB satisfies  $\ell$ -consistency if for any two correct nodes at any time, their chains can differ only in the last  $\ell$  blocks, except for negligible probability.

The parameter  $\ell$  defines the degree of consistency guarantee. Some based applications require RB to have *finality*, i.e., at any time, correct nodes do not have conflicted views on the blockchain. The *finality* here is equivalent to 0-consistency. In §6 we discuss two approaches for adding *finality* to RANDCHAIN.

*Liveness* requires DRB to produce no less than  $\lfloor \tau \cdot t \rfloor$  random outputs for every time period of  $t$ . Without *liveness*, DRB may stop producing randomness forever. Existing papers usually define *termination* [5], [43], [44] or *Guaranteed Output Delivery (G.O.D.)* [45]–[48] that, for every round, there will always be a new random output. We do not follow their definitions, as RANDCHAIN neither iterates single-shot DRG protocols nor uses the concept of rounds. In §7 we will

compare RANDCHAIN with existing DRBs in detail.

**Definition 12** (Liveness). Parametrised by  $t, \tau \in \mathbb{R}^+$ . A DRB satisfies  $(t, \tau)$ -liveness if for any time period  $t$ , every correct node receives at least  $\lfloor \tau \cdot t \rfloor$  new outputs.

Then, as a DRB, each output should be pseudorandom, i.e., uniformly distributed.

**Definition 13** (Uniform distribution). A DRB satisfies uniform distribution if every output is indistinguishable with a uniformly random string with the same length, except for negligible probability.

Last, each output should be *unpredictable*: given the current blockchain, no one can learn any knowledge of the next output. If one can predict the next output, it may take advantage in protocols based on the DRB. From a node’s perspective, this includes two scenarios: 1) the next output is generated by itself, and 2) the next output is generated by other nodes. In some papers [47]–[50], unpredictability in the first scenario is referred as *bias-resistance*. We follow the *IND1-secrecy* (Indistinguishability of secrets) notion in SCRAPE [45] to define *unpredictability*. IND1-secrecy requires that each node cannot learn anything about the final output before finishing the protocol.

**Definition 14** (Unpredictability). A DRB satisfies unpredictability if no adversary can obtain non-negligible advantage on the following game. Assuming all messages are delivered instantly and nodes agree on a blockchain of length  $\ell$ . Before the  $(\ell + 1)$ -th block is mined (either by other nodes or by the adversary), the adversary makes a guess on the random output in the  $(\ell + 1)$ -th block. Let  $r$  be the guessed random output, and  $r'$  be the real random output. The adversary’s advantage is quantified as  $\Pr[r = r']$ .

### 3. Security analysis

For the four desired properties, RANDCHAIN’s *consistency* and *liveness* are guaranteed by Nakamoto consensus. There have been extensive works [38]–[41], [51]–[54] analysing and proving consistency and liveness guarantee of Nakamoto consensus. As RANDCHAIN works in the same system model as that of Ren [41], RANDCHAIN at least satisfies the *consistency* and *liveness* bound proved by Ren [41]. For the *uniform distribution* property, it is straightforward to see that as each random output is produced by a hash function, RANDCHAIN satisfies uniform distribution. In partial, for every block  $B$ ,  $B.rand$  is produced by the hash function  $H'(\cdot)$ . By pseudorandomness of hash functions,  $B.rand$  indistinguishable with a uniformly random  $\kappa$ -bit string. Thus, we put our focus on proving the *unpredictability*.

We then analyse *unpredictability*. Assuming all messages are delivered instantly, and nodes agree on a blockchain of length  $\ell$ . In the prediction game, the  $(\ell + 1)$ -th block is either produced by correct nodes or the adversary’s nodes. If the adversary’s advantage is negligible for both cases, then RANDCHAIN satisfies *unpredictability*. We prove the

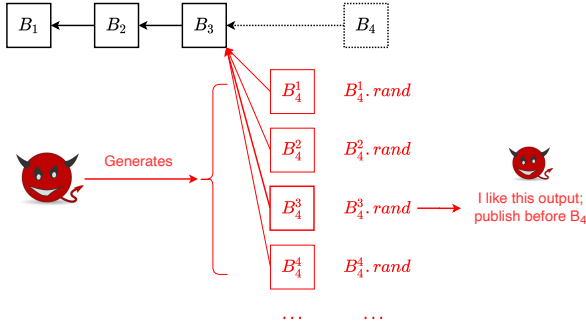


Fig. 8: Unpredictability game.

adversary's advantage for both cases. For the first case that the  $(\ell + 1)$ -th block is produced by correct nodes, the adversary's best strategy is guessing, leading to negligible advantage.

**Lemma 2.** *Assuming all messages are delivered instantly and nodes agree on a blockchain of length  $\ell$ . If the  $(\ell + 1)$ -th block is produced by a correct node, then the adversary's advantage on the prediction game is  $\frac{1}{2^\kappa}$ .*

We then consider the case that the next output is produced by the adversary's nodes. By *sequentiality*, the adversary cannot predict each of its SeqPoW solution. Thus, as shown in Figure 8, the adversary's best strategy is to produce as many blocks before the next honest block as possible. With more blocks, the adversary has more random outputs to choose, leading to higher advantage. We first analyse RANDCHAIN using SeqPoW without *uniqueness*, e.g., SeqPoW<sub>VDF</sub>.

