Distributed Random Number Generation Method on Smart Contracts

Kentaro Sako
Waseda University
Tokyo, Japan
ksbowler@nsl.cs.waseda.ac.jp

Shiníchiro Matsuo
Georgetown University
NTT Research Inc.
Washington D.C., USA
Shinichiro.Matsuo@georgetown.edu

Tatsuya Mori
Waseda University
Tokyo, Japan
mori@nsl.cs.waseda.ac.jp

ABSTRACT

We propose *N-choice game* (NCG), a decentralized pseudo-random number generation method that can be executed on smart contracts. Of the M participants, one is a dealer, and the rest are players, each with a different role. Each participant randomly chooses one value between 0 and $N-1$ and receives a score determined by the NCG rule. The amount of reward each participant receives is determined by the score. The values chosen by the participants are combined and hashed into a pseudo-random number. The NCG framework is designed to achieve the following three goals: (1) Incentivize participants to provide random choices, (2) Evaluate the level of randomness in the decentralized environment, and (3) Establish high performance. We implement the NCG framework in Solidity and evaluate its performance. Our extensive experiments revealed that unless more than 90% of NCG players collide, the generated random numbers have high randomness that can pass the NIST randomness test. The experiments also demonstrated that the throughput of random number generation in NCG was 129 times faster than in the existing framework, Random Bit Generator [2].

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*.

KEYWORDS

blockchain, Smart Contracts, distributed random number generators

ACM Reference Format:
Kentaro Sako, Shiníchiro Matsuo, and Tatsuya Mori. 2022. Distributed Random Number Generation Method on Smart Contracts. In *2022 4th Blockchain and Internet of Things Conference (BIOTC 2022), July 8–10, 2022, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3559795.3559796

1 INTRODUCTION

Since the proposal of Bitcoin [11], there have been many previous works to enable autonomous economic activities without the need for trusted parties. While Bitcoin has simple financial transactions such as making payments, Ethereum aims to realize "smart contracts", which go beyond simple payment transactions with the dedicated programming languages for smart contracts such as Solidity [6], recently expanding to decentralized finance.

BIOTC 2022, July 8–10, 2022, Tokyo, Japan
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9662-2/22/07.
https://doi.org/10.1145/3559795.3559796

In this work, we address the problem of implementing random number generator (RNG) functions on smart contracts. Having RNGs in smart contracts has the following advantages: (1) providing fair mining opportunities based on probability rather than the Proof-of-Work, which is determined by the number of resources held by the miner, and (2) providing applications where probabilistically generated variables are essential. Generating random numbers with sufficiently high randomness in smart contracts is expected to be useful for various applications such as securing digital signatures, establishing unlinkability as a privacy element, and establishing the fairness in the game, lottery, or the millionaire protocol.

To the best of our knowledge, smart contracts platforms do not provide RNG functions. For example, Solidity, the most popular programming language implementation for smart contracts, does not implement any RNG functions/libraries. As smart contracts are required to be deterministic, they must produce the same results regardless of the environment the code runs; this principle may explain why the implementation of RNGs has been avoided. We note that using the output of the RNG function from another platform violates the decentralization principle of blockchain because it requires placing trust in that external platform.

Given this background, few works have aimed to provide the smart contracts platform RNG functions. Chainlink Verifiable Randomness Function (CVRF) [5] is a provably fair and verifiable RNG that can be used for smart contracts. CVRF receives a random seed from smart contracts and returns a value calculated using the CVRF secret key as a pseudorandom number. However, we note that CVRF is not a fully decentralized approach because it assumes that the Chainlink oracle is trustworthy. Chatterjee et al. [2] proposed a distributed random number generator named "Random Bit Generator" (RBG). While RBG established RNG functionalities for smart contracts in a fully decentralized manner, it has the following three fundamental problems:

Problem 1: No incentive mechanism to let participants randomize their choices.

In the RBG framework, each participant can make two choices, and the expected reward is the same no matter which one is chosen; i.e., even if only one of the two choices is made, the expected reward remains the same. Therefore, it is questionable whether users have the incentive to randomize their choices.

Problem 2: Evaluating the randomness

In the RGB paper, the authors did not evaluate the randomness of the output. It is not certain that the generated sequence had high randomness even when some participants made biased choices.

Problem 3: Low performance

The RBG framework generates one bit per contract, so the generation throughput is quite small. The authors reported that the throughput of RBG is 176.8 [bit/s] with parallel processing. According to their parameter settings, one contract takes ten times longer than the block's generation rate of 14,133 ms at the shortest, which means it takes more than 2 minutes.

Based on the above problems, we set the following three design goals (DGs) that need to be achieved in the fully distributed RNG framework for smart contracts.

DG1: Incentivize participants to provide random choices.
DG2: Evaluate the level of randomness in the decentralized environment.
DG3: Establish the high performance.

We propose a distributed random number generator, named *N-choice game* (NCG), that aims to achieve the DGs shown above. This game gathers M participants, one of whom is a dealer and the others are players. Each participant chooses one value, a random seed, between 0 and $N-1$. A player who makes the same choice as the dealer gives $N-1$ points to the dealer, and a player who makes a different choice than the dealer receives 1 point from the dealer. The values chosen by each participant are concatenated, and the cryptographic hash function is applied to obtain a 256-bit random number.

The key idea of the NGC framework is to design the game in such a way that it would be most reasonable for participants to choose random values. In the NCG framework, a player cannot manipulate the output RNG. Therefore, the motivation for players to participate in the game is to maximize the reward they receive for participating in the game. Since the expected reward decreases if other participants predict the value a player will choose, it is reasonable for them to make a random choice that is difficult to predict.

