RandChain: Practical Scalable Decentralized Randomness Attested by Blockchain

Gang Wang*, Mark Nixon†
*University of Connecticut
†Emerson Automation Solutions
Email: gang.wang@uconn.edu

***Abstract*—Reliable and verifiable public randomness is not only an essential building block in various cryptographic primitives, but also is a critical component in many distributed and decentralized protocols, e.g., blockchain sharding. A 'good' randomness generator should preserve several distinctive properties, such as public-verifiability, bias-resistance, unpredictability, and availability. However, it is a challenging task to generate such good randomness. For instance, a dishonest party may behave deceptively to bias the final randomness, which is toward his preferences. And this challenge is more serious in a distributed and decentralized system. Blockchain technology provides several promising features, such as decentralization, immutability, and trustworthiness. Due to extremely high overheads on both communication and computation, most existing solutions face an additional scalability issue. We propose a sharding-based scheme, *RandChain*, to obtain a practical scalable distributed and decentralized randomness attested by blockchain in large-scale applications. In RandChain, we eliminate the use of computation-heavy cryptographic operations, e.g., Publicly Verifiable Secret Sharing (PVSS), in prevalent approaches. We build a sub-routine, *RandGene*, which utilizes a *commit-then-reveal* strategy to establish a local randomness, enforced by efficient Verifiable Random Function (VRF). RandGene generates the randomness based on statistical approaches, instead of cryptographic operations, to eliminate computational operations. RandChain maintains a two-layer hierarchical chain structure via a sharding scheme. The first level chain is maintained by RandGene within each shard to provide a verifiable randomness source by blockchain. The second level chain uses the randomnesses from each shard to build a randomness chain.**

I. Introduction

A reliable randomness engine is an essential component in various distributed protocols (e.g., blockchain protocols). The reliable randomness is of importance and highly used in modern cryptography domains [1], e.g., generating a high-entropy ciphertext. The *coin tossing* protocol [2] was first introduced to achieve randomness, which enables the property that multiple parties can generate a uniform randomness output as long as there exists at least one honest party. As the popularity of distributed and decentralized ledgers [3] [4] [5], the coin-tossing protocol as a randomness generator has gained great attention. However, a coin-tossing scheme typically relies on a trusted authority, which is further based on a centralized scheme. Generating a trustworthy and reliable randomness in a distributed and decentralized manner is still a complementary problem.

Random beacon protocols are the foundation of many practical protocols, and can be applied in various applications. It is desirable to generate public randomness in many scenarios without resorting to a trusted third authority. However, it is challenging to obtain public unbiased randomness, as dishonest parties may behave deceptively to bias the final randomness towards their preferences. This can be done, for example, by selectively injecting and providing error-prone input values or by manipulating their own inputs at the final stage. These issues can be handled on a small scale; however, it is a challenging task on a large-scale to achieve an enhanced security level.

Postprint online version of paper published in proceeding of 2020 IEEE International Conference on Blockchain (Blockchain'20). DOI https://doi.org/10.1109/Blockchain50366.2020.00064

In this paper, we present *RandChain*, a practical scalable decentralized random beacon protocol, attested by blockchain. RandChain is a decentralized random beacon protocol designed to provide *continuous* randomness at regular intervals and provides guaranteed output delivery. To build RandChain, we design a randomness generation protocol via VRF, *RandGene*, which is a consensus protocol used to establish the single randomness values in a small scale. RandChain is a public randomness beacon protocol that can generate a series of random outputs at a regular pace based on the RandGene protocol. Roughly speaking, both RandGene and RandChain provide a same level of security on some critical properties, e.g., availability, unpredictability, bias-resistance, and third-party verifiability. The overall design follows a two-layer hierarchical blockchain based on the concept of sharding. The first layer is formed by RandGene, which serves as an attestation to RandChain; while the second layer builds a blockchain on all distinct shards, and each shard contributes randomness based on its local blockchain.

The rest of the paper is organized as follows. Section II provides the preliminaries and motivations. Section III describes the RandGene and RandChain protocols. Section IV provides the analysis of the proposed protocol. Section V concludes this paper.

II. Preliminaries and Motivation

A. Good Randomness

Along with the advances of various cryptographic primitives, obtaining a reliable randomness source becomes more and more critical. In general, this can be fulfilled by utilizing local randomness, e.g., relying on some trusted agents. However, establishing such a common randomness generally relies on trusted dealers, at least for the initial setup [6], which can potentially bias the system and make the system subject to single-failure attacks. The goal of *random beacon protocols* is to generate reliable, bias-resistant, publicly-verifiable, and unpredictable random values at a regular interval in a distributed and decentralized manner. When introducing random beacon protocols into decentralized applications, several unique properties are still required to guarantee to be a *good* random beacon protocol [7] [8] [9]:

1). *Public-Verifiability:* An external party, e.g., the one who even did not actively participate in the generation process, still has the ability to verify the validity of the generated randomness. This means any party has the ability to verify the validity of the generated randomness simply by resorting to publicly available information.

2). *Bias-Resistance:* This property is to assure that the final randomness is a uniform and unbiased random value. This means any single party or multiple colluding parties (e.g., controlled by an active adversary) do not have the ability to influence the future randomness to its favor.

3). *Unpredictability:* Any party (either honest or adversarial) should not have the ability to pre-compute or predict the future randomness in advance. This means no party learns anything about the final randomness before the generation process is completed.

4). *Availability:* This property indicates that any single party or multiple colluding parties (e.g., controlled by an active adversary) should not have the ability to prevent the progress of randomness generation.