**Lemma 3.** *Consider RANDCHAIN using SeqPoW without uniqueness. Assuming all messages are delivered instantly and nodes agree on a blockchain of length  $\ell$ . If the  $(\ell + 1)$ -th block is produced by the adversary, then the adversary's advantage on the prediction game is  $\frac{k}{2^\kappa}$  with  $\alpha^k \beta$  probability, where  $k \leq \alpha n$ .*

*Proof.* The adversary controls  $\alpha n$  nodes, and  $k \leq \alpha n$ . Let  $V_k$  be the event that “the adversary mines  $k$  blocks before correct nodes mine the first block”. By *hardness*, each node can find unlimited valid SeqPoW solutions given a fixed input. Then, we have

$$\Pr[V_k] = \alpha^k \beta$$

When  $V_k$  happens, the adversary's advantage is  $\frac{k}{2^\kappa}$ .

Thus, with the probability  $\alpha^k \beta$ , the adversary mines  $k$  blocks before correct nodes mine a block, leading to the advantage of  $\frac{k}{2^\kappa}$  (where  $k \leq \alpha n$ ).  $\square$

We then analyse RANDCHAIN using SeqPoW with *uniqueness*, e.g., SeqPoW<sub>Sloth</sub>.

**Lemma 4.** *Consider RANDCHAIN using SeqPoW with uniqueness. Assuming all messages are delivered instantly and nodes agree on a blockchain of length  $\ell$ . If the  $(\ell + 1)$ -th block is produced by the adversary, then the adversary's advantage on the prediction game is  $\prod_{i=0}^{k-1} \frac{\alpha n - i}{n - i} \cdot \beta$  with  $\alpha^k \beta$  probability.*

*Proof.* The adversary controls  $\alpha n$  nodes, and  $k \leq \alpha n$ . Let  $V'_k$  be the event that “the adversary mines  $k$  blocks before correct nodes mine the first block”. By *uniqueness*, each node can find only one valid SeqPoW solutions given a fixed input. Let  $\alpha_k$  be the mining power of the adversary's  $k$ -th node, and  $\sum \alpha_k = \alpha$ . Then, we have

$$\Pr[V'_0] = \beta \quad (1)$$

$$\Pr[V'_1] = \alpha \beta \quad (2)$$

$$\Pr[V'_2] = \frac{\alpha n - 1}{n - 1} \alpha \beta \quad (3)$$

$$\Pr[V'_3] = \frac{\alpha n - 2}{n - 2} \cdot \frac{\alpha n - 1}{n - 1} \alpha \beta \quad (4)$$

$$\dots \quad (5)$$

$$\Pr[V'_k] = \prod_{i=0}^{k-1} \frac{\alpha n - i}{n - i} \cdot \beta \quad (6)$$

When  $V'_k$  happens, the adversary's advantage is  $\frac{k}{2^\kappa}$ .

Therefore, with less than the probability  $\prod_{i=0}^{k-1} \frac{\alpha n - i}{n - i} \cdot \beta$ , the adversary mines  $k$  blocks before correct nodes mine a block, leading to the advantage of  $\frac{k}{2^\kappa}$  (where  $k \leq \alpha n$ ).  $\square$

**Remark 1.** Note that the probability that the adversary achieves a certain advantage in RANDCHAIN with *uniqueness* is always smaller than in RANDCHAIN without *uniqueness*. In particular, for every  $k$ ,  $\Pr[V'_k] \leq \Pr[V_k]$ . Given  $k$ , we have

$$\Pr[V'_k] = \prod_{i=0}^{k-1} \frac{\alpha n - i}{n - i} \cdot \beta \quad (7)$$

$$= \prod_{i=0}^{k-1} \left[ \alpha \cdot \frac{\alpha n - i}{\alpha n - \alpha i} \right] \cdot \beta \quad (8)$$

As  $i \leq k$  and  $\alpha \in [0, 1]$ ,  $\frac{\alpha n - i}{\alpha n - \alpha i} \leq 1$ . Thus,

$$\Pr[V'_k] \leq \prod_{i=0}^{k-1} \alpha \cdot \beta \quad (9)$$

$$= \alpha^k \beta = \Pr[V_k] \quad (10)$$

#### 4. Efficiency

Mining in RANDCHAIN is non-parallelisable: 1) SeqPoW is sequential, and 2) given the last block, each node's input of SeqPoW is fixed. Thanks to non-parallelisable mining, RANDCHAIN is more energy efficient and decentralised than PoW-based consensus.

Mining in PoW-based consensus is parallelisable. Parallelisable mining leads to mining marketisation, which further leads to centralisation and huge energy consumption. When mining is parallelisable, miners can employ specialised mining hardware such as GPUs, FPGAs and ASICs to maximise their mining reward. Due to the mining reward, blockchain mining has been marketised: people manufacture, invest and trade high-end mining hardware for higher profit. Eventually,

most blocks are mined by several miners with massive high-end mining hardware, while few blocks are mined by normal miners. Mining power centralisation weakens PoW-based consensus’ security, as powerful miners can perform various attacks, e.g., 51% attacks [12] to break PoW-based consensus. In addition, mining – which enforces hardware to fully operate all the time – is energy inefficient. As miners deploy powerful mining hardware, mining has cost a great amount of electricity. For example, in 2019, Bitcoin costs 58.93 KWh energy – more than Switzerland which costs 58.46 KWh [55].