We conducted an evaluation experiment on NCGs and revealed the following:

- Unless more than 90% of NCG players collide, the generated random numbers have high randomness that can pass the NIST randomness test.
- The throughput of random number generation in NCG was 129 times faster than in RBG.

The remainder of this paper is structured as follows. We first present the background knowledge in section 2. Then, we describe how the NCG works in section 3. Section 4 presents the experimental results to validate that the NCG achieves its Design Goals. We discuss limitations and future work in section 5. section 6 discusses related works. Finally, Section 7 concludes our work.

2 BACKGROUND

In this section, we first describe the overview of the RBG framework [2]. We then describe the NIST randomness test, which we use to test the randomness of our distributed RNG outputs.

2.1 Overview of the RBG framework [2]

The RBG framework lets participants choose two options. In the framework, it is assumed that all participants' choices are pseudorandom because the expected value of the reward is the same regardless of which choice is made. Participants are divided into two groups according to their IDs, even or odd. The odd group chooses 1 or 3, and the even group chooses 0 or 2. The volume of reward participants receive is determined by the value they choose. The reward is increased for those who choose a value one higher than their choice and decreased for those who choose one lower than their choice (mod 4). For example, if a participant in the odd-numbered group chooses 3, the reward is increased for those who chose 0 and decreased for those who chose 2. Since the expected value of the reward amount is the same whether n or $n+2$ ($n \in \{0,1\}$) is chosen in either group, it is stated that all participants' choices are pseudorandom.

However, we claim it is doubtful that RBG meets the two requirements of having an incentive to randomize the input and guaranteeing randomness. Some players may make biased choices, such as choosing one side all the time. Another example is when the stakeholders of the random numbers generated by RBG participate. The authors have not verified that the generated sequence is still random under these circumstances.

2.2 NIST randomness test

The NIST randomness test is published by the National Institute of Standards and Technology (NIST) [15]. NIST randomness test consists of 15 different tests, and the P-value is calculated for each test. The significance level is set at 0.01, and the pass/fail is determined by whether the P-value obtained for each test exceeds the significance level. The Random Excursions Test and the Random Excursions Variant Test have the largest recommended test size of 1M bits. In addition, when converting (0,1) to (-1,+1) in binary data, the cumulative sum is calculated from the beginning to the last bit, and the number of times it becomes 0 in the process must exceed 500.

This paper tests the sequences that satisfy these two recommended rules. The definition of randomness is that an output sequence is computationally indistinguishable from a valid random sequence by any polynomial-time algorithm. Although the NIST randomness test is not completely capable of determining whether an output sequence fits this definition, to the best of our knowledge, the NIST randomness test is the most practical way and widely used in the research community. Referring to [7, 8, 14] that use the NIST random number test, we will consider the criteria for determining that it is random. Both papers judge the sequence randomly if the passing rate of 15 different tests is 90% or more. In this paper, 100 recommended sequences are generated, and if the p-value exceeds 0.01 at least 90 times for each test, it is determined as random.

3 N-CHOICE GAME

In this section, we describe the proposed method, NCG. We first present an overview of the NCG framework and describe the difference between NCG and RBG. Next, we explain the requirements for

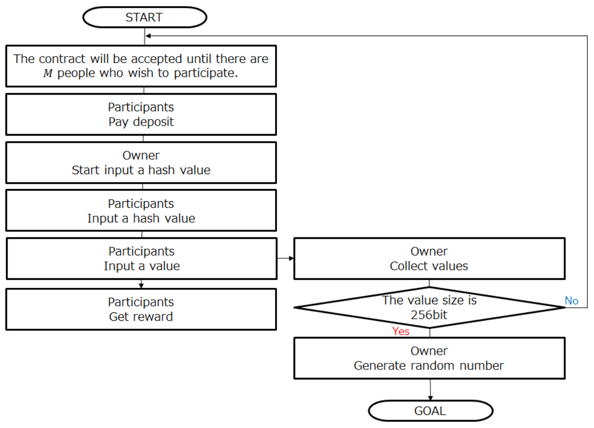
START
The contract will be accepted until there are M people who wish to participate.
Participants Pay deposit
Owner Start input a hash value
Participants Input a hash value
Participants Input a value
Participants Get reward
Owner Collect values
The value size is 256bit
No
Yes
Owner Generate random number
GOAL

Figure 1: N-choice game's flowchart

the parameters of the NCG. Finally, we show an example implementation of the NCG for a smart contract application, CryptoKitties [3]. An example implementation is available online at [16].

3.1 Overview of NCG

In the following, we describe the steps of the NCG framework. Figure 1 presents the flowchart of NCG.

Step 1: Parameter check

In the NCG framework, an owner who wants to generate a random number checks whether the parameters of the NCG meet the requirements shown in section 3.3. If the parameters satisfy the requirement, the owner initiates the contract process using the parameters.

Step 2: Recruitment of participants

Next, the contract process recruits M participants. Anyone can join NCG if they wish by paying a deposit. Participants are given a reward score that determines the reward amount, which is initialized at 0 points. The first node registered as a participant becomes the dealer and all other nodes (participants) become players.

Step 3: Start game

When M participants have gathered, the owner starts the game. The timestamp of the latest block when the game is started is stored in the valuable named T_S.

