BCRAND: A Framework to Provide Runtime Random Numbers for Smart Contracts

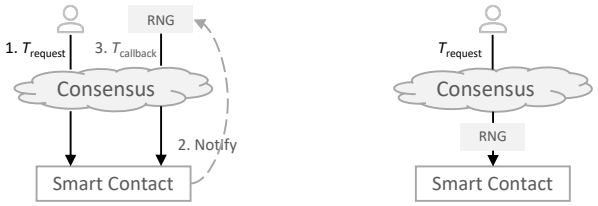
Abstract

Random numbers are crucial to decentralized applications in the blockchain ecosystem. Modern random number providers (RNPs) that generate and distribute random numbers to smart contracts rely on callback transactions to defend against MEV attacks on BFT-based blockchains. However, the cost of callback function is not negligible, evidenced by extra fees, processing delay, and bloated on-chain storage.

We propose BCRAND, a novel random number provider framework that can employ off-the-shelf distributed random beacons (DRBs) to build runtime RNP protocols. A runtime RNP can generate and distribute random numbers to smart contracts within the same consensus round in which the random numbers are requested. To prevent MEV attacks without requiring callback transactions, BCRAND first locks the transaction list and then enables consensus nodes to jointly generate an unbiased, unpredictable, and verifiably unique beacon by the underlying DRB scheme at the end of the BFT protocols. This beacon is then used to generate random numbers for requested transactions. In additional to the innovative architectural design, we also overcome many other challenges, such as a malicious revert action that may compromise the fairness of the executed applications. We further prove that the framework satisfies the intended security and confirm its performance advantage through thorough experiments. The experimental results show that the average fee cost for requesting a random number with BCRAND is reduced by at least 89% compared to the callback-based solutions. If 0.1% of the total transactions request random numbers, BCRAND generates 89% less on-chain data. Besides, BCRAND has minimal impact on the performance of underlining BFT consensus.

1 Introduction

Smart contracts [22, 88] are at the core of the decentralized applications (DApps) ecosystem. Many DApps, including DeFi and decentralized games, require access to randomness provided by public RNPs. For instance, decentralized games use randomness to determine the winner [60]; in Non-fungible tokens (NFTs), randomness ensures uniqueness [32] and rarity [72]; a legal toolkit relies on random jurors to resolve disputes [61].



(a) Callback RNP (b) Runtime RNP

Figure 1: Comparison between callback RNP and runtime RNP. $T_{\texttt{request}}$ and $T_{\texttt{callback}}$ denote transactions for requesting and providing random numbers respectively. Solid line represents one consensus round. Callback RNP (a) requires two transactions with at least two separate consensus rounds to provide one random number; while runtime RNP (b) completes one request within one round. RNG stands for random number generator.

An RNP protocol produces random values that are bias-resistant and unpredictable, such that no entity can interfere with the generation and foresee the result. In the early development of blockchain RNPs, blockchain data was used as an entropy source. Since the blockchain data is public, the random numbers generated by on-chain RNPs can be computed in advance, leading to the Miner Extractable Value or Maximum Extractable Value (MEV) attacks [35, 51, 84, 86]. Many lottery contracts hosted on the EOS blockchain [38] were compromised as a result, where attackers profited by correctly predicting the winning number [83]. Loot [72], one of the most successful NFT projects, generates random numbers similarly using on-chain data. Prior to claiming tokens, attackers could know their rarity. Random number related attacks have been widely regarded as one of the most prevalent types of blockchain attacks [84].

In the literature, callback-based RNP protocols were proposed to mitigate MEV attacks [2, 14, 24, 30, 45, 53, 79, 80, 87].

In these methods, a random number is emitted at intermittent intervals during consensus. To mitigate MEV attacks, they explicitly leverage a callback mechanism as shown in Figure 1a. User parameters are first recorded in the transaction $T_{\texttt{request}}$ to initiate the random number request during a block interval. Subsequently, a second transaction $T_{\texttt{callback}}$ is needed in the following block time to produce random numbers. With $T_{\texttt{request}}$ coming before $T_{\texttt{callback}}$ and locating in different consensus rounds, the attacker cannot predict the random number in advance. Callback RNPs, as promising as they are, are plagued by the following limitations:

(i) Expensiveness: Each transaction on the blockchain incurs a cost in the form of transaction fees. As callback RNPs require two transactions to complete one random number request, the user initiating the request must pay transaction fees for both $T_{\texttt{request}}$ and $T_{\texttt{callback}}$. One might consider reducing the amortised cost by storing multiple $T_{\texttt{request}}$s and providing one single $T_{\texttt{callback}}$ to callback all requests. However, this idea is unrealistic due to the fact that $T_{\texttt{request}}$s may trigger various smart contracts, and it is impossible to activate all of these smart contracts inside a single $T_{\texttt{callback}}$.

(ii) Latency: $T_{\texttt{request}}$ and $T_{\texttt{callback}}$ must exist in different blocks in order to store user parameters prior to the creation of random numbers, i.e., it takes at least two blocks to generate a random number for a smart contract [27,42,76]. Consequently, the execution of the smart contract would be significantly delayed.

(iii) Storage Overhead: Despite the fact that a random number might be as short as a few bytes, the transaction required to obtain it is much larger. In addition to the random number, the transaction also contains the coordinator's address, token's address, key hash, fee, and function code. For example, $T_{\texttt{callback}}$ in Chainlink VRF [40] is 624 bytes, while the random number is only 32 bytes. This results in extra storage burden to the blockchain.

Looking ahead, our solution, a runtime RNP framework for smart contracts, can fulfill random number requests within the same consensus round and aims to tackle above concerns. The idea of runtime RNP is illustrated in Figure 1b.

BCRAND - A Framework to Provide Runtime Random Numbers. In this work, we propose a novel framework to generate and use random numbers, which achieves the goals: *(i)* Low transaction fees – obtaining random numbers without paying transaction fees for extra callback transactions; *(ii)* Low latency – random numbers are fulfilled within one round of consensus; *(iii)* Low storage overhead – only one transaction for requesting random numbers needs to be stored on blockchain.

We present BCRAND, a secure runtime RNP framework for smart contracts on BFT-based blockchain. BCRAND does not require the transaction $T_{\texttt{callback}}$ [47] to work. The main idea of BCRAND is to generate a random beacon value from a distributed random beacon protocol for each block during the consensus. Then the beacon is fed into a pseudorandom number generator to generate the required random numbers for smart contracts. As such, a random number request can be completed at runtime by using only one transaction, $T_{\texttt{request}}$, which significantly reduces the time required for randomness-related smart contact functions. Meanwhile, BCRAND also reduces the relevant transaction fees and on-chain storage overhead by removing $T_{\texttt{callback}}$ (see Section 7).

Technical Challenges. In order to turn the above idea into a practical solution for a BFT-based blockchain system, BCRAND needs to address the following challenges. ***(i)*** Since BCRAND does not rely on a callback function, it is important to consider mitigation against MEV attacks. ***(ii)*** The random beacon value is generated during the consensus. Other than being bias-resistant and unpredictable, the generated seed should also be verifiably unique in the presence of Byzantine faulty nodes in the system. ***(iii)*** Because smart contracts can communicate with one another, users can verify the execution result of a smart contract and deliberately revert the transaction if the result is not profitable (we call this action malicious revert, see Section 3.2.1).

BCRAND adopts the following methods to address aforementioned issues. First, we lock the transaction list proposed by the leader node in a BFT consensus to ensure that it cannot be modified later, thus mitigating the MEV attacks. After locking the transaction list, we generate a verifiable random beacon for each block.