RANDCHAIN is more energy efficient and decentralised compared to PoW-based consensus. In RANDCHAIN, miners cannot parallelise mining: each miner can only use a single processor to mine. Compared to massively parallel PoW mining, a single fully operating processor costs negligible electricity. In addition, the performance gap on SeqPoW.Solve( $\cdot$ ) between high-end and low-end hardware is small. Given the sequentiality, SeqPoW.Solve( $\cdot$ ) can only be accelerated by using processors with higher clock rate. The highest clock rate achieved by processors is 8.723 GHz, while laptops’ clock rate is usually more than 2 GHz [56]. This means one can only speed up SeqPoW mining for  $4 \sim 5$  times. In addition, given the voltage limit of processors, high clock rate is hard to improve further [13]. Thus, powerful nodes cannot obtain much speedup on mining, leading to high degree of mining power decentralisation.

## 6. DISCUSSION

### 1. Security issues in Nakamoto consensus

RANDCHAIN employs Nakamoto consensus, so inherits most security issues from Nakamoto consensus. Nonetheless, RANDCHAIN is a DRB rather than a transaction ledger, and security issues for transaction ledgers may not be critical for DRB. For example, RANDCHAIN is immune to all attacks on Nakamoto consensus’ incentive mechanisms [57]–[60], as it works in permissioned networks. In addition, RANDCHAIN can resist some attacks in Nakamoto consensus. SeqPoW can be used for detecting withholding behaviours. If a node broadcasts a valid block with solution index  $i$  after another valid block with solution index  $j < i$ , other nodes can know that the node withholds the block with  $i$ .

### 2. Adding finality

*Finality* [61] is a property for Byzantine consensus that, previously agreed proposals cannot be reverted. RANDCHAIN does not satisfy *finality* due to the probabilistic consensus guarantee of Nakamoto consensus. In some scenarios, we require a DRB to satisfy finality. For example, considering a leader election scheme where a group of nodes elect a leader with a random output from RANDCHAIN as input, if the used random output is reverted before the end of leader election, then nodes may stop working and lose liveness. RANDAO [7] and Proofs-of-Delay [10] bypass this by replacing  $H'(\cdot)$  with a VDF which takes time more than generating  $\ell$  blocks to execute, where  $\ell$  is the degree of  $\ell$ -consistency. Nodes can reveal every random output only after the block deriving

this random output becomes stable. However, this enables *frontrunning* – nodes with fast processors always learn random outputs earlier than normal nodes – which may lead to serious fairness issues (e.g., in decentralised exchanges [62]).

We consider principled approaches for adding *finality*. Adding *finality* is equivalent to achieve *0-consistency*: correct nodes decide the same block at every height. This is further equivalent to making RANDCHAIN to execute in rounds, in each of which nodes execute a *Byzantine agreement* [63] to agree on a block. RANDCHAIN satisfies *validity* and *termination*, but provides eventual consistency rather than the *agreement* property of *Byzantine agreement*, as nodes may temporarily agree on different blocks at the same height. We discuss two approaches to achieve *agreement*, namely the quorum-based approach and the herding-based consensus.

**Quorum-based approach.** In Byzantine agreement, quorum [64] is the minimum number of votes that a proposal has to obtain for being agreed by nodes. If a proposal obtains a quorum of votes in a view, then this means that nodes have reached an agreement on this proposal. The quorum size is  $n - f$ , where  $n$  and  $f$  be the number of nodes and faulty nodes in the system, respectively. Existing research [64], [65] shows that  $n \geq 3f + 1$  is required to achieve agreement in partially synchronous networks, and  $n \geq 2f + 1$  is required to achieve agreement in synchronous networks. A quorum certificate of a proposal is a quorum of votes on this proposal. A vote is usually represented as a digital signature on the proposal, view ID and other metadata.

To achieve *agreement* in RANDCHAIN, we can apply the quorum mechanism as follows. The system should additionally assume  $2f + 1$  correct nodes. A node signs a block to vote this block. The node’s view is represented as the previous block hash, which is inside the signed block. Nodes actively propagate their votes – i.e., signatures on blocks – the same way as propagating blocks. Each node keeps received votes locally, and considers a block as finalised if collecting a quorum certificate, i.e.,  $\geq 2f + 1$  signatures on this block. RANDCHAIN still keeps Nakamoto consensus as a fallback solution. If there are multiple forks without quorum certificates, nodes mine on the longest fork. A block can be considered finalised with a sufficiently long sequence of succeeding blocks, even without a quorum certificate. This resembles the Streamlet blockchain [42].

**Herding-based consensus.** There have been a family of consensus protocols based on *herding*. *Herding* is a social phenomenon where people make choices according to other people’s preference. In herding-based consensus, nodes keep exchanging their votes with each other and decide the proposal with most votes. Existing research [66], [67] shows that, with overwhelming probability, nodes will eventually agree on a proposal by herding in a short time period. In addition, herding-based consensus introduces much less communication overhead than traditional Byzantine consensus.