Step 4: Generating a hash value

One of the parameters set in **Step 1** is T_V, which is the time limit to send values to the contract. In this step, participants generate a hash value to obtain a reward. First, a participant selects $x (x < N)$. This value x affects the reward received after the game. x determines their scoring rewards. Next, they compute $y = x + k \times N$ using a large positive value k. y must not exceed 256 bits. Then, they calculate the hash value of y. Finally, they execute the `generateHashValue` function with y value as an argument between T_S and $T_S + T_V$.

Step 5 Transmission

Then, participants can enter the input value of the hashed value in **Step 4**. The time from $T_S + T_V$ to $T_S + T_V \times 2$ is set as the time for accepting value input. The participants call the `inputValue` function with y an argument. If the hash value of y does not match the value sent in **Step 4**, the execution does not complete, and an error is returned. Participants who completed without returning an error are considered honest participants, and those who did not are considered dishonest participants.

Step 6: Assigning rewards

After inputting values, the honest participants are granted rights to a refund of their deposit and participation and scoring rewards. They will be rewarded for performing the `finishGame` function. The reward for the score is determined by the reward score points. In **Step 4**, participants selected a value, x. If everyone is an honest participant, and the player chooses a value different from the dealer, the player gains 1 reward score point, and the dealer loses 1 point. Conversely, if the player chooses the same value as the dealer, the player's score is reduced by $N - 1$ points, and the dealer's score is increased by $N - 1$ points. In the presence of dishonest participants, if the dealer is the dishonest participant, players who are honest participants are assumed to have chosen values different from the dealer's. If the dealer is an honest participant, the player who is the dishonest participant is assumed to have chosen the same value as the dealer. The scoring reward is the variable R_g multiplied by this score. If there are dishonest participants, the owner is compensated for the failure to generate random numbers when the `finishGame` function is executed.

Step 7: Random number generation

Finally, the owner gets a random number. This step is available only if all participants successfully transmit the valid values. The values chosen by the participants are collected and converted into a binary sequence before being concatenated together. When $N = 16$, the participant P_1 chooses 13, and the participant P_2 chooses 4, the resulting binary sequence is computed as `11010100`, where the first and second 4 bits represent the value selected by P_1 and P_2, respectively. If there are more participants, the next 4 bits are the value chosen by those additional participants. Binary values are collected from all participants until 256 bits are collected. If there are not enough participants to accumulate 256 bits, the process is repeated from **Step 1** until 256 bits are collected. Once 256 bits have been collected, the value is hashed using SHA-256. This hash value is outputted as a pseudo-random number.

3.2 Incentive to choose a random value

In the following, we explain that it is reasonable for participants to choose a random value as long as all participants are working towards the reward.

Random choices

We calculate the expected value of the reward when the dealer and all the players randomly choose a value. The probability of a dealer and a player selecting the same value is $1/N$. If the dealer makes the same choice with one of the players, the dealer receives $N-1$ points, and if the dealer makes a different choice with the player, the dealer gives 1 point to the player. The expected value of the reward the dealer receives is calculated as follows:

$$\frac{1}{N} \times (N-1) - \frac{N-1}{N} \times 1 = 0$$

Similarly, the expected value of the reward a player receives is calculated as follows:

$$\frac{1}{N} \times (-(N-1)) + \frac{N-1}{N} \times 1 = 0$$

These observations demonstrate that the expected values of rewards for the dealer and the players are equal if they choose the values randomly.

Biased choices

Next, we study the expected value of reward if a participant does not select randomly and the other participants are aware of this tendency. The player's reward is either $-(N-1)$ points for choosing the same value as the dealer or 1 points for choosing a different value. To obtain a high reward, a player should choose a value different from the dealer's. If the dealer can predict the value, the player will choose, i.e., by knowing that the player always generates the same fixed value, the expected reward for the player is $-(N-1)$ points, since the dealer gets points for entering the predicted value and the player loses points for that. Therefore, it is reasonable for the player to avoid being predicted by the dealer.

Next, we consider the case where m ($m \leq M-1$) players are aware of the tendency of the dealer's choice. The m players can predict the value that the dealer will choose next, so they choose any other value and get 1 point of the reward from the dealer. The remaining $M-1-m$ players will choose the same value as the dealer with the probability of $1/N$. The expected value of the dealer's reward is calculated as follows:

$$1 \times (-1) \times m + \left\{ \frac{1}{N} \times (N-1) + \frac{N-1}{N} \times (-1) \right\} \times (M-1-m) = -m$$

The expected score of the dealer is highest when $m = 0$, implying that the dealer should choose randomly.

Summary

In summary, the expected value of the reward decreases when others predict the choice. It is reasonable for all the participants to choose random values as long as they expect to maximize the reward. Thus, the NGC framework establishes **DG1**. We note that such an incentive mechanism was not realized in RBG. The existence of participants whose objective is not the reward is discussed in Section 4.

3.3 Requirements for parameters

We explain the requirements for the parameters introduced in the NGC framework.

Number of choices (N)

In **Step 7**, the participants' values are concatenated together and collected up to 256 bits. To collect exactly 256 bits, both N and the bit length of N should be a power of 2. In addition, N should be greater than or equal to 2 to perform multiple selections.

Number of participants (M)

Multiple participants are required to establish a game. Furthermore, the number of participants must be a power of 2 in order to collect exactly 256 bits. Therefore, the number of participants M should be a power of 2 greater than or equal to 2.