Finally, BCRAND provides a library for developing the malicious revert defender for smart contracts. In order to evaluate the proposed framework and compare it with existing callback RNP, we instantiate a concrete runtime RNP protocol BCRAND-DBLS by leveraging distributed BLS (DBLS) threshold signature that has been used as the core building block for many DRB protocols [2,15,37]. We also implement BCRAND-DBLS$_{callback}$ as baseline to simulate the behavior of a callback RNP on neo [67]. We discover that if 0.1% of transactions are $T_{\texttt{request}}$, BCRAND saves 89% of on-chain data and reduces the monetary cost to complete a random number request by at least 89% while only incurs a negligible 0.002% more consensus time. To the best of our knowledge, **BCRAND is the first secure runtime RNP framework designed for smart contracts of blockchain projects running on BFT protocols**.

Contributions. The main contributions of this paper are summarized as follows:

- We present BCRAND, a novel runtime random number provider framework for smart contracts on BFT-based blockchains. Developers can create their own runtime RNP using existing distributed random beacon protocols and pseudorandom number generators.
- We remove the callback procedure by embedding BCRAND into the BFT consensus to enable the effi-

cient random number provision without affecting the consensus function. BCRAND is fault-resilient by BFT and is immune to MEV attacks due to the removal of callback.

- We identify Malicious Revert, which poses a potential risk to runtime RNP. We offer a defense mechanism to mitigate its impact.
- We design, implement, and evaluate BCRAND-DBLS, an instance of BCRAND. Smart contracts on BFT-based blockchain that use BCRAND-DBLS can securely fulfill any number of (if not bounded by consensus) random number requests at runtime.
- Four applications are evaluated to show the effectiveness and efficiency of BCRAND. We also compare BCRAND to other RNPs to demonstrate its unique advantages.

2 Background

Before getting into details, we first give some notations used throughout our paper in Table 1.

Symbol	Description
$\mathcal{L}$	Block proposal in the consensus.
λ	Security parameter.
σ_i	DBR partial beacon, from leader when $i=0$; from consensus node otherwise.
σ/O	Generated beacon from random beacon protocol.
r	Random number for smart contract.
p_i	Consensus node. Leader when $i=0$, replica otherwise.
n	Number of system participants.
t	Maximum Byzantine nodes allowed.
k	Maximum malicious key sharing nodes allowed.
f	Size of faulty nodes s.t., $f \leq t < k \leq 2t+1$.

Table 1: Notations.

2.1 Blockchain

Blockchain technology creates a tamper-proof source of truth by utilizing distributed consensus algorithms and cryptography to collect and organize user transactions across the network. Blocks are data structures within the blockchain, where transactions are permanently recorded.

2.1.1 Smart Contract

Smart contracts are computer programs that define and execute a set of rules on blockchains. When a contract is invoked, it is performed by all nodes in the network. Through consensus protocols, the whole network verifies the computation result, therefore establishing a fair and trustless environment conducive to the development of a variety of decentralized applications [62, 93]. An increasing number of smart contracts require randomness to perform their functions, such as in NFT contracts where randomness is required for fair distribution [72] and in GameFi contracts where randomness is used to determine winners and rarity for entities [48].

2.1.2 Byzantine Fault Tolerance (BFT)

BFT is a family of consensus algorithms, such as PBFT [26] and its variants [3,6,9–12,23,25,33,54,56,58,59,67,90–92], designed to overcome the Byzantine Generals' Problem in distributed settings [57]. At the time of writing, 17 out of the top 80 chain projects adopt BFT consensus [34], such as Theta [69], neo [67], and EOS [38].

The BFT consensus enables a distributed system to reach an agreement in the presence of a group of faulty nodes that may behave arbitrarily. BFT consensus occurs in rounds and each round includes one or more views [26]. When a round ends, a new block is formed to represent the current network consensus. For a BFT protocol containing a total of n nodes, it can tolerate up to t faulty nodes. Let f be the actual number of Byzantine nodes, the BFT consensus is (f,t,n)-secure if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, $0 \leq f \leq t = \lfloor n/3 \rfloor$ for asynchronous BFT [26]. BFT consensus protocol meets the following two properties: *(1) Consistency:* Protocols in this model never violate their consistency properties during asynchronous periods. *(2) Liveness:* It is assured that the nodes will terminate the protocol [25].

We use PBFT [26], a popular BFT variant, for relevant discussion. PBFT proceeds via a succession of consensus rounds, each of which includes at least one view organized by a leader. The leader for each view is determined in a round-robin manner. PBFT consists of three phases [26]:

- *Prepare.* The leader accepts transactions from users (request), then initiates consensus by sending a "prepare" message to non-leader consensus nodes (replicas).
- *Response.* After receiving the "prepare" message, the node sends a "respond" message to all other nodes including the leader.
- *Commit.* A consensus node sends a "commit" message if it has collected more than $2f+1$ "respond" messages. If a node obtains more than $2f+1$ "commit" messages, a consensus is reached and the consensus node provides the execution result to users (reply).

2.2 MEV Attacks

An MEV [35,51,84,86] attacker can manipulate a transaction list, such as inserting new transactions, swapping transaction orders, and removing transactions, to determine the order of the transactions execution in its interest. While the blockchain consensus guarantees the uniqueness of block order, the transaction order within a block is entirely up to its creator, the leader node in BFT. As a result, a rational node may prioritize

the processing of certain transactions based on transaction fees. Such free manipulation can cause damage to the fairness of order-sensitive applications [35]. Most of the blockchains are vulnerable to MEV attacks [77] with the exception of some confidential ledgers [29, 73]. There are many types of MEV attacks, such as front-running, back-running, and sandwich attacks [77]. We target a kind of MEV attack that poses direct threat to runtime RNP, defined in Section 3.2.1.

2.3 Decentralized random beacon (DRB)

A DRB [43] provides a way to collaboratively agree on a (pseudo)random value without the involvement of a central party. To create a beacon in DRB, each node generates a partial beacon, together with a proof of its validity. Then, each node obtains the partial beacon from the other nodes and validates it. Once a node collects sufficient number of valid partial beacons, it computes the beacon. A k out-of-n DRB on a set of nodes $P = \{p_0, \cdots, p_{n-1}\}$ is specified as follows. For simplicity, we omit the proof of partial beacon:

- *Setup*. This is an interactive protocol run by the nodes in P to set up a random beacon committee. The protocol outputs a committee public key cpk, a list of secret keys $\{sk_0, ..., sk_{n-1}\}$ and their corresponding public keys $\{pk_0, ..., pk_{n-1}\}$.

- *PartialRand*. On input m, a secret key sk_i and a committee public key cpk, the algorithm computes a partial beacon $\sigma_{m,i}$. Output is $\sigma_{m,i}$.

- *CombRand*. On input a set $\{\sigma_{m,i}\}_{i \in I}$ of partial values from $\mid I \mid > k$ different nodes, and a committee public key cpk, this algorithm outputs a beacon value σ_m, or $\perp$.

- *VerifyRand*. On input m, a beacon value σ_m and a committee public key cpk, this algorithm verifies whether σ_m is valid by outputting 0 or 1.

The security requirements for a typical DRB protocol are as below [43]:

- *Pseudorandomness*. Pseudorandomness guarantees that generated beacon is indistinguishable from uniformly random value in the presence of active adversaries, which implies the unpredictability and bias-resistance [80, 87].

- *Uniqueness*. This requires the generated random beacon values are unique even if the adversary can access the secret keys of honest parties[1].

[1] Note that the uniqueness of a pseudorandom output directly provides strong bias-resistance [43].

3 Synopsis

For ease of illustration we consider a PBFT-based blockchain system with smart contracts, consisting of n nodes $P = \{p_0, \cdots, p_{n-1}\}$. A smart contract may require random numbers for the input to the underlying applications for fairness. When a blockchain user invokes the smart contract, it is desirable to get the execution result with a minimal cost in terms of fees and processing time.