To achieve *agreement* in RANDCHAIN, we can apply the herding-based consensus as follows. Upon a new block, nodes

execute a herding-based consensus on it. If a block is the only block in a long time period, then nodes will agree on this block directly. If there are multiple blocks within a short time period, then nodes will agree on the most popular block among them with overwhelming probability. This approach has also been discussed in Bitcoin Cash community, who seeks to employ Avalanche [67] as a finality layer for Bitcoin Cash [68].

### 3. Making SeqPoW mining non-outsourcable

RANDCHAIN does not prevent *outsourcing*: the adversary can solve others' SeqPoW puzzles. An adversary with massive processors and others' public keys can mine for all nodes. If the adversary's processors are with high clock rate, then it can always learn randomness earlier than others, or even publish its preferred random outputs only.

To avoid this, we can adopt the idea of *VRF-based mining* [69]. Verifiable Random Function (VRF) – which can be seen as a public key version of hash functions – takes the prover's secret key  $sk$  and an input  $x$ , and outputs a random string  $y$ . The prover can also generate a proof  $\pi$  that, 1)  $y$  is a valid output of  $x$ , and 2)  $y$  is generated by the owner of  $sk$ . We replace  $H(pk||S_{i+1}) \leq \frac{2^k}{T}$  with  $\text{VRFHash}(sk, S_{i+1}) \leq \frac{2^k}{T}$  in SeqPoW's difficulty mechanism, where  $\text{VRFHash}(\cdot)$  is a VRF [70]. As correct nodes do not share their secret keys to others, the adversary cannot execute  $\text{VRFHash}(\cdot)$  for others. While this modification adds negligible overhead to SeqPoW<sub>VDF</sub>, it greatly increases the proof size of SeqPoW<sub>Sloth</sub>, as the proof should carry all VRF outputs and proofs for proving no prior solution satisfies the difficulty. More efficient SeqPoW constructions with *uniqueness* are considered as future work.

### 4. Dynamic difficulty

PoW-based blockchains employ difficulty adjustment mechanism for stabilising the block rate, i.e., the average number of new blocks in a time unit. This is particularly useful when churn [71] is high and/or the network size is frequently changing. Although we analyse RANDCHAIN while assuming a fixed difficulty and a fixed set of nodes, RANDCHAIN can support dynamic difficulty adjustment with little change. First, similar to PoW-based blockchains, RANDCHAIN can include a timestamp to each block, so that RANDCHAIN can infer historical block rate using timestamps. In addition, RANDCHAIN includes the number  $i$  of iterations running SeqPoW.Solve( $\cdot$ ), and  $i$  can also infer the historical block rate. If historical values of  $i$  are large, then this means that mining is too hard and the difficulty should be reduced, and vice versa.

## 7. COMPARISON WITH EXISTING DRB PROTOCOLS

In this section, we compare RANDCHAIN with existing DRB protocols. There are three paradigms of constructing Decentralised Randomness Beacons (DRBs), namely 1) DRB from external randomness source, 2) DRB from Distributed Randomness Generation (DRG), and 3) DRB from iteratively sequential functions. RANDCHAIN – which is constructed from

SeqPoW and Nakamoto consensus – does not belong to any of them. Compared to existing paradigms, RANDCHAIN does not rely on strong assumptions such as trusted third party and lock-step synchrony, while being secure, scalable, energy efficient and decentralised.

**DRBs from external randomness source.** Some DRBs that derive randomness from real-world random entropy, including financial data, e.g., stock prices [8] and public blockchains [6], [9], [10] Such DRBs introduce little communication overhead. However, these DRB protocols' security relies on the randomness source's security. For example, if the randomness source is biasible, then these DRB protocols are likely to be biasible as well. In addition, clients and/or servers should access the randomness source from trustworthy communication channels. Otherwise, an adversary who hijacks the channels can bias the randomness arbitrarily.

**DRG-based DRBs.** A large number of DRB protocols are constructed by executing a Distributed Randomness Generation (DRG) protocol in rounds. DRG allows a group of nodes to generate a random output or a batch of random outputs. It has a well-known variant called *multi-party coin tossing/flipping* [72], where the random output is only a binary bit. DRG can be constructed from various cryptographic primitives, such as commitments [7], [73], threshold signatures [3], [43], [44], VRFs [2], [5], [74]–[76], secret sharing [1], [45], [46], [49], [50] and homomorphic encryption [47].

There are some issues in DRG-based DRBs. First, if the DRG relies on a leader, then a leader should be elected for every round. The leader is elected either by a trusted third party or running a leader election protocol. Previous research [77] shows that constructing leader election is challenging. Boneh et al. [78] propose two leader election constructions, which however rely on a RB in return. Second, if the network is not synchronous, then the DRG-based DRB should rely on a *pacemaker* [79] for liveness. Without synchrony, a node cannot know whether other nodes have received its message, and nodes may not agree on the round number. When this happens, the system would lose liveness. The pacemaker is responsible to inform nodes when to start a new round. Last, a DRG-based DRB inherits the assumption, security and performance from its used DRG. If the DRG does not scale, then DRBs based on this DRG cannot scale as well. Existing research [50], [80] shows that existing DRG protocols either rely on strong assumptions, fail to be unpredictable, or suffer from high communication complexity of more than  $O(n^2)$ . This makes DRG-based DRBs hard to scale.