Voting time (T_V)

In solidity, *block.timestamp* can be used to obtain the timestamp of a block as an integer. The block update frequency of Ethereum is about 15 seconds. We suppose that *block.timestamp* is updated from $bt0$ to $bt1$, for example. If T_V is small (e.g. 5), it could be computed as follows: $bt0 < startTime + T_V$ and $startTime + 2 \times T_V < bt1$. In this case, **Step 5** may not be executed. Therefore, the T_V should be at least as large as the Ethereum block update frequency.

Reward for participating the game (R_p)

Honest participants earn a participation reward. If all participants are honest, the total amount of the participation fee is $R_p \times M$. Since the owner bears the participation fee, the owner's budget should be equal to or greater than $R_p \times M$.

Amount of deposit (D)

If a participant fails to make an entry within the time limit, they are considered a dishonest participant, and their deposits will be forfeited. Honest participants should be rewarded for their correct behavior. Additionally, owners do not want to pay for their participation if they do not get a random number. Therefore, in the case of a dishonest participant, the forfeited deposit is used to reward other participants for their participation. In addition, compensation to the owner who originally wanted a random number will be made up from the deposit. Therefore, the amount of the deposit must be $D = R_p \times (M-1) +$ compensation to the owner.

Reward for the earned game score (R_g)

The reward each participant will receive as the earned game score, R_g, should be tuned carefully, as it will affect the motivation of participants to randomize their choices. If the R_g is too small, the incentive to aim for higher scores will be smaller, and the choices may not be randomized. If R_g is smaller than R_p, they will act correctly. A player's score will be the smallest possible value when choosing the same value as the dealer, $-(N-1)$ points. A dealer's score is at its lowest when choosing a value different from all players, $(-1) \times (M-1)$ points. The lower limit of the scoring reward is $R_g \times \min(N-1, M-1) \geq R_p$. On the contrary, if R_g is too large, the negative value in the scoring reward may exceed the deposited deposit and the participation reward. In that case, it would be more reasonable for participants to become dishonest participants. The balance of a dishonest participant is $-D$. The maximum value an honest participant loses is $R_p - R_g \times \max(N-1, M-1)$ To summarize, we need to tune the parameters so that the following conditions are satisfied:

$$\begin{aligned} D + R_p &\geq R_g \times \max(N-1, M-1) \\ R_g \times \min(N-1, M-1) &\geq R_p \end{aligned}$$

3.4 Proof-of-concept Implementation

We evaluate our proposed method by applying it to a proof-of-concept implementation on an existing application. In this paper, we adopt CryptoKitties [3] as an example of a smart contract application.

CryptoKitties is a blockchain-based game in which players earn Ethereum by trading kitties. This game can be run on Smart Contracts. If you own two or more kitties, a player can generate a new kitty by performing an operation called breeding. This breeding requires a 256-bit random number.

The proposed contract will be called whenever breeding is conducted on CryptoKitties. First, the CryptoKitties administrator executes the preparationOwner function and sets the parameters to fulfill the requirements. After M participants are gathered, the CryptoKitties operator calls the startGame function to accept the participant's input. After the time limit for accepting input values, either the CryptoKitties operator or the participant calls the finishGame function to give the participant refund, and the CryptoKitties operator calls the generateRandomNumber function to obtain a pseudo-random number. The following is an example of how parameters may be set. Let $N = 256$, $M = 32$ so that a single 256-bit random number is generated per game. In this case, the reward for participating will be approximately USD 10 per hour. In the RBG experiment, the length of time for participants to publish their hashes is 7 times the average block generation frequency of 14113 ms. The input selection time, *votingTime*, is set to its approximate value of 100 seconds. Roughly, the time required for one contract is about twice the *votingTime*, resulting in roughly 200 seconds. The participation fee is thus set to 0.0002 Eth, which is approximately USD 0.5 (1 Eth = USD 2781.33 (as of 15:16, 26/02/2022)). The compensation amount will be set to 0.008 Eth of the breeding fee. Calculating the scoring reward and deposit amount to meet the requirements, we obtain 0.000007 Eth and 0.0111 Eth, respectively. Alternatively, setting values to $N = 16, M = 64$ can generate a single 256-bit random number per contract as well. If the number of participants does not reach 64, parameters can be changed to $M = 16$ and playing 4 consecutive games.

4 EVALUATION

In this section, we evaluate whether the NCG framework achieves the three design goals, **DG1**, **DG2**, and **DG3** through the experiments. First, we identify the criteria to satisfy the three DGs. Next, we experiment to see if the NCG framework achieves the criteria. Finally, we evaluate whether the NCG satisfies all the Desing Goals based on obtained results.

4.1 Criteria to achieve the Design Goals

In the following, we describe the evaluation criteria for achieving the DGs.

Criteria for DG1

To satisfy this Design Goal, we must show that participants have an incentive to select values randomly. Section 3.1 showed that it is reasonable for participants who aim for rewards to choose values at random. We consider the purpose of a node's participation in addition to the reward. Nodes will receive a reward and output a 256-bit random number value for participating. They will participate for either the reward or an arbitrary output value. If they can get an arbitrary output value, they will act so that there is no incentive to choose randomly. It is necessary to show that no participant is interested in the output value. While obvious, it is important to consider that, as a pseudo-random number generator, participants must not be able to manipulate it to produce arbitrary output values. Additionally, the generated sequence must not be biased in any way. In other words, it is necessary to prove that it is impossible to attack the system so that it can generate a biased sequence.