We denote the *proposal* for block b from the consensus leader p_0 at view $v \geq 0$ as $\mathcal{L}_{\langle b,v,0 \rangle}$. A *proposal* is a candidate block from the leader which will be confirmed at the end of consensus if majority of nodes agree on it. Hereafter, we slightly abuse the notation b to also indicate the consensus round to generate block b.

3.1 Assumptions

We assume that adversary $\mathcal{A}$ can control up to $t = \lfloor n/3 \rfloor$ nodes, which may behave arbitrarily, i.e., being Byzantine faulty. We assume that $\mathcal{A}$ is static in the sense that it can only choose the nodes corrupt before the start of the PBFT protocol. $\mathcal{A}$ has complete control over a faulty node, including its network connections and operating systems. In other words, $\mathcal{A}$ knows all the secret keys of the node and can forge any message on behalf of the node. $\mathcal{A}$ can further monitor transactions and launch MEV attacks. If the leader node is corrupted, it can propose a transaction list that benefits the attacker most.

We assume that the underlying BFT consensus works as expected. In the system, a node that is not faulty throughout the consensus is considered to be honest and executes the BFT protocol as specified. The honest nodes are assumed to have stable network connections, adequate computing power, and sufficient storage space to process blockchain transactions. An honest node will not collude with a faulty one. It has secure network connections with other honest nodes. When an honest node receives a valid transaction, it will immediately broadcast the transaction to all other consensus nodes.

Out of Scope. This work focuses on the runtime RNP for BFT-based blockchains. RNPs for non-BFT blockchains are out of the scope of this work. For the security of smart contracts, we only consider the identified malicious revert (see Section 3.2.1) and the MEV attacks that pose direct threats to the runtime RNP. MEV attacks on other blockchain services, e.g., decentralized exchange [35, 77], are not covered in this work. The existing countermeasures could be taken for defense [9].

3.2 Problem Statement

3.2.1 Research Problem

Our research problem is how to securely provide random values to smart contracts in the BFT-based blockchain at

runtime. A runtime RNP is expected to meet the following requirements: *(i)* No adversary should be able to interrupt the RNP operations and predict the random number in advance. It also cannot interfere with the transaction execution according to the generated random number. In other words, the RNP should be immune to MEV attacks even without callback mechanisms. *(ii)* Smart contracts invoking transactions can request multiple random values during execution. *(iii)* The runtime RNP does not undermine the security and efficiency of the BFT-based blockchain. Namely, it should not introduce extra consensus steps to the existing ones and adversaries should not be able to exploit runtime RNP to compromise the fairness of smart contracts. We identify two potential risks to runtime RNP.

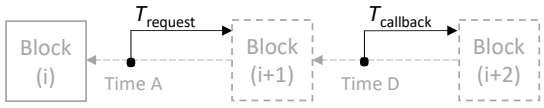
MEV attacks for runtime RNP. The MEV attack considered in this paper is illustrated in Figure 2b. once the beacon is known during the consensus, anyone, including the leader, may construct new transactions to exploit smart contracts that rely on random numbers generated from this beacon, such as predicting the lottery winning number. That is why existing RNPs use the callback mechanism to generate random values in future blocks, as shown in Figure 2a. However, callback RNP is an expensive approach for supplying random numbers, whereas runtime RNP aims to provide random numbers within the same consensus round as $T_{\texttt{request}}$ is generated, necessitating new mechanisms to protect against MEV attacks.

Threat of Malicious Revert. Blockchain transactions are atomic [7]. In other words, a transaction is considered to be successful if it is completed without error. Otherwise, the entire transaction is rolled back to undo the actions that have been performed. Atomic is critical to ensure the security of the smart contract state. However, in a blockchain system that supports runtime random numbers, an adversary could craft a malicious contract to call the victim contract and check the result against its expectations. If the result is not desirable, the malicious contract may revert the transaction intentionally. We call this Malicious Revert. The workflow of the Malicious Revert is depicted in Figure 3. We use an example of a "blind box" contract to demonstrate the Malicious Revert. A blind box contract C is used to distribute NFT tokens. Each token has a unique rarity, decided randomly when the token is claimed. Anyone can pay to get an NFT token. The adversary $\mathcal{A}$ could deploy an malicious contract to call C and then verify the rarity of the mined token when C returns. If $\mathcal{A}$ thinks the NFT token is rare, $\mathcal{A}$ keeps it; otherwise, $\mathcal{A}$ reverts the execution and gets the payment back. The adversary can repeat the operation until it obtains a desired token.

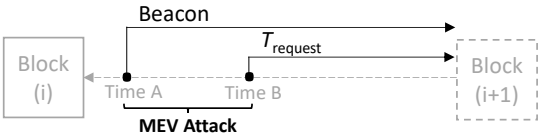
3.2.2 Security Goals

Here we summarize the security goals of BCRAND as below.

Goal 1. *Anti-MEV for RNPs.* An anti-MEV protocol ensures fair ordering by prioritizing transactions in a first-in-first-out fashion [52]. Let Δ be the maximum peer-to-peer latency between honest nodes in a blockchain network, T_i be a transaction that an honest node broadcasts to the blockchain network at time i, T_j be a transaction that an adversary node broadcasts at time j, s.t., $j \geq i+\Delta$. An anti-MEV protocol ensures that T_i is always executed earlier than T_j. However, existing solutions by fair ordering is complicated and may be an overkill for a RNP protocol [52], because the adversary $\mathcal{A}$ cannot perform MEV attacks against RNP as long as the random values remain unpredictable and unknown to $\mathcal{A}$ at the time the transaction list is generated. As a result, we consider a relaxed anti-MEV definition for RNPs.



(a) The workflow of callback based RNP, where $T_{\texttt{request}}$ is recorded in block i+1 and the $T_{\texttt{callback}}$ is recorded in block $\geq$ i+2. Random number generated at Time D is for $T_{\texttt{request}}$.



(b) Demonstration of how MEV attacks may happen for a runtime RNP. Random beacon is generated at Time A and $T_{\texttt{request}}$ is constructed at Time B, where Time A $<$ Time B.

Figure 2: The workflows of RNPs in the view of blocks. Block i is the latest persisted block, block i+1 and block i+2 are blocks to be generated.

Definition 3.1 (Relaxed Anti-MEV)**.** Let $\mathcal{L}_{\langle b,v,0\rangle}$ be a *proposal* from the leader node p_0 for block b at view v, T_i and T_j are any two transactions within $\mathcal{L}_{\langle b,v,0\rangle}$ and $i < j$. A RNP protocol is relaxed anti-MEV if it produces a random value after p proposes $\mathcal{L}_{\langle b,v,0\rangle}$ and the order of T_i and T_j cannot be modified.

Goal 2. *Secure Randomness Providing Function.* BCRAND should generate random values that are bias-resistant and unpredictable. Further, the beacon for each block should be verifiably unique. Specifically, we want to realize secure random number generator defined as follows.

Definition 3.2 (Secure Random Number Generator)**.** Let O be a random beacon for block b with a proposal $\mathcal{L}_{<b,v,0>}$. Assume T is a transaction included in $\mathcal{L}_{<b,v,0>}$ which requests m random numbers. $R = \{r_0, \cdots, r_{m-1}\}$ is the set of random

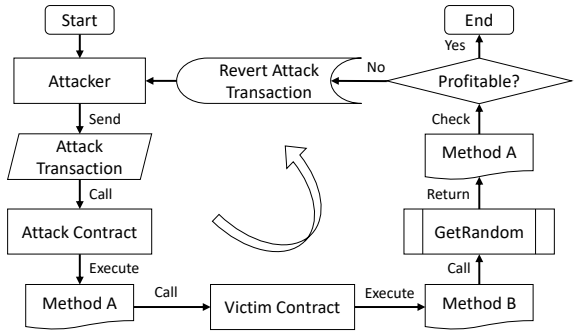


Figure 3: The Malicious Revert against runtime RNPs. Adversary repeatedly constructs malicious transactions to call the 'Method B' of victim contract from 'Method A' of an attack contract and checks the execution result of the victim contract against its expectation in 'Method A'.