**DRBs from iteratively sequential functions.** DRBs can also be constructed from an iteratively sequential function  $f(\cdot)$ . Given a random seed,  $f(\cdot)$  can produce random outputs continuously. As  $f(\cdot)$  is deterministic, no one can bias random outputs. As  $f(\cdot)$  is sequential, no one can obtain outputs without computing  $f(\cdot)$  honestly. Lenstra and Wesolowski [15] and Ephraim et. al. [81] construct DRBs from Sloth and Continuous VDFs, respectively.

The biggest challenge of this paradigm is *frontrunning*. If

one can learn the random seed earlier than others, then it can pre-compute and learn random outputs earlier than others. This can be solved by a trusted setup, e.g., the Sapling ceremony for Zcash [82]. In addition, nodes with faster processors learn random outputs earlier than others.

RandRunner [48] extends this paradigm by allowing nodes to execute iterations of  $f(\cdot)$  in turn. In each round, nodes elect a leader with the last random output as input. With the last output as input, the leader runs the *trapdoor* VDF while others run the normal VDF to obtain the next random output. Due to the trapdoor, the leader learns the next random output earlier than others. If the leader is malicious, other nodes can still learn the next random output, but slower. This does not prevent frontrunning: the leader always learns randomness earlier than others. In addition, RandRunner relies on a leader election, which as discussed can be challenging in dynamic networks.

## 8. CONCLUSION AND FUTURE WORK

In this paper, we propose RANDCHAIN, a new family of Decentralised Randomness Beacon (DRB) protocols that are simple, secure and scalable. To construct RANDCHAIN, we introduced Sequential Proof-of-Work (SeqPoW), a variant of Proof-of-Work that is sequential, i.e., non-parallelisable. For SeqPoW, we provided concrete constructions of SeqPoW, and showed that SeqPoW is practical and useful for various cryptographic protocols. For RANDCHAIN, we showed that while inheriting simplicity, security, and scalability from PoW-based consensus, RANDCHAIN remains energy efficient and decentralised.

Compared to existing DRBs, RANDCHAIN explores a new design direction with several unique design choices, which may inspire new research in designing secure decentralised randomness beacon systems.

**Competitive nodes v.s. collaborative nodes.** In RANDCHAIN, nodes are *competitive* with each other: each node tries to propose the next random output earlier than others. Meanwhile, in most existing DRBs – especially DRG-based DRBs – nodes are *collaborative*: nodes contribute their local random outputs and combine them to a unique one.

Compared to collaborative DRBs, competitive DRBs introduce less communication overhead. In competitive DRBs, each random output is generated by a single node, and a single message, i.e., a block, needs to be propagated for each random output. Meanwhile in collaborative DRBs, the majority of nodes should contribute to the randomness, and all of these nodes need to broadcast some messages. This introduces non-negligible communication overhead.

**Reusing entropy v.s. regenerating entropy.** Consider RANDCHAIN works in an ideal setting, where all nodes mine at the same speed and are correct, and all messages are delivered instantly. Given the latest block, who solves SeqPoW first is deterministic. Given the SeqPoW puzzle, the solution and the random output are also deterministic. Thus, in this ideal setting, RANDCHAIN resembles a DRB based on iteratively sequential functions, which takes the genesis block as random

seed and is run by multiple nodes in turn. As analysed in §7, DRBs based on iteratively sequential functions are strongly unpredictable, but suffer from frontrunning attacks.

**Uncontrollable entropy v.s. node-chosen entropy.** Then, consider RANDCHAIN works in real-world settings where nodes can be Byzantine and messages are delivered with random latency. Given the latest block, the next output are no longer determined. First, the node  $A$  that should have found the first SeqPoW solution may be offline or crashed. Second,  $A$  may propagate its block slower than another node  $B$  who finds a SeqPoW solution a bit later than  $A$ .

This means that in addition to the genesis block, nodes' Byzantine behaviours and network delay also affect the outputs of RANDCHAIN. The adversary cannot control both entropy sources, so frontrunning can be no longer realistic. This makes RANDCHAIN even more secure than DRBs based on iteratively sequential functions.

## REFERENCES

- [1] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*, Springer, 2017, pp. 357–388.
- [2] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 66–98.
- [3] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.
- [4] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [5] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.
- [6] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source.," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1015, 2015.
- [7] *Randao: A dao working as rng of ethereum*, last accessed on 02/08/2020. [Online]. Available: <https://github.com/randao/randao>.
- [8] J. Clark and U. Hengartner, "On the use of financial data as a random beacon.," *EVT/WOTE*, vol. 89, 2010.
- [9] M. Andrychowicz and S. Dziembowski, "Distributed cryptography based on the proofs of work.," *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 796, 2014.
- [10] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum," *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