Criteria for DG2

A random sequence should be generated regardless of the node participating. As mentioned in the **DG1** criteria, the objective of participation in the NCG is to obtain a reward or arbitrary output. We must consider the scenario in which the objective of a given participant is not to obtain an arbitrary output but instead to generate a biased sequence. We must show that a biased sequence cannot be generated in such a case. We consider **DG2** to be satisfied if the output sequence is random, even if the objective of all participants is to obtain an arbitrary output sequence.

Criteria for DG3

We compare the performance, specifically their throughput of generating random numbers, of NCG and RBG.

4.2 Experiment method

We will evaluate whether the NCG satisfies the criteria mentioned above. The following three experiments are conducted to check the criteria mentioned in Section 4.1.

Experiment 1: Evaluating the randomness of the output sequence when there are attackers who intentionally want to make the sequence non-random
Experiment 2: Evaluating the randomness of the output sequence when the participants are only interested in rewards
Experiment 3: Compare the throughput of random number generation for the NCG and RBG

The methods of each experiment are described in this section.

Experiment 1

To evaluate the randomness of the output sequence when there exist attackers who want to make the sequence non-random, we first formulate the threat scenarios. We then present how to generate the sequence according to the participants' behavior in each threat scenario.

Threat scenarios. The NCG outputs hashed 256-bit values created by the participants. If the participants' actions are restricted, the values to be hashed are limited. Specifically, we consider the case where the expected value of the scoring reward varies depending on the values chosen by the participants. We propose two scenarios: In the first scenario, the dealer is the attacker, and in the second scenario, several players are attackers.

Case 1. The dealer is an attacker
We consider the case in which the dealer tries to make the output sequence non-random. When the attacker is the dealer, they can limit the actions of other players by choosing values that follow a pattern. Specifically, if the dealer chooses the value x and all players know this, they can increase their score rewards by choosing a value other than x. This gives players the incentive to choose values other than x, generating a non-random sequence. By doing this, the dealer generates a non-random sequence by influencing the players' choices.

Case 2. Players are attackers
We consider the case in which there are colluding attackers within the players. The colluding attackers can limit the dealer and other players' choices by introducing a pattern in their choices. If the attackers all choose the value x, then the dealer, who predicted it, also chooses x, thereby increasing the score reward. At the same time, other players will choose a value other than x by predicting that the dealer will choose x. In this way, the dealer's actions can be limited to one and the player's actions to $N - 1$. The same can be said in the case of collusion between some players and the dealer.

Evaluation Method. The methods for generating the output sequences in the two threat scenarios are described.

Case 1. The dealer is an attacker
We evaluate the scenario in which the dealer is an attacker by assuming that the participants take the following actions. The dealer first chooses a value x. Next, all players choose a value other than x because they know that the dealer chooses x. x is pre-determined and is not changed until a generated sequence meets the test requirements. Three combinations of M and N are considered during evaluation, as shown in Table 1. One hundred sequences in each pattern are generated and tested with the NIST random number test.

Case 2. Players are attackers
We assume the participants take the following actions when some players are colluding. We suppose that m ($2 \le m \le M$) participants choose the value x, and the other players choose values other than x. We examine the output sequence for varying values of m between 2 and M. Three combinations of M and N are considered during evaluation, as shown in Table 1. One hundred sequences in each pattern are generated and tested with the NIST random number test.

Experiment 2

Table 1: Patterns of parameters.

parameter	M	N
Standard	16	16
Many choices	16	65536
Many participants	64	16

We evaluate whether the sequence output by the NCG is random when the objective of all participants is the reward. In this evaluation, we set the parameters to the standard pattern, $N = 16, M = 16$, as a representative pattern. We consider the following scenario to evaluate the output sequence when there are no attackers present. Then, we explain how to generate sequences with those action scenarios.

Action scenarios of participants. Again, the participant whose objective is the reward randomly chooses a value less than N. Therefore, we need to consider several different methods to simulate an unpredictable value chosen by the participants. We assume that the participant will use one of the four following methods to determine their values. A randomness value will be set for each of these four methods according to how difficult it is to predict their outcomes.

(1) *NIST random beacon*: It is a random sequence issued by NIST [12]. Readers can examine the output at [13]. Since there is currently no output prediction algorithm for NIST random beacon, the randomness number is set to 3.

(2) *Mersenne-Twister (MT)*: It is a pseudo-random number generator with a period of $2^{19937} - 1$, proposed by Matsumoto and Nishimura in 1996 [10]. We note that MT is not a cryptographically-secure pseudo-random number generator (CSPRNG). It is a generator that can predict what will be output next once the output of 624 times 32-bit MT is known, generated by a linear asymptotic formula. So, the randomness number is set to 2.

(3) *Linear congruential generators*: In this method, the following is calculated. $X_{n+1} = a \times X_n + b \mod m$ In this work, $m = N$, and a and b are set so that the maximum period is m. Although a, b, and m can be set as parameters, the outcome of this generator can be predicted if approximately ten consecutive outputs are given. The randomness number is thus set to 1.

(4) *Choosing the same value repeatedly.* A single value less than N is chosen repeatedly, under any circumstances. While it is infeasible for participants to behave this way, we consider this method for experimentation. The randomness number is set to 0.

Evaluation Method. We evaluate whether the sequences generated by participants who use methods such as those introduced are random. If more than 3 of the methods mentioned above are used among 16 participants in a given NCG, a considerable number of patterns will need to be considered. For simplicity's sake, we limit the maximum number of methods used among participants in a given game to 2 or fewer. For example, 9 participants may be using the NIST randomness beacon, and the rest may be using the Mersenne Twister. In this case, the average randomness number of the methods used by the participants is $(9 \times 3 + 7 \times 2)/16 = 2.5625$. This method results in 94 possible patterns, which we evaluate the output sequences. For comparison, we also validate the output sequence generated with RBG. For simplicity, we validate the output sequence with all participants either choosing the same value repeatedly or choosing values according to the NIST random beacon method.