B. Motivation

Coin-tossing protocols have been used to generate randomness; however, they require a centralized "bulletin board", which is not practical and infeasible in many decentralized systems. Another shortcoming of existing randomness generators is that the scalability is very low, where the number of participants is limited in a small range. A scalable randomness scheme is desired to fit large-scale distributed systems. The third shortcoming of the current distributed randomness schemes is that most of them primarily rely on complex cryptographic operations, such as Verifiable Secret Sharing (VSS) or Publicly Verifiable Secret Sharing (PVSS). The computation and communication complexities of these cryptographic operations are extremely high. And, the cryptography-based schemes are time-consuming to perform cryptographic operations. For instance, a recent work *RandHound* takes more than 240 seconds to generate randomness and 76 seconds to verify it using the produced 4 Mbyte transcript [9].

Comparing with VSS or PVSS, Verifiable Random Functions (VRF) [10] is a lightweight cryptographic tool. For example, a typical PVSS runs in three steps: (1) secrete generation and distribution; (2) share verification, encryption and decryption; and (3) shares verification and combination. When applying PVSS into a real randomness generation protocol, it may involve more steps, e.g., RandHound taking six-step to generate one randomness, and each step involves complex computation. While VRF will not have this to provide verifiable service. If we assume there exists a random oracle model, and we can simplify VRF construction as a hash function $H(\bullet)$. $H(\bullet)$ is a simple VRF under the random oracle model. The only obstacle is to design a robust consensus protocol to get good randomness based on VRF. This is the goal of this paper.

It is critical to developing a practical scalable continuous decentralized randomness scheme, which can be used for large-scale scenarios. In this paper, we propose to utilize blockchain as the "trusted entity" to generate good randomness.

C. Models

1) System Model: RandChain can work on untrusted networks, in which malicious nodes can perform arbitrary behaviors on messages, e.g., drop or delay messages. We assume the overall communication network is based on the synchrony, as defined in [11], in which there exists a fixed bound, Δ, on a message's traversal. We assume each node i has a public/private key pair (pk_i, sk_i), and the public key pk_i is used to identify its identity. All messages communicated in the network are authenticated with the sender's private key. In addition, we made similar assumptions in previous work, e.g. [12] [13] [14]. For example, our network model has assumption that (a) the network connections among the honest nodes are well connected and authenticated, and (b) the communication channels among the honest nodes are synchronous. However, we do not require the order of the messages to their destinations to be preserved.

2) Threat Model: In our threat model, we distinguish between honest nodes and Byzantine nodes. We consider a replica (node) as honest if it follows the protocol faithfully and behaves honestly. For a Byzantine node, we do not have control over its behavior, and the node may behave arbitrarily. Our threat model considers a Byzantine adversary who has an ability to corrupt any $f < n/3$ of nodes at the beginning of the protocol run. We assume that once a node broadcasts a message (even a wrong message), this message will arrive to all participating nodes. Also, the Byzantine adversary is a probabilistic polynomial-time adversary with bounded computation capacity, which cannot break any established cryptographic primitives.

III. RandChain: Randomness Beacon Protocol

This section presents the proposed *RandChain*, a scalable, verifiable randomness beacon.

A. Protocol Overview

RandChain seeks to offer a stand-alone randomness generator that is bias-resistant, unpredictable, and publicly verifiable, which can emit random values at a predetermined and regular time interval. We aim the RandChain, with a fixed number of participating nodes, at a permissioned blockchain setting (e.g., requiring authentication and authorization for membership). Especially, RandChain uses a *commit-then-reveal* mechanism to generate randomness, with the help of VRF, and it then uses a Byzantine Fault Tolerant (BFT) protocol to build a blockchain to secure the protocol's output. The blockchain is used to attest agreement on generated randomness among all honest participants.

RandChain employs a two-layer hierarchical blockchain structure, in which the sharding technology is adopted to achieve this hierarchical blockchain. The participating nodes are uniformly sharded into distinct committees (alternatively called "shards"), and each committee maintains its own local blockchain. The local blockchain runs a *RandGene* protocol, which is used to generate a "good" local randomness (defined in Section II-A) and record this local randomness into its local blockchain. Then, all committees together form a RandChain, which serves as a randomness source for all participating nodes. RandChain utilizes the concept of *epoch*, and each epoch generates one randomness. Compared with traditional distributed randomness generation protocols, we eliminate the use of computational-heavy cryptographic primitives, such as PVSS and threshold signatures. Instead, we use a statistical approach to obtain local randomness, which reduces both computation and communication complexities. To achieve this, we split each epoch E into k sub-epoch e_i, where $0 \leq i \leq k$ and $\sum_k e_i = E$. The process of generating randomness is based on a statistical scheme $RandGene$. We have two types of epochs: one is the sub-epoch e within each shard (the basic unit for the second-layer blockchain, shard blockchain); the other is the epoch E for overall shards (the basic unit for the first-layer blockchain, RandChain). Within an epoch E, each shard will have a local block generated, consisting of the agreed-upon "good" randomness. Fig. 1 shows an overview of the proposed RandGene protocol.

B. Description of RandGene

The RandGene protocol proceeds in rounds where each round consists of six distinct phases[1]: *propose, pre-commit, release, re-*

[1]One round represents one epoch E to generate a local block, and each phase might consist of a different number of sub-epoch e.