- [11] M. Jakobsson and A. Juels, “Proofs of work and bread pudding protocols,” in *Secure information networks*, Springer, 1999, pp. 258–272.
- [12] S. Nakamoto *et al.*, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [13] “Why has cpu frequency ceased to grow?,” last accessed on 01/08/20. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/blogs/why-has-cpu-frequency-ceased-to-grow.html>.
- [14] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, “Verifiable delay functions,” in *Annual international cryptology conference*, Springer, 2018, pp. 757–788.
- [15] A. K. Lenstra and B. Wesolowski, “A random zoo: Sloth, unicorn, and trx.,” *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 366, 2015.
- [16] K. Pietrzak, “Simple verifiable delay functions,” in *10th innovations in theoretical computer science conference (itcs 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- [17] C. Dwork and M. Naor, “Pricing via processing or combatting junk mail,” in *Annual International Cryptology Conference*, Springer, 1992, pp. 139–147.
- [18] B. Wesolowski, “Efficient verifiable delay functions,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 379–407.
- [19] R. L. Rivest, A. Shamir, and D. A. Wagner, “Time-lock puzzles and timed-release crypto,” 1996.
- [20] M. Mahmoody, T. Moran, and S. Vadhan, “Publicly verifiable proofs of sequential work,” in *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, 2013, pp. 373–388.
- [21] N. Döttling, S. Garg, G. Malavolta, and P. N. Vasudevan, “Tight verifiable delay functions.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 659, 2019.
- [22] S. King and S. Nadal, “Ppcoin: Peer-to-peer cryptocurrency with proof-of-stake,” *self-published paper, August*, vol. 19, p. 1, 2012.
- [23] *Nakamoto consensus with vdf and vrf*, last accessed on 01/08/20. [Online]. Available: <https://ethresear.ch/t/nakamoto-consensus-with-vdf-and-vrf/5671>.
- [24] *Pos based on synthetic pow using vdf and vrf*, last accessed on 01/08/20. [Online]. Available: <https://ethresear.ch/t/pos-based-on-synthetic-pow-using-vdf-and-vrf/7271>.
- [25] J. Long and R. Wei, “Nakamoto consensus with verifiable delay puzzle,” *arXiv preprint arXiv:1908.06394*, 2019.
- [26] H. Abusalah, C. Kamath, K. Klein, K. Pietrzak, and M. Walter, “Reversible proofs of sequential work,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 277–291.
- [27] P. Valiant, “Incrementally verifiable computation or proofs of knowledge imply time/space efficiency,” in *Theory of Cryptography Conference*, Springer, 2008, pp. 1–18.
- [28] M. Naor, O. Paneth, and G. N. Rothblum, “Incrementally verifiable computation via incremental pcps,” in *Theory of Cryptography Conference*, Springer, 2019, pp. 552–576.
- [29] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, “Recursive composition and bootstrapping for snarks and proof-carrying data,” in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 111–120.
- [30] S. Bowe, J. Grigg, and D. Hopwood, “Halo: Recursive proof composition without a trusted setup.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1021, 2019.
- [31] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner, “Proof-carrying data from accumulation schemes.,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 499, 2020.
- [32] D. Boneh, B. Bünz, and B. Fisch, “A survey of two verifiable delay functions.,” *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 712, 2018.
- [33] V. Attias, L. Vigneri, and V. Dimitrov, “Implementation study of two verifiable delayfunctions.,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 332, 2020.
- [34] “Randchain,” [Online]. Available: ANONYMISED FOR SUBMISSION.
- [35] “Crates/rug,” last accessed on 01/08/20. [Online]. Available: <https://crates.io/crates/rug>.
- [36] “Cargo-bench,” last accessed on 01/08/20. [Online]. Available: <https://doc.rust-lang.org/cargo/commands/cargo-bench.html>.
- [37] “Criterion.rs,” last accessed on 01/08/20. [Online]. Available: <https://github.com/bheisler/criterion.rs>.
- [38] J. Garay, A. Kiayias, and N. Leonardos, “The bitcoin backbone protocol: Analysis and applications,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015, pp. 281–310.
- [39] R. Pass, L. Seeman, and A. Shelat, “Analysis of the blockchain protocol in asynchronous networks,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2017, pp. 643–673.
- [40] J. A. Garay, A. Kiayias, and N. Leonardos, “Full analysis of nakamoto consensus in bounded-delay networks.,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 277, 2020.
- [41] L. Ren, “Analysis of nakamoto consensus.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 943, 2019.
- [42] B. Y. Chan and E. Shi, “Streamlet: Textbook streamlined blockchains.,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 88, 2020.
- [43] C. Cachin, K. Kursawe, and V. Shoup, “Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography,” *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.
- [44] E. Kokoris-Kogias, A. Spiegelman, D. Malkhi, and I. Abraham, “Bootstrapping consensus without trusted