Experiment 3

In this experiment, we compare the throughput at which the NCG and RBG generate one bit. We evaluate and compare the time it takes for the NCG and RBG to complete a contract.

Parameter Setting. In order to fulfill a contract, it is necessary to set the parameters for the time-limited phases in both the NCG and the RBG. The time-limited phases are the time to accept participants and the time to enter values. In the work where RBG was proposed, the average block generation frequency t is set to 14113 ms, the participant acceptance time $tReg$ is set to 3 times t, and the minimum time for one contract $tMin$ is set to 10 times t. After $tReg$ time has elapsed since the random number request, the value can be entered and lasts until the contract end time. Hence, the time to enter a certain value is $tMin - tReg$ = 7 times t. In this work, for simplicity, t is set to 15,000 ms, the participant reception time $tReg$ is set to 3 times t, and the time to enter a value $tVal$ is set to 7 times t. Regarding the recruitment of participants, it shall take only $tReg$ to gather M participants for either game. The time limit for hash value and value input acceptance in the NCG is $tVal$. The $tMin$ in RBG is assumed to be $tReg + tVal$. Participants must always decide on a value and input it during the time $tVal$. The value of N, M should be Many choices pattern.

The contract is executed ten times with the parameters set as described above. Evaluation is conducted by setting Go Ethereum (Geth) private net as the Solidity execution environment [9]. The average of those ten times is the time required for one contract.

4.3 Results

We evaluate the results of Experiments 1, 2, and 3 to see whether the NCG framework achieves the design goals.

Experiment 1

We evaluate the randomness in cases 1 and 2. Based on the evaluation, we determine whether the NCG satisfies DG1.

Case 1. The dealer is an attacker
The results of the three patterns are shown in Table 2. Table 2 shows the pass rate for the test with the lowest pass rate out of the 15 tests for the NIST randomness test and the overall average pass rate. All three patterns can be random since the pass rate for the test with the lowest pass rate is above 90%. Hence, it would be impossible for the dealer alone to attempt an attack that would make the sequence non-random.

Table 2: The result when the dealer is an attacker.

parameter	Standard	Many choices	Many participants
Minimum pass rate [%]	92	92	92
Average pass rate [%]	98.467	98.400	97.933

Table 3: Minimum m value resulting in non-random output.

parameter	Standard	Many choices	Many participants
Minimum m	15	16	61

Case 2. Players are attackers
Randomness validation results for each of the three patterns are shown in separate figures. The results for the standard pattern are shown in Fig 2, the results for the many choices pattern are shown in Fig 3, and the results for the many participants' pattern is shown in Fig 4. The vertical axis represents the test pass rate for all figures, and the horizontal axis represents the number of participants colluding. The *min* line represents the lowest pass rate out of the 15 tests for the NIST randomness test. The *ave* line represents the overall average pass rate for the NIST randomness test. The least amount of colluding participants needed in all patterns to cause a non-random output is shown in Table 3. 90% or more of the total participants must collude in all three patterns to make the sequence non-random. Thus, collusion attacks can be assumed to be impossible.

From Experiment 1, we conclude that the NCG satisfies DG1. As shown in Table 3, if one wants to create a non-random sequence intentionally, about 90% of all participants must collude. It is thus unlikely that an attack conducted by colluding attackers aimed to make the sequence non-random occurs. In addition, if a participant has a stake in the generated number, they may try to obtain an arbitrary output. However, since the output is a hash of everyone's chosen values, it is impossible to obtain an arbitrary output even if everyone colludes. Therefore, since it is impossible to obtain arbitrary values or conduct colluding attacks, participants in the NCG will participate for participation and scoring rewards. Since we showed in Section 3 that it is reasonable for participants who aim for rewards to choose values at random, the NCG satisfies DG1.

Experiment 2

The results of the randomness validation of NCG for the output sequence when all participants aim for score rewards are shown in Fig 5. The results of RBG are shown in Fig 6. The vertical axis

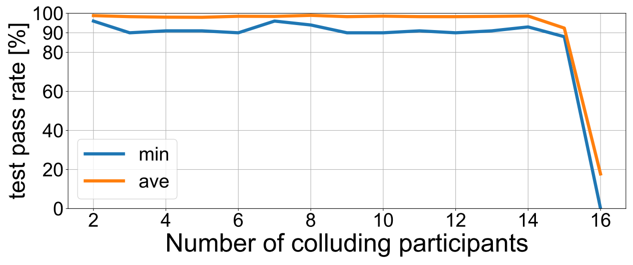

Figure 2: Colluding in Standard pattern.

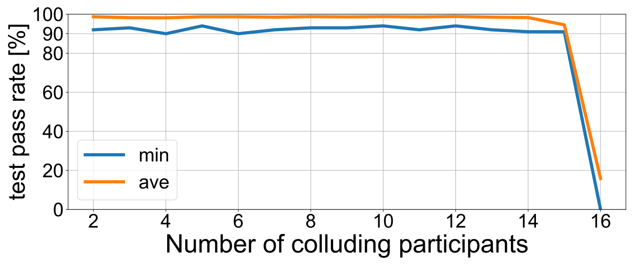

Figure 3: Colluding in Many choices pattern.