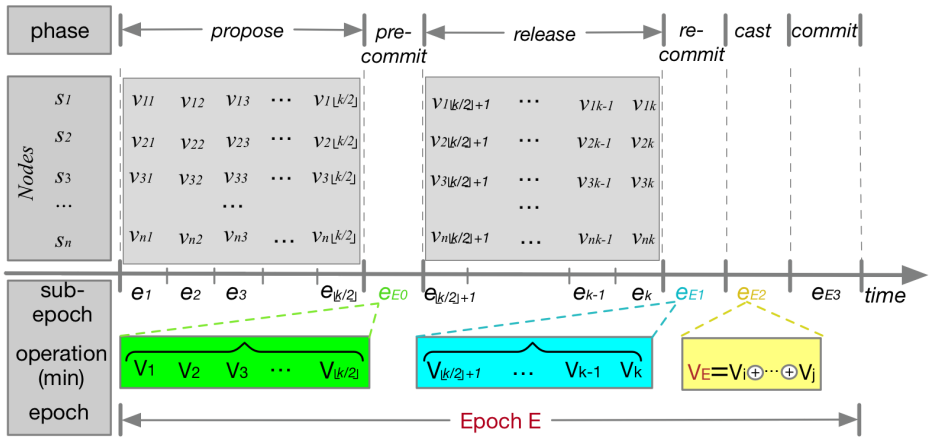

Fig. 1. Overview of RandGene Protocol, Consisting of $k+4$ Sub-epochs and Six Phases: propose, pre-commit, release, re-commit, cast and commit.

commit, cast, and *commit*. Each round is counted by epoch E, and each epoch E consists of $k+4$ equal-sized sub-epochs. To simplify the expression in RandGene, in *this* section, the terms "sub-epoch" and "epoch" have the same meaning e, and are exchangeable. In $k+4$ epochs, k epochs are used to propose random values by participating nodes, where each final randomness consists of k random values from each node. Let $S = \{s_1, s_2, ..., s_n\}$ denote the list of nodes, and node s_i has a public/private key pair (pk_i, sk_i). Let v_{ij} denote the random value committed by node s_i in sub-epoch e_j, and V_j denotes the randomness in sub-epoch e_j after the *pre-commit* / *re-commit* phases. The final random values proposed by nodes have two parts sequentially. *Pre-commit* phase deals with the first half of random values proposed by nodes, while *re-commit* phase commits the second half of random values. Fig. 1 shows an overview of RandGene protocol. For each participating node, it runs a six-phase protocol to get randomness, and these phases can be pipelined to improve efficiency.

RandGene is based on a well-established VRF. In VRF, for any input string, say x, VRF returns two components: one is a hash value h and the other is the proof π on h. In our design, string x consists of the id of the node, sub-epoch information as well as the randomness from the previous epoch, $x_{ij} = id_{s_i}||e_j||R_E$, where x_{ij} is the message proposed by node s_i during sub-epoch e_j, and R_E is the randomness of the previous epoch E. The construction of string x is critical to our design, so that everyone can publicly verify this message. We describe the six-phase RandGene as follows.

1) Propose: In *propose* phase, node s_i accesses VRF, by providing the string x, to obtain its random value v_{ij} for the sub-epoch e_j, where $i \in \{1, ..., n\}$, $j \in \{1, ..., \lfloor k/2 \rfloor\}$. This value v_{ij} is to be committed during the sub-epoch e_j by node s_i, and v_{ij} will be sent to all other nodes. Initially, each node marks each v_{ij} as a infinity value (e.g., $+\infty$). Upon receiving a message, the node must check the validity of each received value v_{ij} by verifying two aspects: (a) the value is indeed sent from its sender (e.g., without modification); and that (b) the value is indeed obtained from VRF[2] (e.g., public verifiability). If the value v_{ij} is valid, then the node will treat it as a valid committed value for the pre-commit phase, and updates the value v_{ij}. The propose phase occupies $\lfloor k/2 \rfloor$ sub-epochs out of $k+4$ sub-epochs, and each sub-epoch processes in sequential.

The string x provided by node s_i consists of its sub-epoch information e_j and identification information i. Thus, it is a trivial task to distinguish the corresponding epoch that a value v_{ij} belongs to.

2) Pre-commit: In *pre-commit* phase, each node needs to independently calculate the committed values V during sub-epoch e_1 and $e_{\lfloor k/2 \rfloor}$ from all received random values (including its own value). Each node s_i only needs to provide one random value, in each sub-epoch, and this can be guaranteed by verifying the string x_{ij}, which consists of the sub-epoch number j. The calculated values during sub-epoch e_1 and $e_{\lfloor k/2 \rfloor}$ are only the *potential* values and some of which may not be selected to be included to the final randomness. Without loss of generality, we calculate the minimum value of v_{ij} (where $i \in \{1, .., n\}$) as the committed value V_j during sub-epoch e_j (where $j \in \{1, .., \lfloor k/2 \rfloor\}$):

$$V_j = \min_{i \in \{1,...,n\}} v_{ij}$$

This pre-commit phase happens during the sub-epoch $e_{\lfloor k/2 \rfloor+1}$ to commit randomness during the *propose* phase (consisting of the first $\lfloor k/2 \rfloor$ sub-epochs). After *pre-commit* phase, the values V_j ($j \in \{1, .., \lfloor k/2 \rfloor\}$) are as the commitment for the first half of randomness.

3) Release: Once the pre-commit phase is finished, all values V_j before the sub-epoch $e_{\lfloor k/2 \rfloor+1}$ are committed and confirmed. We still need another half randomness to construct the whole randomness for the epoch E. The *release* phase consists of the sub-epochs from $e_{\lfloor k/2 \rfloor+1}$ to e_k (as shown in Fig. 1), which is used to release the remaining random values from each node. The operations in this release phase are similar to those in the propose phase. This phase is as a kind of *future* commitment, for the unpredictability of the final randomness.

4) Re-commit: This *re-commit* phase is exactly same as the *pre-commit* phase, with only one exception: this phase operates on the random values between sub-epochs $e_{\lfloor k/2 \rfloor+1}$ and e_k, while pre-commit phase operates on sub-epochs before $e_{\lfloor k/2 \rfloor+1}$.