- setup: Fully asynchronous distributed key generation.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1015, 2019.
- [45] I. Cascudo and B. David, “Scrape: Scalable randomness attested by public entities,” in *International Conference on Applied Cryptography and Network Security*, Springer, 2017, pp. 537–556.
- [46] —, “Albatross: Publicly attestable batched randomness based on secret sharing,”
- [47] A. Cherniaeva, I. Shirobokov, and O. Shlomovits, “Homomorphic encryption random beacon.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1320, 2019.
- [48] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. Weippl, *Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness*, Cryptology ePrint Archive, Report 2020/942, <https://eprint.iacr.org/2020/942>, 2020.
- [49] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, “Scalable bias-resistant distributed randomness,” in *2017 IEEE Symposium on Security and Privacy (SP)*, Ieee, 2017, pp. 444–460.
- [50] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, “Hydrand: Efficient continuous distributed randomness,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 32–48.
- [51] J. Garay, A. Kiayias, and N. Leonardos, “The bitcoin backbone protocol with chains of variable difficulty,” in *Annual International Cryptology Conference*, Springer, 2017, pp. 291–323.
- [52] L. Kiffer, R. Rajaraman, and A. Shelat, “A better method to analyze blockchain consistency,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 729–744.
- [53] A. Dembo, S. Kannan, E. N. Tas, D. Tse, P. Viswanath, X. Wang, and O. Zeitouni, “Everything is a race and nakamoto always wins,” *arXiv preprint arXiv:2005.10484*, 2020.
- [54] P. Gazi, A. Kiayias, and A. Russell, “Tight consistency bounds for bitcoin,” 2020.
- [55] “Bitcoin’s energy consumption ‘equals that of switzerland’,” last accessed on 01/08/20. [Online]. Available: <https://www.bbc.com/news/technology-48853230#:~:text=Bitcoin%20uses%20as%20much%20energy,the%20University%20of%20Cambridge%20shows.&text=Currently%2C%20the%20tool%20estimates%20that,0.21%25%20of%20the%20world’s%20supply..>
- [56] *Amd breaks 8ghz overclock with upcoming fx processor, sets world record*, last accessed on 01/08/20. [Online]. Available: <http://hothardware.com/News/AMD-Breaks-Frequency-Record-with-Upcoming-FX-Processor/>.
- [57] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” in *International conference on financial cryptography and data security*, Springer, 2014, pp. 436–454.
- [58] J. Bonneau, E. W. Felten, S. Goldfeder, J. A. Kroll, and A. Narayanan, “Why buy when you can rent? bribery attacks on bitcoin consensus,” 2016.
- [59] K. Liao and J. Katz, “Incentivizing blockchain forks via whale transactions,” in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 264–279.
- [60] A. Judmayer, N. Stifter, A. Zamyatin, I. Tsabary, I. Eyal, P. Gazi, S. Meiklejohn, and E. R. Weippl, “Pay-to-win: Incentive attacks on proof-of-work cryptocurrencies.,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 775, 2019.
- [61] V. Buterin and V. Griffith, “Casper the friendly finality gadget,” *arXiv preprint arXiv:1710.09437*, 2017.
- [62] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 566–583.
- [63] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
- [64] D. Malkhi and M. Reiter, “Byzantine quorum systems,” *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.
- [65] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.
- [66] T.-H. H. Chan, R. Pass, and E. Shi, “Consensus through herding,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 720–749.
- [67] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, “Scalable and probabilistic leaderless bft consensus through metastability,” *arXiv preprint arXiv:1906.08936*, 2019.
- [68] “Bch avalanche transactions show finality speeds 10x faster than ethereum,” last accessed on 01/08/20. [Online]. Available: <https://news.bitcoin.com/bch-avalanche-transactions-show-finality-speeds-10x-faster-than-ethereum/>.
- [69] “Vrf-based mining: Simple non-outsourcable cryptocurrency mining,” last accessed on 01/08/20. [Online]. Available: <https://github.com/DEX-ware/vrf-mining/blob/master/paper/main.pdf>.
- [70] S. Micali, M. Rabin, and S. Vadhan, “Verifiable random functions,” in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, IEEE, 1999, pp. 120–130.
- [71] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ACM, 2006, pp. 189–202.

- [72] M. Blum, “Coin flipping by telephone a protocol for solving impossible problems,” *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.
- [73] B. Awerbuch and C. Scheideler, “Robust random number generation for peer-to-peer systems,” in *International Conference On Principles Of Distributed Systems*, Springer, 2006, pp. 275–289.
- [74] Y. Dodis, “Efficient construction of (distributed) verifiable random functions,” in *International Workshop on Public Key Cryptography*, Springer, 2003, pp. 1–17.
- [75] V. Kuchta and M. Manulis, “Unique aggregate signatures with applications to distributed verifiable random functions,” in *International Conference on Cryptology and Network Security*, Springer, 2013, pp. 251–270.
- [76] D. Galindo, J. Liu, M. Ordean, and J.-M. Wong, “Fully distributed verifiable random functions and their application to decentralised random beacons,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 96, 2020.
- [77] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. Raynal, “Fault-tolerant leader election in mobile dynamic distributed systems,” in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, IEEE, 2013, pp. 78–87.
- [78] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, “Single secret leader election,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 25, 2020.
- [79] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus with linearity and responsiveness,” in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.
- [80] G. Wang, Z. J. Shi, M. Nixon, and S. Han, “Sok: Sharding on blockchain,” in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 41–61.
- [81] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass, “Continuous verifiable delay functions,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2020, pp. 125–154.
- [82] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, “Zcash protocol specification,” *GitHub: San Francisco, CA, USA*, 2016.

## APPENDIX

### 1. Security proof of two SeqPoW constructions

We formally prove security guarantee of two SeqPoW constructions. We start from SeqPoW<sub>VDF</sub>.