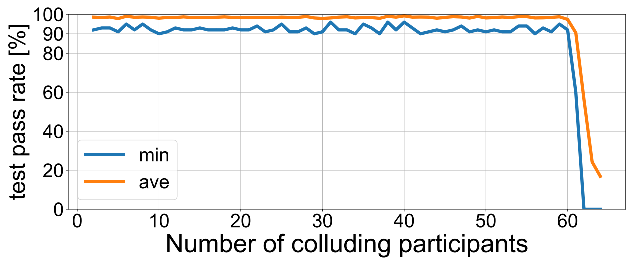

Figure 4: Colluding in Many participants pattern.

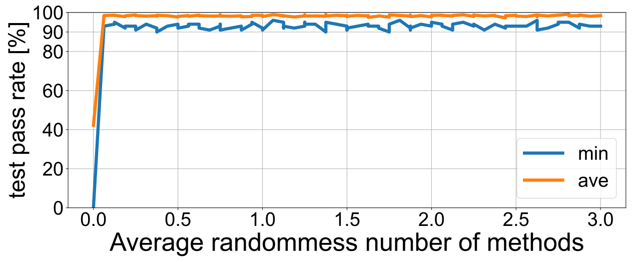

Figure 5: The result when all participants are honest.

is the pass rate for the NIST randomness test. The horizontal axis is the average randomness number of the methods used by the participants. The *min* line represents the lowest pass rate out of the 15 tests for the NIST randomness test. The *ave* line represents the overall average pass rate for the NIST randomness test. The pass

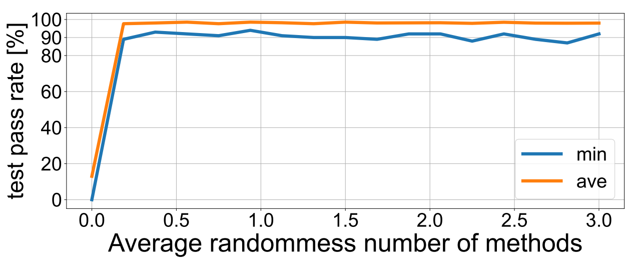

Figure 6: The result of RBG.

rate for all tests exceeded 90%, except when everyone's randomness value is 0, i.e., everyone chooses the same value repeatedly. In contrast, the RBG outputs sequences that are non-random, even with participants choosing random numbers.

Results of the Experiment show that the NCG satisfies DG2. We explain in section 3.1 that participants in the NCG participate for rewards. Experiment 2 had shown that the sequence output in such cases met the criteria of being random except when everyone kept choosing the same value. This is not a rational move on the participant's part, as the expected reward will be lower, as it will be easier for other participants to predict that value. Randomness is guaranteed if there is more than one reasonable participant.

Experiment 3

NCG takes on average 348.18 [s] to complete one contract, while RBG takes 176.09 [s]. While NCG takes longer than RBG to complete one contract, the NCG generates 256 bits per contract, while RBG generates 1 bit per contract. Given the time it takes to complete one contract, NCG 348.18/256 = 1.3601 [s/bit]. The throughput of NCG is 129 times faster than RBG, which achieves DG3.

5 DISCUSSION

The throughput of random number generation for the NCG framework depends on the block creation period of Ethereum. This is because we use solidity's *block.timestamp* to get the time. *block.timestamp* is a variable that returns the timestamp listed in the latest block. As of 2022, one Ethereum block is generated approximately every 15 seconds. Since the acceptance period for both hash and inputs can be no shorter than the block creation period of Ethereum, the fastest a random number could be generated in 30 seconds. The acceptance time could be reduced if blocks were generated in a shorter period, but this would negatively impact the security of the Ethereum blockchain. If a function could obtain UNIXTIME without trusting a third party, it would be possible to generate many random numbers.

The experiments conducted in this paper compared the generation speed of both NCG and RBG when done with a single thread. It would be possible to generate more random numbers if multiple NCG contract programs were deployed. Comparison with other methods, such as RBG under the condition that a single job can be processed in parallel, has not yet been made.

This study assumes that one person can operate only one node. We consider the incentive for a Sybil attack in which one person creates and manipulates multiple nodes. The purposes of participation in NCG are reward and output value. Since NCG is a zero-sum game where money moves among participating nodes, the expected value of the R_g remains the same no matter how many nodes are created. First, R_p is paid to each node that participates, a participant will receive more reward based on the number of nodes created. In addition, when someone manipulates nearly 90% of all participating nodes, an attack will happen according to the threat scenario in Experiment 1. Hence, creating multiple nodes is advantageous for both reward and output values. However, the probability that more than 90% of the total number of nodes can be manipulated by either making the output sequence non-random is relatively low. We consider the probability of this attack. We suppose $M = 16$, and there are only two people who create a large number of nodes and wish to participate in NCG. The probability that either one of them can manipulate more than 15 nodes in a game is 0.052%. The probability of a successful attack is low, but it is still a threat. Although it conflicts with anonymity, if there is a system in which one person can create only one node with identity verification, the likelihood of an attack occurring can be reduced.

6 RELATED WORK

While not on smart contracts, multiple random beacon protocols for distributed random number generation have been proposed in previous works. We present three representative protocols.

Hydrand

Schindler proposes a new distributed random beacon protocol based on Publicly Verifiable Secret Sharing (PVSS) [17]. They state that it allows for continuous output without a trusted party, even in the presence of an adversary. Specifically, it is divided into proposal, approval, and voting phases. There is one leader who broadcasts their data set. A given node will receive the data set, sign it, and broadcast its shared values. That value determines the beacon value. While existing PVSS-based methods reduce the amount of communication to the cube of the number of nodes, Hydrand reduced it to the square of the number of nodes.