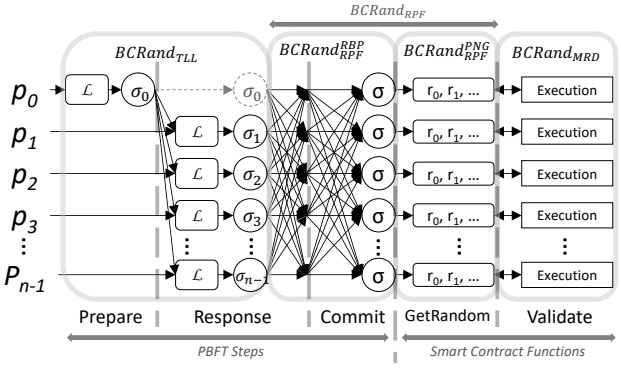


Figure 4: High-level BCRAND description mapping to PBFT phases and smart contract execution. $\mathcal{L}$ is the *proposal* from the leader. $\{\sigma_i\}$ are the *DRB* partial beacons, whereas σ is the beacon. $\{r_i\}$ are random numbers. Execution entails running smart contracts with transaction inputs on a virtual machine.

numbers generated from BCRAND_{RPF} for T. Then we say that an algorithm is a random number generator if it satisfies:

- *Unpredictability*. Any PPT $\mathcal{A}$ should not be able to generate (predict) any random number r_i for $i \in [0, m-1]$ provided for the transaction. That is, the probability of any PPT $\mathcal{A}$ correctly predicting a random number in R is negligible.
- *Bias-resistance.* Any PPT $\mathcal{A}$ cannot influence the provided random numbers for its own advantage.
- *Determinism*. If we fix a proposal $\mathcal{L}_{<b,v,0>}$ and a included transaction T that requests random numbers at certain time, then the random numbers in R, provided by this algorithm, are deterministic.
- *Liveness*. Any PPT $\mathcal{A}$ should not be able to prevent a random number from being generated.

Goal 3. *Irrevocable Execution.* To mitigate the Malicious Revert, transactions that trigger smart contract C should be prohibited from reverting transactions arbitrarily. This implies that the transaction's execution should be irrevocable; hence, we define irrevocable execution as follows:

Definition 3.3 (Irrevocable Execution)**.** Let C be a smart contract, T be a transaction that invokes C, and *res* be the result of C executing T. The execution of C is irrevocable if T is incapable of checking the value of *res*, thus unable to deliberately revert when *res* is not desired.

Note that we aim to achieve irrevocable execution by preventing cheating transactions from executing specific contract logic. This is not at odds with the atomic execution principle of the blockchain.

4 Runtime RNP Framework

BCRAND contains three major components: a) the anti-MEV attack BCRAND_{TLL}, which locks the transaction list during the consensus; b) the randomness providing function BCRAND_{RPF}, including $\text{BCRAND}^{RBP}_{RPF}$ and $\text{BCRAND}^{PNG}_{RPF}$; and c) the Malicious Revert defender $\text{BCRAND}_{\text{MRD}}$ that prevents the adversary from reverting $T_{\texttt{request}}$.

Transaction List Lock BCRAND_{TLL}. The BCRAND_{TLL} is a distributed lock system that prevents anyone from modifying the transaction list to defend against MEV attacks. Locking the transaction list ensures that the *proposal* $\mathcal{L}_{\langle b,v,0\rangle}$ from the leader p_0 for block b at view v is immutable during and after the beacon generation. When the leader proposes $\mathcal{L}_{\langle b,v,0\rangle}$, BCRAND_{TLL} obtains the hash value of the internal transaction list using a Merkle tree [65] (denote the hash used in Merkle tree as H), and then get a partial beacon σ_0 from that hash value with *DRB.PartialRand*. When a replica p_i receives a valid σ_0, it creates its own partial beacon σ_i.

Randomness Providing Function BCRAND_{RPF}. The randomness providing function consists of two steps: $\text{BCRAND}^{RBP}_{RPF}$ and $\text{BCRAND}^{PNG}_{RPF}$. $\text{BCRAND}^{RBP}_{RPF}$ generates a random beacon during each consensus round. The beacon is expected to be bias-resistance, unpredictable, and verifiably-unique so that no one can predict or manipulate it. $\text{BCRAND}^{RBP}_{RPF}$ is intended to aggregate multiple input signatures into a single one with *DRB.CombRand*. $\text{BCRAND}^{PNG}_{RPF}$ is a pseudorandom number generator. Since one transaction may trigger multiple requests for random numbers, $\text{BCRAND}^{PNG}_{RPF}$ is used to generate the requested random numbers from $\text{BCRAND}^{RBP}_{RPF}$ with the beacon O.

Malicious Revert Defender $\text{BCRAND}_{\text{MRD}}$. $\text{BCRAND}_{\text{MRD}}$ is a smart contract programming library that includes built-in functions to prevent transactions from performing Malicious Revert on contracts. To defend against Malicious Revert, the

DRB is a (k,n)-threshold decentralized random beacon protocol. Run *DRB.Setup* to generate a committee public key cpk and a secret key sk_i for each node. Let BFT be the (f,t,n)-secure BFT consensus and $0 \le f \le t < k \le 2t+1$, $b > 1$ be the current consensus round , $Q_{\langle b,v\rangle}$ be the set of partial beacons for view v of round b, O_b be the beacon for block b. $(\cdot)_{p_i}$ indicates the message $(\cdot)$ is signed by p_i. For each b, node p_i performs the following phases:
(1) **Prepare:** While in consensus round $b-1$, all nodes $p_i \in \mathcal{P}$ collect transactions from the blockchain network and catch them in transaction pool. (Leader only) When view v of consensus round b starts, leader p_0 generates a *proposal* $\mathcal{L}_{\langle b,v,0\rangle}$ and gets the partial beacon: $\sigma_{\langle b,v,0\rangle} \leftarrow \text{BCRAND}_{TLL}(\mathcal{L}_{\langle b,v,0\rangle})$, then broadcasts $(prepare, \mathcal{L}_{\langle b,v,0\rangle}, \sigma_{\langle b,v,0\rangle})_{p_0}$.
(2) **Response:** For a node p_i in round b at view v. When p_i receives a valid *prepare* from p for the first time, do the following:

- obtains the proposal $\mathcal{L}_{\langle b,v,0\rangle}$ and verifies the internal transactions, and gets the proposal's hash value $hash \leftarrow H(\mathcal{L}_{\langle b,v,0\rangle})$;
- verifies $\sigma_{\langle b,v,0\rangle}$ with *hash*, and then calculates the partial beacon $\sigma_{\langle b,v,i\rangle} \leftarrow \text{BCRAND}_{TLL}(\mathcal{L}_{\langle b,v,0\rangle})$;
- broadcasts *response* message $(response, \sigma_{\langle b,v,i\rangle})$.

(3) **Commit:** If receives *response* message from some $p_j \in \mathcal{P}$, $\texttt{Add}(Q_{\langle b,v\rangle}, \sigma_{\langle b,v,j\rangle})$. With $\texttt{Len}(Q_{\langle b,v\rangle}) > 2t$, get random beacon $O_{\langle b,v\rangle} \leftarrow \text{BCRAND}^{RBP}_{RPF}(Q_{\langle b,v\rangle})$. Broadcast commit message $(commit, \mathcal{L}_{\langle b,v,0\rangle}, O_{\langle b,v\rangle})_{p_i}$.
(4) **Validate:** For a transaction T from block b that triggers C, call $\text{BCRAND}_{\text{MRD}}(T)$ from C.
(5) **GetRandom:** For random beacon O and each transaction T that calls for random from block b , get random number $r \leftarrow \text{BCRAND}^{PNG}_{RPF}(T,O)$, and output r.