After *re-commit* phase, each node obtains all potential random values which are used to construct the final randomness for epoch E. However, not all committed random values from sub-epoch e_1 to e_k are used for the final randomness. We select only part of them for epoch E.

5) Cast: The goal of the *cast* phase is to calculate the final randomness for epoch E. Until this phase, each node has k random

[2]The string x to VRF is known to every node, and the proof π can verify the validity of v_{ij}.

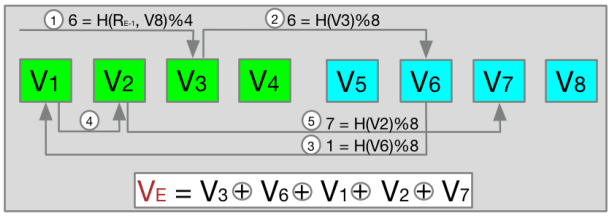

Fig. 2. Example of the *Cast* Phase of RandGene Protocol, where $k = 8$.

values V_j, coming from the previous k sub-epochs. Each node needs to independently select $\lfloor k/2 \rfloor + 1$ out of k randomnesses to construct the final randomness. According to the pigeonhole theorem, this guarantees that at least one random value comes from the commitment of *re-commit* phase.

This phase will utilize a publicly agreed randomness attested by the blockchain. From the final randomness of previous epoch $E-1$, e.g., R_{E-1} in blockchain, each node needs to calculate the starting index of the first random value (among k values V_j) which contributes to the epoch E, e.g., index l calculated by $l = H(R_{E-1}||V_k) \ mod \ \lfloor k/2 \rfloor$ [3] (where V_k is the random value of the sub-epoch e_k in *re-commit* phase), which guarantees that the first random value is in the first half of the *committed* randomnesses. And the input value of V_k to l is used to guarantee that the first random value is not a pre-calculated value before the end of *pre-commit* phase. Then, V_l will be the first random value contributed to the final randomness. The second random value will be $V_m = H(V_l) \ mod \ k$, following the same procedure ($H(\cdot) \ mod \ k$) until the node selects $\lfloor k/2 \rfloor + 1$ *distinct* random values. The selected $\lfloor k/2 \rfloor + 1$ distinct random values will be calculated via the XOR (w.r.t., $\oplus$) operation together to form a final randomness V_E for epoch E. Finally, each node sends its calculated final randomness to all other nodes.

Fig. 2 shows an example of the *cast* phase of RandGene protocol when the number of sub-epochs $k = 8$. In this example, the index l is 3, so the value V_3 is selected as the first randomness to contribute the final randomness. Following the operations provided in *cast* phase, each node will get a final randomness V_E.

6) Commit: In *commit* phase, the node receives the final randomness from all other nodes with a full verifiable proof, e.g., its selected $\lfloor k/2 \rfloor + 1$ *distinct* randomnesses. It first verifies the validity of each received randomness. Only validated final randomness can be counted into the commit phase. Only if at least $1/2$ of the total nodes have the same final randomness, can the node then add this final randomness to its local blockchain.

Put All Together: The six-phase RandGene protocol is used to generate a randomness per epoch E, which will be appended into shard chains as a block. RandGene protocol itself can be implemented as a consensus protocol to form a blockchain to record the randomness. Our overall RandChain is used to provide a continuous randomness beacon based on the local chains, and Section III-C will present how to build this randomness beacon.

We need to emphasize that each sub-epoch of RandGene proceeds in sequence, e.g., sub-epoch e_{i+1} proceeded just after sub-epoch e_i, where $1 \leq i \leq k+3$ in one round [4]. In traditional randomness generation, to bias a final result, the adversary can perform a grinding attack [5] [15], in which an adversary can try multiple times to get a solution that can benefit its choice. For example, the adversary may prepare multiple random values v_{ij}, and chooses the one which can bias the final result. Our scheme can prevent this kind of attack by the design of string x_{ij} and the system model. Each sub-epoch can easily verify if a presented random value v_{ij} is in the right epoch, and the recipient only processes the random value targeted to that sub-epoch and discards all other random values. The synchronous model guarantees that the adversary cannot go back to choose its potential random values for any previous sub-epochs.

[3] H is a random oracle. If $l = 0$, then the index is $\lfloor k/2 \rfloor$, which represents the last sub-epoch of the *propose* phase

[4] The sub-epochs here include the functional sub-epochs, e.g., $e_{E1}, e_{E2}, e_{E3}, e_{E4}$ in Fig. 1.

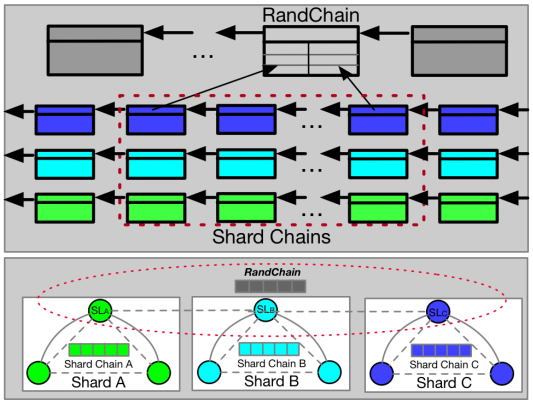

Fig. 3. Overview on the RandChain Protocol Design. The upper shows on the contraction of chains, and the lower part shows the connections among participating nodes.

The string x_{ij} contains the identity information of the node i. If a node receives multiple distinct random values from the same node, e.g., s_m, with a correct signature for one sub-epoch, then the node will discard all random values from the node s_m. When a node detects this kind of attack, it can optionally mark this node as a malicious node, since the communicated message cannot be modified by any intermediate nodes with a correct signature.