RandPiper

Bhat proposes RandPiper, which enables efficient generation even with node turnover in communications where the communication volume is quadratic [1]. They list the following requirements for a random beacon: unpredictable and unbiased output, optimal resilience, low communication overhead, efficient node replacement, and use of an efficient cryptographic scheme. They point out that existing research does not meet any of the listed requirements. Furthermore, Hydrand argues that achieving better communication complexity with minimal assumptions sacrifices resilience. They design a resilient Byzantine fault-tolerant (BFT) state machine replication (SMR) protocol to reduce communication complexity while using efficient cryptography. Two random beacon protocols, GRandPiper and BRandPiper, are created as components of this protocol. GRandPiper uses PVSS and enables efficient communication. However, since the output depends on the leader, the output is predictable when the static adversary is the leader. Using PVSS to achieve substantial unpredictability would complicate communication, so another method was proposed. BRandPiper uses improved VSS and its secret sharing isomorphism to make it considerably less advantageous and predictable for the adversary. However, the communication complexity is proportional to the actual number of faults. Compared to the practical beacon protocol DRand, the beacon generation speed is comparable, and node replacement was efficient.

SPURT

Das rejects existing studies as either having incomplete security guarantees, being viable only within limited settings, or having high costs. They also state that BRandPiper is proportional to the cube of the number of nodes in the worst case [4]. They state that their proposed method, SPURT, provides secure results even if one-third of the nodes in the network are malicious participants. They modify the SMR protocol so that all nodes get beacons at approximately the same time. Using the standard Decisional Bilinear Diffie-Hellman assumption, there is no need to trust some parties. Furthermore, by using PVSS, each node computes its value. These modifications allow random beacon generations at a low cost in general and critical security settings. As a result, they state that a network of 32 nodes can generate 84 beacons per minute, which is comparable to other studies.

7 CONCLUSION

We propose a distributed pseudo-random number generator, NCG, which can be run on Smart Contracts. Since it is impossible to obtain arbitrary outputs in the NCG due to the use of hash functions, participants aim to receive more score rewards in the game. The more a participant's choice is predicted by others, the lower the expected value of the score reward. The game can incentivize participants to choose a value at random, which is impossible in the RBG game in existing research. The NIST random number test verifies the high randomness of the output sequences, and the NCG can achieve higher throughput than the RBG. Further improvement of NCG is left for future study.

REFERENCES

[1] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. 2021. RandPiper - Reconfiguration-Friendly Random Beacons with Quadratic Communication. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 3502–3524. https://doi.org/10.1145/3460120.3484574

[2] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. 2019. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. *CoRR* abs/1902.07986 (2019). arXiv:1902.07986 http://arxiv.org/abs/1902.07986

[3] CryptoKitties. 2017. CryptoKitties. Retrieved April 23, 2022 from https://www.cryptokitties.co/

[4] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. 2021. SPURT: Scalable Distributed Randomness Beacon with Transparent Setup. *IACR Cryptol. ePrint Arch.* (2021), 100. https://eprint.iacr.org/2021/100

[5] Chainlink Developers. 2022. Introduction to Chainlink VRF. Retrieved April 23, 2022 from https://docs.chain.link/docs/chainlink-vrf/

[6] Solidity developers. 2016. Solidity. Retrieved April 30, 2022 from https://docs.soliditylang.org/en/v0.8.13/

[7] Farah Ferdaus, Bashir M. Sabquat Bahar Talukder, Mehdi Sadi, and Md. Tauhidur Rahman. 2021. True Random Number Generation using Latency Variations of Commercial MRAM Chips. In *22nd International Symposium on Quality Electronic Design, ISQED 2021, Santa Clara, CA, USA, April 7-9, 2021*. IEEE, 510–515. https://doi.org/10.1109/ISQED51717.2021.9424346

[8] Viktor Fischer and Milos Drutarovský. 2002. True Random Number Generator Embedded in Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers (Lecture Notes in Computer Science, Vol. 2523)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.). Springer, 415–430. https://doi.org/10.1007/3-540-36400-5_30

[9] The go-ethereum Authors. 2013. Go Ethereum. Retrieved April 30, 2022 from https://geth.ethereum.org/

[10] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30. https://doi.org/10.1145/272991.272995

[11] Satoshi Nakamoto. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System.* Retrieved April 23, 2022 from https://bitcoin.org/bitcoin.pdf

[12] NIST. 2019. Interoperable Randomness Beacons. Retrieved April 23, 2022 from https://csrc.nist.gov/projects/interoperable-randomness-beacons

[13] NIST. 2019. NIST Randomness Beacon (Version 2.0 Beta). Retrieved April 23, 2022 from https://beacon.nist.gov/home

[14] Charles W. O'Donnell, G. Edward Suh, and Srinivas Devadas. 2004. PUF-Based Random Number Generation. Retrieved April 23, 2022 from http://people.csail.mit.edu/cwo/publications/mit-csail-csg-481.pdf

[15] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. 2010. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Retrieved April 23, 2022 from https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[16] Kentaro Sako. 2022. N-choice game source code. Retrieved April 26, 2022 from https://github.com/ksbowler/N-choice_game

[17] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar R. Weippl. 2020. HydRand: Efficient Continuous Distributed Randomness. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 73–89. https://doi.org/10.1109/SP40000.2020.00003