Figure 5: BCRAND: secure runtime random number provider framework. Each step corresponds to Figure 4.

smart contract C must restrict the invoking transaction T from doing the following actions: (a) calling C from a *delegate contract*; and (b) running scripts that may validate C's execution outcome. When contract developers create a smart contract C to call random numbers, they must execute $\text{BCRAND}_{\text{MRD}}$. When C is invoked by transaction T, $\text{BCRAND}_{\text{MRD}}$ first verifies whether C is the first smart contract invoked by T. If not, abort the execution. Then $\text{BCRAND}_{\text{MRD}}$ verifies whether the calling script of T contains extra script logic to validate the execution of C. Because each C has a unique calling script logic, $\text{BCRAND}_{\text{MRD}}$ requires a developer-supplied parameter specifying the length of the script.

4.1 Our Full Framework: BCRAND Protocol

We present BCRAND protocol while tolerating $f \le t < n/3$ Byzantine faulty nodes. The overall framework with respect to the PBFT is shown in Figure 4, where "Prepare", "Response", and "Commit" are corresponds to the original PBFT steps while "Validate" and "GetRandom" are functions utilized by smart contracts. Figure 5 details BCRAND protocols.

Prepare. (PBFT step) In this step, the leader constructs a transaction list and generates a *proposal* as expected. The leader is also required to generate a partial beacon and broadcasts it to the consensus network along with the *proposal*.

Response. (PBFT step) A replica receives and verifies the *proposal* and the partial beacon from the leader. If verification passes, the replica generates its partial beacon and broadcasts a response message.

Commit. (PBFT step) A consensus node calls $\text{BCRAND}^{RBP}_{RPF}$ to produce the beacon and then issues a commit message if a consensus node obtains over $2t$ valid responses. This is the end of the consensus.

Validate. (Smart contract function) Recall that C is a smart contract that implements $\text{BCRAND}_{\text{MRD}}$. When transaction T calls C, $\text{BCRAND}_{\text{MRD}}$ will verify if the calling script of the transaction has a valid format.

GetRandom. (Smart contract function) The interface through which smart contact C requests random values. For a transaction T that triggers C, $\text{BCRAND}^{PNG}_{RPF}$ is called to return the resulting random number r to the smart contract.

4.2 BCRAND-DBLS: A Concrete Runtime RNP Instance

We instantiate a concrete runtime RNP by utilizing the proposed BCRAND framework and distributed BLS scheme (DBLS; see Appendix D and Appendix E for detail) and denote it as BCRAND-DBLS. DBLS can be treated as a DRB and it has been used as a core building block in many DRB protocols [2, 15, 37]. One can easily replace DBLS with other DRB protocols into BCRAND for different characters. More DRB instantiations can be found in [43].

Setup. Run the $DBLS.KeyGen(1^\lambda)$ protocol to generate a BLS key pair for each consensus node.

BCRAND-DBLS$_{TLL}$. BCRAND-DBLS_{TLL} offers the function of BCRAND_{TLL}. When the leader proposes $\mathcal{L}_{\langle b,v,0\rangle}$ in BCRAND-DBLS, BCRAND-DBLS_{TLL} obtains the hash value of the internal transaction list using a Merkle tree [65], and then signs it using *DBLS* by $\sigma_0 \leftarrow DBLS.Sign(sk_p, \mathcal{L}_{\langle b,v,0\rangle})$. When a replica p_i receives σ_0, it verifies the signature by $DBLS.Verify(pk_p, \mathcal{L}_{\langle b,v,0\rangle}, \sigma_0)$. p_i creates a DBLS signature σ_i if the verification succeeds.

BCRAND-DBLS$^{RBP}_{RPF}$. $\text{BCRAND-DBLS}^{RBP}_{RPF}$ realizes the function of $\text{BCRAND}^{RBP}_{RPF}$ as shown in Algorithm 1. With a (k,n)-threshold DBLS signature, $\text{BCRAND-DBLS}^{RBP}_{RPF}$ aggregates the input signatures σ_i into σ when the number of signatures is equal to or greater than k. The aggregated signature σ is used as the random beacon.

Algorithm 1: BCRAND-DBLS$^{RBP}_{RPF}$.

Input: $\{\{\sigma_i\}|\sigma_i$ is the *DBLS* signature of $p_i\}$
Output: Random Beacon (O)
Data: The threshold of DBLS (k)

```
if |{σ_i}| < k then
    return
σ ← DBLS.Aggregate({σ_i})
Assign σ to O          // DBLS signature as random beacon
return O
```

BCRAND-DBLS$^{PNG}_{RPF}$ and BCRAND-DBLS$_{MRD}$. The counterpart instantiations of BCRAND$^{PNG}_{RPF}$ and BCRAND$_{\text{MRD}}$. BCRAND-DBLS does not add more instructions.

5 Security Analysis

In this section, we analyze the security of BCRAND. BCRAND aims to achieve the following goals described in Section 3.2.2: (1) anti-MEV for RNPs (**Goal 1**), (2) secure random number function (**Goal 2**), and (3) irrevocable execution (**Goal 3**). For simplicity, we use DRB to represent a (k,n)-threshold DRB scheme.

5.1 Security Proof for Goal 1

We first show that BCRAND can defend against MEV attacks on runtime RNPs, hence achieving **Goal 1**.

Lemma 5.1. *If the underlying hash function H used in Merkle tree is collision-resistant defined in Appendix B, then any PPT adversary $\mathcal{A}$ cannot change the order of the transaction list inside a proposal, $\mathcal{L}_{<b,v,0>}$, with the same root hash unless with negligible probability.*

Proof. We use $O_{\mathcal{L}}$ to denote the order of transaction lists in proposal $\mathcal{L}_{<b,v,0>}$ and denote its root hash as $h_{O_{\mathcal{L}}}$, s.t., $h_{O_{\mathcal{L}}} = H(O_{\mathcal{L}})$. We denote the event that $\mathcal{A}$ can generate two different order of same transactions lists, $O_{\mathcal{L}}$ and $O'_{\mathcal{L}}$, with same root hash as MK-collision$_{\mathcal{A},\pi^*}$. That is, $H(O_{\mathcal{L}}) = H(O'_{\mathcal{L}})$, s.t., $O_{\mathcal{L}} \neq O'_{\mathcal{L}}$. Then such an $\mathcal{A}$ in our system can be directly used to break collision-resistant of H by using $O_{\mathcal{L}}$, $O'_{\mathcal{L}}$ and $h_{O_{\mathcal{L}}}$. We have,

$$\Pr[\text{Hash-collison}_{\mathcal{A},H}] \leq \Pr[\text{MK-collison}_{\mathcal{A},\pi^*}].$$

As derived from B, we can have that $\Pr[\text{MK-collison}_{\mathcal{A},\pi^*}] \leq \text{negl}(\lambda)$, which ends our proof. □

Now we prove our scheme achieves the security goal defined in Goal 1.

Theorem 5.1. *We assume that H used to construct Merkle tree is collision resistant defined in Def. B, for a view $v \geq 0$ of consensus in round $b > 1$, our system achieves security goal 1 in Definition 3.1.*

Proof. Given the fact that our system generates random beacon and numbers after p_0 proposing $\mathcal{L}_{<b,v,0>}$ and locking the transaction list and by Lemma 5.1, BCRAND achieves relaxed anti-MEV defined in Definition 3.1. □

5.2 Security Proof for Goal 2

We prove that BCRAND achieves **Goal 2** by reducing the security of BCRAND to the security of underlying BCRAND$^{RBP}_{RPF}$ and BCRAND$^{PNG}_{RPF}$. We prove the security of BCRAND$^{RBP}_{RPF}$ and BCRAND$^{PNG}_{RPF}$ in Lemma 5.2 and Lemma 5.3 respectively, which leads to security proof by Theorem 5.2 for **Goal 2**. We give formal proof as follows.