C. Building RandChain

This section introduces RandChain, a blockchain-based protocol to generate unbiasable and verifiable randomness. RandChain serves as a decentralized randomness beacon which builds on RandGene protocol.

1) Why Sharding Scheme: With the pervasiveness of large-scale distributed applications, one common issue is how to obtain good randomness on a large-scale. RandChain adopts the concept of blockchain sharding, which distributes the overall processing overhead among multiple, smaller sets of participating nodes (alternatively called *shards*). By integrating a sharding scheme into randomness generation protocol, RandChain improves scalability by sharing random values, generated by participating nodes, not directly among all nodes, but only within a smaller set of nodes. Also, picking a smaller shard leads to lower network latency and bandwidth consumption, which can effectively reduce the communication and computation complexity on each participating node.

Each node in RandChain only needs to share random values with its respective shard members, decreasing the communication and computation overhead from $O(n^2)$ to $O(cn)$, where n is the total number of participating nodes and c is the average size of a shard. Detailed analysis on this refers to Section IV-C. Different from

a generic sharding scheme, in which each shard maintains its own distinct chain independently from others, RandChain scheme aims to achieve a randomness beacon among all participating nodes. To achieve this purpose, we adopt a two-layer hierarchical blockchain structure: the local chains and the RandChain. The local chain is maintained by each distinct shard, the RandChain is maintained by all distinct shards, and each shard is one of its members. Fig. 3 shows an overview of this hierarchical RandChain design. To reduce the communication overhead among shards, we need to elect a leader, as the representative of that shard, to participate in the formation of the RandChain among distinct shards. Also, we need a consensus protocol to achieve an agreement between them. We assume each shard has been estimated at the beginning of epoch, based on the previous randomness of RandChain. The sharding process can be simplified to the problem of assigning n nodes into m groups, so that each group has a nearly identical size, say $c = n/m$ nodes.

2) Leader Election for Shard: In our shard-based hierarchical structure, each shard as a whole can be considered as one participating node of RandChain, as shown in Fig. 3. However, each shard has multiple participating nodes; thus, we need to elect a leader to represent its shard to propose the shard's choice. Each shard leader is considered as a committee member of RandChain. And, the randomness recorded in RandChain is as epoch randomness for the sharding scheme to assign nodes into distinct shards. This is critical for any shard-based schemes [16].

At the beginning of each round r ($r \geq 1$), a node i learns the shard's current configuration C, based on the available local information it has gathered so far, e.g., listing the public keys of participating nodes in its shard. We can typically consider this configuration C as a *view* of the current status of the shard. With publicly-available randomness, e.g. R_{r-1} in previous consensus round, a node i can deterministically select a leader from a group of potential leaders. We need to establish this potential list. The leader election process is very similar to the lottery process. When RandChain starts, each node generates a lottery token $t_i = H(R_{r-1}||C||X_i)$ for every $i \in N$, where X_i is the public key of node i, and sorts those tokens in an ascending order. It is necessary to note that those tokens are verifiable and publicly-available to all shard members. We choose the winner of this lottery process as the one with the *lowest* lottery token value t_i. And the node i with this lowest lottery token value will become a *tentative* RandGene leader for its shard. However, in case that this tentative leader becomes unavailable or provides forged information, the leadership will proceed to the next available node in ascending order of the lottery token list.

This selects a leader for each shard to represent as a shard to participate in the construction of RandChain. Our RandChain protocol is based on a BFT protocol, taking the advantages of instant finality.

3) RandChain: After a BFT leader is selected for RandChain, each shard leader will broadcast its shard-block (containing the generated randomness of RandGene). The BFT leader will decide which blocks are included in the BFT consensus process. To make this block proposed by BFT leader publicly verifiable, we require that the BFT leader perform VRF function on each shard-block. The hash values from VRF are sorted in ascending order, and the BFT leader needs to select k smallest hash values, where $k = (n/c)/3 + 1$ (n is the total number of nodes and c is the average size of shard)[5], to construct the final block. The BFT leader will create a transcript containing all shard-blocks received and this final block and broadcast it. Then, each participating node follows the consensus process to reach an agreement on this final block. If errors happen on the consensus process, it will start a view-change protocol. Finally, the BFT protocol will get epoch randomness.

After epoch randomness is obtained and appended into RandChain, this epoch randomness can be used for the sharding process to assign nodes into distinct shards. And each shard runs RandGene protocol to get new randomness for that shard. However, in practice, we can run multiple rounds of RandGene and then run once RandChain to start the sharding process.

[5]We assume the adopted BFT is a $(3f + 1)$-based protocol.

IV. Security Properties

This section shows that RandGene achieves the desirable properties as a random beacon protocol, outlined in Section II-A.

A. Good Randomness

Definition 1 (Good Randomness): We say an epoch has a good randomness, with high probability, if it meets: (i) availability; (ii) unpredictability; (iii) bias-resistance; and (iv) public-verifiability.

We need to guarantee that *RandGene* protocol meets these properties to be good randomness. We now prove these necessary properties.

Lemma 1 (L1:Availability): For an honest node, the protocol successfully completes and produces a final randomness block with high probability, even in the presence of a malicious adversary.

Proof: According to the overall design of RandChain, the participating nodes are randomly assigned to distinct shards based on publicly available information, such as the previous randomness block of RandChain and the node identity. Simply, we first assume the participating nodes have already been randomly assigned to distinct shards. We aim to achieve availability (or the guaranteed output delivery) of final randomness based on *RandGene*. According to the proof in Section IV-B, the probability of honest nodes in each shard is greater than $1/2$ with high probability, which means the number of honest nodes is greater than the number of malicious nodes in each shard with high probability.