**Lemma 5.** SeqPoW<sub>VDF</sub> satisfies completeness.

*Proof.* Assuming there is an  $(\lambda, \psi, T)$ -valid tuple  $(pp, sk, i, x, S_i, \pi_i)$ . By completeness and Lemma 1, VDF.Verify( $\cdot$ ) will pass. As hash functions are deterministic, difficulty check will pass. Therefore,

$$\text{SeqPoW}_{\text{VDF}}.\text{Verify}(pp, pk, i, x, S_i, \pi_i) = 1$$

**Lemma 6.** SeqPoW<sub>VDF</sub> satisfies soundness. □

*Proof.* We prove this by contradiction. Assuming there exists a tuple  $(pp, sk, i, x, S_i, \pi_i)$  that is not  $(\lambda, \psi, T)$ -valid such that

$$\text{SeqPoW}_{\text{VDF}}.\text{Verify}(pp, pk, i, x, S_i, \pi_i) = 1$$

By soundness and Lemma 1, if  $(y, y^+, \pi^+, \psi)$  is generated by  $\mathcal{A}$ , VDF.Verify( $\cdot$ ) will return 0. As hash functions are deterministic, if  $S_i > \frac{2^{\kappa}}{T}$ , difficulty check will return 0. Thus, if  $(pp, sk, i, x, S_i, \pi_i)$  is not  $(\lambda, \psi, T)$ -valid, then the adversary should break soundness. Thus, this assumption contradicts soundness. □

**Lemma 7.** SeqPoW<sub>VDF</sub> satisfies hardness.

*Proof.* We prove this by contradiction. Assuming

$$\Pr \left[ b_{i+1} = 1 \mid \begin{array}{l} S_{i+1}, b_{i+1} \leftarrow \\ \text{Solve}(pp, sk, T, S_i) \end{array} \right] > \frac{1}{T}$$

By sequentiality, the value of  $S_{i+1}$  is unpredictable before finishing Solve( $\cdot$ ). By pseudorandomness of hash functions,  $H(pk \| S_{i+1})$  is uniformly distributed, and the probability that  $H(pk \| S_{i+1}) \leq \frac{2^{\kappa}}{T}$  is  $\frac{1}{T}$  with negligible probability. This contradicts the assumption. □

**Lemma 8.** SeqPoW<sub>VDF</sub> does not satisfy uniqueness.

*Proof.* By hardness, each of  $S_i$  has the probability  $\frac{1}{T}$  to be a valid solution. As  $i$  can be infinite, with overwhelming probability, there exists more than one honest tuple  $(pp, sk, i, x, S_i, \pi_i)$  such that  $H(pk \| S_i) \leq \frac{2^{\kappa}}{T}$ . □

**Lemma 9.** If the underlying VDF satisfies  $\sigma$ -sequentiality, then SeqPoW<sub>VDF</sub> satisfies  $\sigma$ -sequentiality.

*Proof.* We prove this by contradiction. Assuming there exists  $\mathcal{A}_1$  which runs in less than time  $\sigma(\psi i)$  such that

$$\Pr \left[ \begin{array}{l} (pp, sk, i, x, S_i, \pi_i) \\ \in \mathcal{H} \end{array} \mid \begin{array}{l} pp \leftarrow \text{Setup}(\lambda, \psi, T) \\ (sk, pk) \xleftarrow{R} \text{Gen}(pp) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_0(\lambda, pp, sk) \\ S_i \leftarrow \mathcal{A}_1(i, x) \\ \pi_i \leftarrow \text{Prove}(pp, sk, i, x, S_i) \end{array} \right]$$

By  $\sigma$ -sequentiality,  $\mathcal{A}_1$  cannot solve VDF.Eval( $pp_{\text{VDF}}, y, \psi$ ) within  $\sigma(\psi)$ . By Lemma 1,  $S_i$  can and only can be computed by composing VDF.Eval( $pp_{\text{VDF}}, y, \psi$ ) for  $i$  times, which cannot be solved within  $\sigma(\psi i)$ . This contradicts the assumption. □

The completeness, soundness, hardness and sequentiality proofs of SeqPoW<sub>Sloth</sub> are identical with SeqPoW<sub>VDF</sub>'s. We prove SeqPoW<sub>Sloth</sub> satisfies uniqueness below.

**Lemma 10.** SeqPoW<sub>Sloth</sub> satisfies uniqueness.

*Proof.* We prove this by contradiction. Assuming there exists two  $(\lambda, \psi, T)$ -valid tuples  $(pp, sk, i, x, S_i, \pi_i)$  and  $(pp, sk, j, x, S_j, \pi_j)$  where  $j < i$ . According to SeqPoW<sub>Sloth</sub>.Solve( $\cdot$ ), we have  $H(pk \| S_i) \leq \frac{2^{\kappa}}{T}$



and  $H(pk||S_j) \leq \frac{2^n}{T}$ , and initial difficulty check in  $\text{SeqPoW}_{\text{Sloth}}.\text{Verify}(\cdot)$  will pass. However, in the for loop of  $\text{SeqPoW}_{\text{Sloth}}.\text{Verify}(\cdot)$ , if  $S_i$  is valid, then verification of  $S_j$  will fail. Then,  $\text{SeqPoW}_{\text{Sloth}}.\text{Verify}(\cdot)$  returns 0, which contradicts the assumption.  $\square$