Lemma 5.2. *If any PPT $\mathcal{A}$ corrupts no more than t nodes where $t < k$ and the underlying DRB achieves pseudorandomness and uniqueness, then* BCRAND$^{RBP}_{RPF}$ *satisfies unpredictability, verifiability, uniqueness and liveness.*

Proof. We prove these properties separately.

- *Unpredictability*. If $\mathcal{A}$ can generate a valid random beacon value by corrupting $t < k$ nodes, then it can be used to break pseudorandomness defined in Section 2.3 for the threshold DRB scheme. We denote this event as Protocol-forge$_{\mathcal{A},\pi^*}$. Specifically, there exists a simulator $\mathcal{S}$ which acts as an adversary in the pseudorandomness experiment in DRB scheme and acts as a challenger in our system. If $\mathcal{A}$ can correctly predict random beacon σ for proposal $\mathcal{L}$, $\mathcal{S}$ can directly use it to distinguish the output by the DRB scheme from a random string.

  Specifically, $\mathcal{S}$ answers $\mathcal{A}$'s query on any proposal $\mathcal{L}^j$ by forwarding it in DRB's experiment as input and returns its partial beacon σ^j_i to $\mathcal{A}$. After several queries, $\mathcal{A}$ outputs a forged beacon σ^* for a proposal $\mathcal{L}^*$, which has never been queried before. $\mathcal{S}$ forwards $\mathcal{L}^*$ to its challenger with getting a full beacon, σ'. $\mathcal{S}$ outputs 1 if $\sigma' = \sigma^*$. We use $\mathcal{S}^{DRB} = 1$ and $\mathcal{S}^{Random} = 1$ to denote events that $\mathcal{S}$ correctly predicts the output from *DRB* scheme or random value separately.

  If $\mathcal{S}$ is interacting with DRB scheme, then $\mathcal{A}$ interacts with the real system built from DRB. We have,

  $$\Pr[\text{Protocol-predict}^{DRB}_{\mathcal{A},\pi^*} = 1] \leq \Pr[\mathcal{S}^{DRB} = 1].$$

  If $\mathcal{S}$ gets response from truly random values, we use Protocol-predict$_{\mathcal{A},\pi^*}$ to denote the event that $\mathcal{A}$ correctly generates a random beacon while interacting with another system built from real random beacons. We have,

  $$\Pr[\text{Protocol-predict}^{Random}_{\mathcal{A},\pi^*}] \leq \Pr[\mathcal{S}^{Random} = 1].$$

  As DRB achieves pseudorandomness property, we have that

  $$|\Pr[\mathcal{S}^{DRB} = 1] - \Pr[\mathcal{S}^{Random} = 1]| \leq \text{negl}(\lambda).$$

Besides, $\Pr[\mathcal{S}^{Random} = 1] = \text{negl}(\lambda)$ as the probability of correctly predicting a true random value is negligible. Thus we have,

$$\Pr[\text{Protocol-predict}^{DRB}_{\mathcal{A},\pi^*}] \leq \text{negl}(\lambda).$$

That is, any adversary, who can break pseudorandomness of $\text{BCRAND}^{PNG}_{RPF}$, only has negligible probability.

- *Verifiability*. As the proposal $\mathcal{L}_{\langle b,v,0\rangle}$ and public key *pk* in DRB is public, the full random beacon from $\text{BCRAND}^{RBP}_{RPF}$ can be verified by running *VerifyRand*.
- *Uniqueness*. As the underlying *DRB* achieves uniquess, for a fixed proposal, the random beacon generated from *DRB* achieves uniqueness.
- *Liveness*. This property is guaranteed as any $\mathcal{A}$ cannot corrupt more than $t < k$ parties. As a result, it cannot prevent the random beacon generation process, which completes the proof.

□

Lemma 5.3. *If* $\text{BCRAND}^{RBP}_{RPF}$ *achieves pseudorandomness, verifiability, uniqueness and liveness, then* $\text{BCRAND}^{PNG}_{RPF}$ *achieves unpredictability, bias-resistance, determinism and liveness defined in Definition 3.2.*

- *Unpredictability. This is directly guaranteed by the unpredictability property of* $\text{BCRAND}^{RBP}_{RPF}$ *and pseudorandomness of* $\text{BCRAND}^{PNG}_{RPF}$.
- *Determinism. As the random beacon O for a fixed proposal generated by* $\text{BCRAND}^{RBP}_{RPF}$ *is unique, the pseudorandom numbers generated by* $\text{BCRAND}^{PNG}_{RPF}$ *for a fixed transaction are deterministic.*
- *Bias-resistance. This property is provided by the properties of unpredictability and deterministic. As the beacon, O, can be generated by* $k+1$ *parties, any* $\mathcal{A}$ *with corrupting no more than* $t < k$ *parties cannot influence the final result of generating O. Thus, the random numbers cannot be biased.*
- *Liveness. This property can be derived from liveness in* $\text{BCRAND}^{RBP}_{RPF}$ *and BFT consensus, which ends our proof.*

Theorem 5.2. *Assume underlying DRB is secure under definition 2.3 and* $\text{BCRAND}^{PNG}_{RPF}$ *achieves pseudorandomness,* BCRAND *(*BCRAND_{RPF} *actually) achieves security goal 2 in Def. 3.2.*

Proof. This can be guaranteed from Lemma 5.2 and Lemma 5.3. □

5.3 Security Proof for Goal 3

BCRAND accomplishes **Goal 3** by implementing $\text{BCRAND}_{\text{MRD}}$ to defend against Malicious Revert.

Theorem 5.3. BCRAND_{MRD} *achieves irrevocable execution under Definition 3.3.*

Proof. For a smart contract C that triggers runtime random number calls and has $\text{BCRAND}_{\text{MRD}}$ implemented. A PPT-adversary $\mathcal{A}$ can perform Malicious Revert on C in two ways: (1) Adversary $\mathcal{A}$ deploys an attack contact C' with the intent of launching a Malicious Revert on C from C'. For an attack transaction T from $\mathcal{A}$ that calls C', the execution of T will fail. This is because when C' calls C, C checks whether it is the first smart contract triggered. $\mathcal{A}$ controls at most t Byzantine nodes and cannot change the smart contract execution logic. Therefore, C will directly revert the transaction without calling any random number. (2) $\mathcal{A}$ verifies the outcome of C's execution in the transaction script logic after calling C. It must add extra logic to the transaction script, which will also be detected by $\text{BCRAND}_{\text{MRD}}$ and causes the execution to fail. Due to T's inability to verify the execution result of C, $\mathcal{A}$ is not able to exploit C with Malicious Revert. Therefore $\text{BCRAND}_{\text{MRD}}$ achieves irrevocable execution. □

6 Implementation

We implemented a BCRAND prototype on the neo blockchain platform [67]. It integrates NFT, distributed storage, Oracles, and built-in virtual machines. It also supports C# for developing smart contracts. The consensus algorithm of neo is the delegated BFT (dBFT), an optimized version of PBFT [26]. We implemented BCRAND into dBFT by adding a few more fields to the messages of the classic three phases, i.e. `prepare`, `response`, and `commit`, without altering its original consensus function. To make the random seed accessible by the neo virtual machine, we change the "nonce" field in the neo header to 32 bytes to store the random seed.

Smart Contract Programming Interface. We extend the neo smart contract with a system runtime method called `GetRandom()`. When a contract invokes `GetRandom()`, it returns a 32-byte unsigned "BigInteger". And every time the contract calls it, the function produces a different random integer.