From the sharding process, we can get each shard with at most $1/2$ malicious nodes with high probability. We need to prove that each secure shard meets the availability at the end of the protocol. Within each shard, the honest nodes always follow the designed protocol and show these randomness shares. And RandGene protocol is based on a synchrony setting, in which the randomness shares from honest nodes will be guaranteed to be delivered among honest nodes. If an honest node knows the randomness value R_{E-1}, it will make satisfactory progress and eventually reach at the end of phase 6 *Commit*. RandGene is a leaderless consensus protocol. As long as at least $1/2$ honest nodes agree on the final randomness, and the protocol will have a final randomness output. From the sharding process, we have at least $1/2$ honest nodes in each shard with high probability. Thus, the shard will output final randomness with high probability. ■

Lemma 2 (L2:Unpredictability): Unpredictability ensures that the output randomness remains unknown to the adversary until the commit phase of RandGene, when all honest nodes commit the random values they hold.

Proof: By intuition, the predictability of a randomness output by an adversary can be possible only if this adversary functions as a leader node, as the leader typically has enough information to recover the randomness output. However, the RandGene protocol is leaderless, so there is no way to leak such information from a special node, and each honest node works independently. While RandGene is based on sub-epochs, and these sub-epochs may leak some information. As we prove below, this is impossible for $\lfloor k/2 \rfloor + 1$ sub-epochs, and there is no way for an adversary to obtain enough information to recover the final randomness before the *commit* phase. The random output of RandGene is a function of $\lfloor k/2 \rfloor + 1$ out of k random shares to construct a final randomness, where roughly half of these random values are at *pre-commit* phase and the other half of them are at *re-commit* phase. Further, the calculation of the index l of the first random share requires the commitment of the last sub-phase in the *re-commit* phase. Following our protocol design, in order to achieve unpredictability, according to the pigeonhole theorem, there must be at least one value from the *re-commit* phase that remains unknown from the adversary until the *commit* phase.

We show that at least one random share exists from which the adversary cannot prematurely recover the final randomness. An adversary, e.g., by controlling enough number of participating nodes, has the ability to deviate from the protocol description and arbitrarily calculate the random shares of the *pre-commit* phase. Assuming that the adversary already knew $\lfloor k/2 \rfloor$ random shares of the first half. Our protocol is based on a synchronous model. When the sub-phase k is done, all previous sub-phases must be committed. In the *cast* phase, to calculate the first random share, which contributes to the final randomness, the sub-phase k should be committed. In the final randomness, we require at least $\lfloor k/2 \rfloor + 1$ out of k random shares to construct the final randomness. Considering the generalized pigeonhole principle, no matter how many dishonest nodes know about the previous $\lfloor k/2 \rfloor$ random shares, it will always have at least one *unknown* random share which comes from the second half random shares. In other words, there must be at least one random share unknown to the adversary from the *re-commit* phase. Thus, the adversary cannot construct final randomness before the commit phase of RandGene. ■

Lemma 3 (L3:Bias-resistance): Bias-resistance ensures that an adversary does not have the ability to influence the final randomness output.

Proof: To achieve bias-resistance, we must guarantee that at least one random share is out of the adversary's control. If, for every random share, an adversary has the ability to observe all honest inputs to the final randomness, which means he has the ability to recover all honest random inputs prematurely. In this case, the adversary may engage in trying different valid values as its commitments, and find a value that is most beneficial to him. If, for each random value, the adversary can exclude honest nodes from contributing input random shares to the final randomness, then the adversary has full control over all shard members, and hence the final randomness. This case is impossible with a high probability according to Section IV-B

As for the discussion of unpredictability, there is at least one random value that is out of the adversary's control before the end of the *re-commit* phase. And the *cast* phase is only to calculate the final randomness, which would not affect the final result as long as both the first and second half of random shares are committed. Roughly speaking, we can consider the *cast* phase as a recovery phase for all participating nodes, which is used to recover the commonly shared and committed random shares. As the protocol finishes the *re-commit* phase, all the random shares which potentially contribute to the final randomness are committed, and hence the final randomness. Each node then runs the *cast* phase to decide the index of the first random share, and the calculation of this index requires the last random shares in the *re-commit* phase, which is already committed based on the synchrony model. The protocol requires $\lfloor k/2 \rfloor + 1$ out of k random values to construct the final randomness, which means that at least one random value comes from the second half commitment. And at least one random value from the second half commitment is partially related to the last committed random share. As a result, the honest nodes' final randomness output would not be affected, no matter how powerful of the adversary. ■

Lemma 4 (L4:Public-verifiability): For each epoch E, an external verifier can verify the validity of the final randomness value R_E at the end of epoch E.

Proof: In RandGene, all the participating nodes obtain the final randomness as a block in the blockchain. Randomness blockchain can act as a trusted third party, which can provide its service to another party, and any third party should have the ability to verify the validity of the final randomness independently. In our design, the output of RandGene contains both the generated final randomness and a transcript L. L should be publicly verifiable on the process of generating final randomness. The transcript L must have some unique features, e.g., that the protocol execution can be replayed by any third party, and the final result should be unforgeable.

To be verifiable, the transcript L must contain all the communication messages of the protocol execution. Also, it must contain the configuration information of the corresponding epoch. As in Section III-B description, the string x proposed by a participating node consists of the id of the node and sub-epoch information, then applying this string x to a VRF to get a random value. This information is publicly available for all participating nodes, and will be recorded in the transcript L. As long as anyone has this transcript L, he/she can verify the validity of this final randomness. For example, a verifying third party who learns that the epoch configuration information is acceptable (i.e., the identities of the participating nodes) can re-execute the whole protocol execution process and verify the behavior of the whole process against the given final randomness.