To avoid Malicious Revert, we evaluate the calling contract and the script's size. To make this process more user-friendly, we incorporate this logic into a user-accessible interface named `FAUDRequire(int size)`. Each method that calls `GetRandom()` may begin with `FAUDRequire(int size)` and takes an argument specifying the anticipated transaction script size. Source code can be found in Appendix G.3.

Implementation of BCRAND-DBLS$^{PNG}_{RPF}$. We implement BCRAND-DBLS$^{PNG}_{RPF}$ with *SHA*256 hash algorithm. BCRAND-DBLS$^{PNG}_{RPF}$ has internal variables *salt* and *index* which are singleton instances that can hold their assigned values across the execution of all transactions in block b. When a transaction T invokes a random number call, if it is the first random number call from T, BCRAND-DBLS$^{PNG}_{RPF}$ gets and saves its hash value to h_T, initializes *salt* to empty, and sets *index* to 0, then, BCRAND-DBLS$^{PNG}_{RPF}$ generates the random number by hashing the xor value of h_T, O, and *index* with *SHA*256: $salt = SHA256(h_T \oplus O \oplus index)$, then increases *index* by one and assign *salt* to the output value r. random number calls from T are fulfilled with the hash value of the *salt*. More detail of BCRAND-DBLS$^{PNG}_{RPF}$ is illustrated in Algorithm 2 in Appendix E.

6.1 Applications

We described the applications that rely on random numbers in order to illustrate the advantage of BCRAND.

Fair NFT Distribution. Loot [72] is one of the most successful NFTs deployed on Ethereum. The rarity of every Loot token is determined by the random numbers, which in turn are created by hashing on-chain data. As a result, anyone interested in acquiring a Loot token is able to see its content prior to claiming it, allowing cheating players to identify the rare bags. To demonstrate the fairness of the NFT distribution, we migrated the Loot contract to neo and randomly assign the rarity of each bag when users claim Loot tokens, guaranteeing that tokens are distributed fairly among all participants.

Multiple Random Number Requests. Neoverse is a "blind box" smart contract that enables buyers to purchase Blind Boxes without knowing what is inside until after opening them in a subsequent transaction. We re-implemented Neoverse to illustrate the ability of BCRAND to generate as many random values as needed in a single transaction by opening multiple Blind Boxes at runtime.

Fair Gaming Outcome. It is vital for a blockchain game contract to convince users to participate in prize giveaways by demonstrating that winners are selected using true random values. We constructed a rock-paper-scissors (RPC) contract that allows participants to play this game with the contract by sending a transaction containing the user's shape. When the RPC contract executes the user transaction, it generates a shape by calling `GetRandom()`, and then compares it to the user's shape. Every time a user participates, he or she must bid 1 GAS, GAS is the token in neo, as a stake in the RPC contract; if the user wins, the RPC contract pays back to the user 2 GAS; otherwise, the contract retains the user's stake.

Comparison. To compare BCRAND-DBLS with callback RNP solutions, we implemented a comparison application that contains three functions: `Runtime()` which calls `GetRandom()` directly to work as BCRAND-DBLS, and two other functions `Callback()` and `Fulfill()` to simulate the workflow of callback-based RNPs. `Callback()` issues a random number request and `Fulfill()` accepts the random number from a simulated RNP. Both `Runtime()` and `Callback()` take one parameter indicating how many random values are required. `Fulfill()` takes multiple parameters, including a random value, proof of the random value, and a request ID. When multiple random numbers are requested, `Runtime()` calls `GetRandom()` multiple times while `Fulfill()` calls the cryptographic function provided by CryptoLib [66], a native contract, "CryptoLib.Sha256" to expand the received random number. For simplicity, we denote `Callback()` and `Fulfill()` together as BCRAND-DBLS$_{callback}$.

7 Evaluation

In this section, we show the performance of BCRAND and demonstrate the advantage of BCRAND by comparing BCRAND with other RNPs. We implemented BCRAND with a quad-core 3.6 GHz Intel(R) E3-1275 v5 CPU [50] and 32 GB of memory. The operating system is Ubuntu 20.04 TLS with the Linux kernel version 5.4.0-91-generic, and the SDK is .NET 6.0.1. Our engineering workload is in the Appendix F.

7.1 Application Transaction Cost

Table 2 summarizes the transaction costs related to running the developed applications. In neo, the network fee is proportional to the length of the transaction script, while the system fee is determined by the OpCodes that a transaction executes. **Loot::**tokenURI provides a Loot token with a randomly determined rarity to the user. **Neoverse::**UnBoxing purchases and opens one blind box, and **Neoverse::**BulkUnBoxing purchases and opens 5 blind boxes. **RPC::**OnNEP17Payment produces a random shape and compares it to the one provided by the user. This evaluation demonstrates that smart contracts can conduct random number-related operations inexpensively with only one transaction.

Table 2: Applications Transaction Fee (GAS).

Method	Network Fee	System Fee
Loot::tokenURI	0.00593250	0.20694257
Neoverse:: UnBoxing	0.00119552	0.07313472
Neoverse:: BulkUnBoxing	0.00125752	0.36183988
RPC:: OnNEP17Payment	0.00616260	0.06588677

7.2 Transaction Fee Cost

We compare the fees to request multiple random numbers between BCRAND-DBLS and BCRAND-DBLS$_{\texttt{callback}}$. Since BCRAND-DBLS$_{\texttt{callback}}$ needs two transactions to complete a random number request, we compare the cost

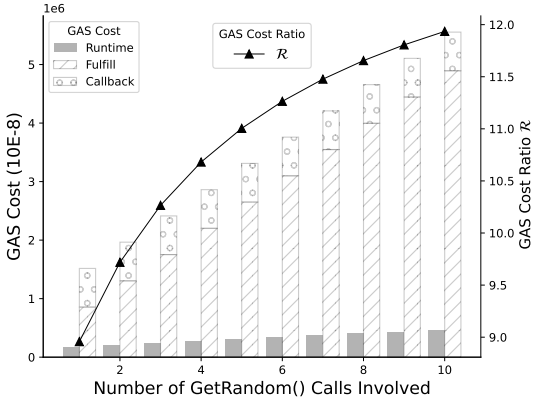


Figure 6: GAS cost when calling `GetRandom()`. The left y axis is the total GAS consumption, while the right y axis is the GAS cost ratio $\mathcal{R} = (\texttt{Callback} + \texttt{Fulfill})/\texttt{Runtime}$.

of BCRAND-DBLS with the sum of two transactions of BCRAND-DBLS$_{\texttt{callback}}$. Generating multiple random numbers with BCRAND-DBLS$_{\texttt{callback}}$ is implemented in C# on the neo platform according to Chainlink VRF[2]. The evaluation result is shown in Figure 6. We can see that if one transaction only requests one random number, the fee to request one random number with BCRAND-DBLS is 0.00169291 GAS, which is only 11% of the cost of issuing a random number request for BCRAND-DBLS$_{\texttt{callback}}$ (0.01516372 GAS). If every transaction requests 10 random numbers, then BCRAND-DBLS saves 91.6% GAS.

GAS Cost of BCRAND-DBLS$_{MRD}$. The cost to run the Malicious Revert defender is less than 0.000001 GAS, which is negligible in comparison with the transaction cost.

7.3 Blockchain Overhead

To evaluate the impact of RNPs on the blockchain ledger, we define the size of data other than $T_{\texttt{request}}$ that has to be written onto the blockchain to complete a random number request as blockchain overhead. Let the transaction size be l, n be the number of $T_{\texttt{request}}$ transactions, h be the average number of $T_{\texttt{request}}$ in each block. While processing n $T_{\texttt{request}}$, the blockchain overhead of BCRAND-DBLS$_{callback}$ is $l * n$, as for every `Callback()`, it requires a callback transaction `Fulfill()` to feed the random number back to the smart contract. The blockchain overhead of BCRAND-DBLS is $32n/h$, because the random number beacon in every block header is 32 bytes. Since Chainlink VRF [27] is one of the most influential RNP solutions, we use it as the baseline. We evaluate the blockchain overhead with the transaction size $l = 193$ bytes (the size of a typical transfer transaction in neo) and $l = 624$ bytes (the size of a callback transaction in Chainlink VRF [40]). As shown in Figure 7, if 0.04% of

[2]Chainlink VRF [28] is the most influential RNP solution, which has processed about 7 million random number requests for multiple chain projects.