When a correct final randomness generated, the only information that is still kept secret is the private keys of the participating nodes. However, during the verification process, the third party does not need to know the private keys to verify the transcript L, as signatures of messages can be verified using the public keys. ■

Lemma 5 (L5:Efficient-verification): For each epoch E, an external verifier can verify the validity of the final randomness value R_E in $O(kn)$ (where k is a constant) at the end of epoch E.

Proof: Anyone with the transcript L can verify the validity of the final randomness. In propose and release phases, the randomness shares provided by replicas are via a VRF function. From the construction of final randomness of RandGene, each replica at most needs to present k shares for k sub-epochs, which in total there are kn random shares to verify. And each verification process is simply which only requires the string x, the public key of the proposer, and the corresponding proof π returned by VRF. To calculate the final randomness, it also needs $\lfloor k/2 \rfloor + 1$ hash operations to calculate the random shares, which were included in final randomness. Comparing

with the computational-heavy PVSS operations, verifying VRF proof can be considered as a lightweight operation. In total, the final randomness value can be verified in $O(kn)$ (where k is a constant). ■

Theorem 6 (Good Randomness): In every epoch, the properties of good randomness meet with high probability. In addition, no node has the ability to know the random outputs ahead of one round. That means, if epoch e obtains a good random output, the epoch $e+1$ also will be a good random output.

Proof: By Lemmas 1 – 4, we can claim that the output of RandGene protocol is a good randomness ■

Put All Together: From the above discussion, RandGene serves as a reliable randomness source, which meets the requirements of availability, unpredictability, bias-resistance, and public-verifiability. The RandGene itself can provide a scheme to generate a reliable random beacon. To deal with the scalability issue, we integrate the sharding scheme into the construction of RandChain. The RandGene builds a local blockchain for each shard, which further can be used to attest RandChain.

B. Secure Shards

This section analyzes the secure sharding process, and then we analyze the failure probability within each shard.

1) Secure Sharding Process: The security of an epoch E is a critical aspect in a sharding scheme to ensure the whole system cannot be compromised. We first need to define what a good shard is for RandChain. For each shard in RandChain, a good shard should follow a *good majority* rule.

Definition 2 (Good Majority): We say that a shard has a good majority, with high probability, if it has at least half of its shard members are honest nodes.

Most of the previous sharding systems are randomness-based schemes, such as as [12] [13] [14] [17], to assign the participating nodes into shards. We need to show each shard in RandChain is with a good majority, and we will use *hypergeometric distribution* to show this property.

Lemma 7 (Secure Sharding): We claim RandChain meets a good majority for each shard with high probability. Alternatively, we call it as a secure sharding process.

Proof: The upper bound failure probability of a sharding scheme is estimated via the *hypergeometric distribution* (a probabilistic security analysis) [18] [19], the calculation of which typically requires $O(m^k)$ time (m is the number of participating nodes in one shard, and k is the number of shards in total), where $m = n/k$ and n is the number of total nodes [14].

We first utilize the recursive formula and hypergeometric distribution to define the failure probability in a random-based sharding protocol. The key idea is to calculate the number of safe assignments and then use this number to calculate the failure probability. Let $W_{safe}(x, y)$ refer to the number of safe assignments, where x represents the number of malicious nodes and y is the number of shards. $W_{safe}(x, y)$ literally means that x malicious nodes are assigned to y shards and the system is safe. Typically, we consider a shard is *unsafe* if *half of* its shard members are malicious nodes or controlled by an adversary (See the consensus protocol in Section III-B). For all sharding assignments, if *any* assignment triggers the unsafe case, we consider this assignment as being a failure assignment. The corresponding failure probability can be calculated by $P(failure) = 1 - W_{safe}(g, k)/C_n^g$, where $g < n/3$ is the number of nodes corrupted by the malicious adversary[6], and C_n^g is a combination operation ($C_n^g = \frac{n!}{g!(n-g)!}$). We utilize a recursive formula to derive $W_{safe}(x, y)$. We first consider the case that there are t malicious nodes assigned to the $y-$th shard and there are $g-t$ malicious nodes left, then the original question is to calculate the number of the safe assignments so that the remaining $g-t$ malicious nodes are assigned to $y-1$ shards, respectively. The safe assignments of $W_{safe}(x, y)$ can therefore be calculated recursively by:

$$W_{safe}(x, y) = \sum_{t=0}^{d} F(x-t, y-1) C_m^t,$$

where $d = \lfloor \frac{m-1}{2} \rfloor$ and $W_{safe}(x, 1) = C_m^x \mathbb{1}_{x \leq d}$. This case is for random-based sharding. For instance, given $n = 1800$ and $k = 8$, the failure probability $P(failure)$ =1.26e-07. Thus, we can obtain a secure sharding process with high probability. ■

2) Secure Shard: In Section IV-B1, we analyze our secure sharding process is with high probability. Simply, we first assume the participating nodes are randomly assigned to distinct shards. It is necessary to analyze the failure probability of the availability, a probability that the protocol fails to maintain the availability [9]. As for a continuous randomness generator, we try to obtain the upper bound for this failure probability in RandGene, which is responsible for each shard. Typically, we can use a random variable, e.g., X, to model a system's failure probability $P[\cdot]$, and we require to obtain an upper bound within a single shard. In our model, X follows the hypergeometric distribution, as well as the Boole's inequality, which together are being as a union bound. At first, we apply a *Chvatal*'s formula [20] to a single shard:

$$P[X \geq E[X] + cd] \leq e^{-2cd^2},$$

where $d \geq 0$ is a constant value and c is the number of draws in a hypergeometric distribution (in this paper it represents the size of a shard). The case of having a disproportionately high amount of dishonest nodes in a specified shard can be modeled by the variable $X \geq c - t + 1$, where t is the threshold required the number of nodes agreed on the random value. In our case, we set the threshold $t = cp + 1$ since the expectation $E[X] = cp$, where $p \leq 0.50$ is the adversarial power in shards. By integrating all the above information into *Chvatal*'s formula and performing some simplifications, the following formula is generated:

$$P[X \geq c(1-p)] \leq e^{-2c(1-2p)^2}.$$

Applying a union bound on this result, we obtain Fig. 4, which shows the trends of the average system failure probabilities q when the shard size increases. Different colors represent different cases of compromised nodes. The $x-$axis shows the size of a shard, and the legend shows the different adversarial powers. While the $y-$axis is in the form of $-log_2(\cdot)$ on the failure probability q. In general, the higher value in $y-$axis, the lower the system failure probability. And the change exhibits exponentially. For instance, a failure probability is at most $2^{-20} \approx 9.537e-7$ and $2^{-30} \approx 9.313e-10$ when the system failure probability is 20 and 30 (shown in $y-$axis), respectively.

[6]We use g, instead of f, to denote the number of malicious nodes; f represents the upper bound of malicious nodes in the designed *whole* system.

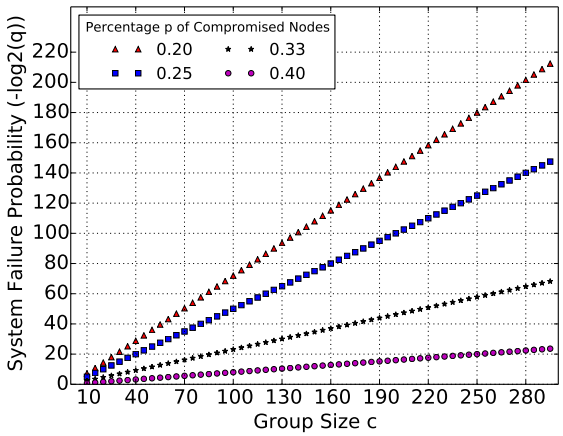

Fig. 4. System Failure Probability vs. Varying Group Sizes

C. Communication Complexity Analysis

We briefly analyze the communication complexity in RandChain. We define that the communication complexity refers to the number of messages exchanged between the nodes in a single run of the protocol.

The reduced overall communication complexity mainly takes advantage of the sharding scheme. Our RandGene scheme is a leaderless consensus protocol in the construction of shard chains. Without considering the sharding scheme, each participating node needs to broadcast its randomness shares to all other nodes. Suppose that we have n participating nodes; this will incur $O(n^2)$ communication complexity.

With the sharding scheme, n participating nodes are randomly assigned to each shard. Suppose the size of each shard is c. Within each shard, to get randomness, each node requires to broadcast its random shares to all other nodes, which will incur $O(c^2)$ communication complexity. In total, we have roughly n/c shards. Thus, the total communication complexity will be $n/c \times O(c^2)$, which is $O(cn)$. This communication complexity is for shard chains for all shards. Recall the construction of RandChain, above the shard chains; we have a second layer chain. This chain is constructed from shard chains, and each shard needs to select one participating node to be the consensus committee, in which it contains n/c members. The construction of this chain is based on the scalable BFT protocol, which will incur another $O(n/c)$ communication complexity when constructing RandChain. Thus, the total communication complexity for RandChain is $O(cn)$.

V. Conclusion

This paper proposes RandChain to generate good randomness via a sharding scheme to achieve scalability. RandChain is based on its primitive, RandGene, which is used to form local randomness. RandGene eliminates the use of computational heavy cryptographic operations to get shard chains; then these shard chains are fed to RandChain. By utilizing the sharding scheme, RandChain achieves scalability. Also, RandGene utilizes a commit-then-reveal approach and VRF to achieve good randomness to its shard chain. As future work, we plan to work on implementing both RandGene and RandChain schemes in applications and thoroughly evaluate its performance, such as the latency and throughput.

References

[1] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[2] M. Blum, "Coin flipping by telephone a protocol for solving impossible problems," *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.

[3] I. Bentov, R. Pass, and E. Shi, "Snow white: Provably secure proofs of stake." *IACR Cryptology ePrint Archive*, vol. 2016, p. 919, 2016.

[4] 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*. ACM, 2017, pp. 51–68.

[5] 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.

[6] J. P. Degabriele, K. G. Paterson, J. C. Schuldt, and J. Woodage, "Backdoors in pseudorandom number generators: Possibility and impossibility results," in *Annual Cryptology Conference*. Springer, 2016, pp. 403–432.

[7] S. Azouvi, P. McCorry, and S. Meiklejohn, "Winning the caucus race: Continuous leader election via public randomness," *arXiv preprint arXiv:1801.07965*, 2018.

[8] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1015, 2015.

[9] 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 *Security and Privacy (SP), 2017 IEEE Symposium on*. Ieee, 2017, pp. 444–460.

[10] 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.

[11] 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.

[12] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 17–30.

[13] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[14] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 931–948.

[15] 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.

[16] 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.

[17] R. K. Raman and L. R. Varshney, "Distributed storage meets secret sharing on the blockchain," in *2018 Information Theory and Applications Workshop (ITA)*. IEEE, 2018, pp. 1–6.

[18] A. Hafid, A. S. Hafid, and M. Samih, "A methodology for a probabilistic security analysis of sharding-based blockchain protocols," in *International Congress on Blockchain and Applications*. Springer, 2019, pp. 101–109.

[19] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "Hydrand: Efficient continuous distributed randomness," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 73–89.

[20] M. Skala, "Hypergeometric tail inequalities: ending the insanity," *arXiv preprint arXiv:1311.5939*, 2013.