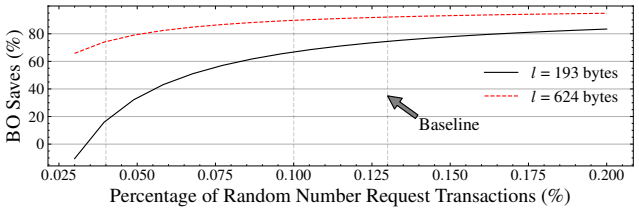


Figure 7: Blockchain overhead (BO) that BCRAND-DBLS saves compared to BCRAND-DBLS$_{callback}$. Baseline is the percentage of Chainlink VRF $T_{\texttt{callback}}$ on Ethereum, which is around 0.13% [40, 49].

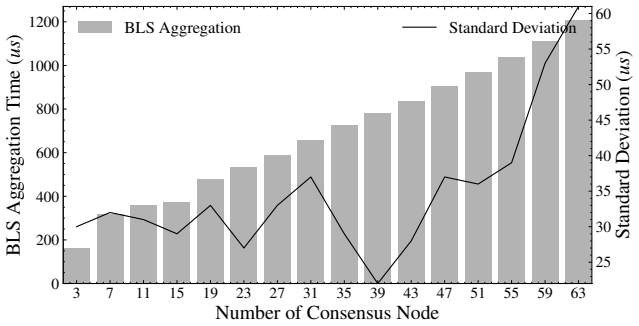


Figure 8: The BLS signature aggregation time.

total transactions are $T_{\texttt{request}}$, i.e., one $T_{\texttt{request}}$ for every 5 blocks, then BCRAND-DBLS saves about 17% blockchain overhead versus BCRAND-DBLS$_{\texttt{callback}}$ for $l = 193$ bytes and 72% for $l = 624$ bytes; if 0.1% transactions are $T_{\texttt{request}}$, BCRAND-DBLS saves at least 67% blockchain overhead for $l = 193$ bytes and 89% for $l = 624$ bytes.

7.4 Evaluation on BLS signature

We evaluate the overhead of BLS signature aggregation. Since we use BLS to generate the random beacon, we evaluate the aggregation time of σ. Figure 8 shows the aggregation time in log scale under k-out-of-n threshold mode where the number of BLS nodes is n and we set k to $\lceil (n+1)/2 \rceil$. When there are 7 consensus nodes (neo blockchain) and $k = 4$, the time cost of aggregating BLS signatures is about 318 μs; when the number of consensus nodes is increased to 23 (comparable to 21 nodes in EOS), the time cost is about 533 μs. Adding BCRAND-DBLS to the neo blockchain only incurs a negligible 0.002% more consensus time.

To evaluate the DKG setup, we implemented DKG in a 7-node neo private network deployed on Azure. The Azure nodes run on Intel (R) Xeon (R) Platinum 8272CL CPU 2.6 GHz with Ubuntu18.04.6 TLS. The ttl between nodes is about 57 ms. The evaluation result is shown in Figure 9 with 2,000 DKG setups. Most of them can finish in about 1 s. But there are cases where the setup takes longer (e.g. 13% needs 2 s). This is mainly affected by the communication cost

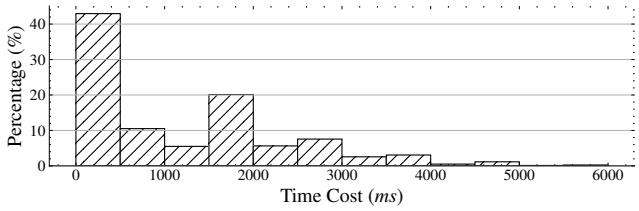


Figure 9: BLS setup time cost in a testnet with 7 consensus nodes.

of collecting messages from all participants. Since the DKG setup is separate from the consensus and runs only when the consensus committee needs to update, the execution of DKG will not introduce significant overhead to the consensus layer.

7.5 RNP Solutions Comparison

BCRAND-DBLS provides an efficient runtime RNP for BFT-based blockchain smart contracts. To demonstrate the advantages of BCRAND-DBLS, we compare BCRAND-DBLS with a host of current random number projects in Table 3. We compare the features among random beacons [14, 24, 30, 45, 87, 87], RNP-enabled blockchain projects [2, 15–18], and random oracles [28, 68]).

Platform Consensus. As shown in Table 3 the table, we can see that most of the protocols are BFT based. The main reason is that to generate random values in a distributed environment, nodes have to interactively communicate with one another to reach consensus. Chainlink VRF, as an oracle, is platform independent and provides randomness for both BFT-based [21] and PoW-based [40] blockchain platforms. Automata [68] is a blockchain middleware, specifically built for Ethereum. As BCRAND-DBLS is built on top of PBFT, it can be easily ported to any PBFT variants blockchain project.

Method. To avoid a faulty node from blocking random number generation/utilization, HDrand, RandRiper, Dfinity, Elrond, and BCRAND-DBLS use the DRB to generate random numbers, whereas Klaytn and Harmoney use verifiable functions to ensure the system generates deterministic random values. Other than threshold signatures and verifiable functions, several projects, including Secret, Chainlink VRF, and Automata, make use of the trusted execution environment (TEE) [5, 8, 63], a hardware-protected isolated execution environment, to generate random values in a discreet and unbiased manner within the TEE. TEE, on the other hand, is a hardware solution; it requires an RNP to operate on TEE-enabled devices.

# of Random Numbers. We compare the ability of RNPs to generate random numbers. Due to the fact that random beacon projects such as RandHerd, RandHound, and BRandRiper concentrate only on beacon generation, the available random value is constrained by the number of beacons O. While Secret and Elrond provide built-in pseudorandom functions that can extend O into numerous random numbers, the random value they can supply is only upper bounded by the consensus (GAS limitation for example), which we represent as ∞. Because BCRAND-DBLS uses $\text{BCRAND}_{RPF}^{PNG}$ to safely create random numbers from the random beacon provided by $\text{BCRAND}_{RPF}^{RBP}$, the number of random numbers can be generated by BCRAND-DBLS is likewise constrained by the consensus.

Runtime. Due to MEV and Malicious Revert, none of the RNPs used by random beacons or blockchain projects can generate random values within a single consensus round, or runtime in our terminology. Automata is unique among the compared projects. It is the only one that can deliver runtime random numbers to smart contracts. Automata is a centralized solution for Ethereum platform. It generates random numbers in TEE and uses Ethereum EIP712 [78] to encapsulate a user transaction and its requested random numbers inside an EIP712 transaction. Although Automata is theoretically vulnerable to Malicious Revert, it is unlikely in practice since it is a centralized, closed solution that only operates on their own services. Compared with Automata, BCRAND is an RNP solution that focuses on BFT-based open blockchain platforms and does not rely on special hardware.

In summary, BCRAND-DBLS is currently the only safe runtime random number provider for BFT-based blockchains.

8 Related Work

Random Number Providers for Smart Contract. Existing works on random beacons adopt different cryptographic primitives and threat models [13, 24, 44, 45, 53, 80, 82]. Some projects [30, 31, 71] are built with homomorphic encryption, but homomorphic encryption introduces heavy overhead to the system. Hedera Hashgraph [55] is a fault-tolerant RNP algorithm for a public directed acyclic graph. There are also projects running in a permissionless setting [1, 4, 15, 18, 28, 36, 37, 41, 55]; unfortunately, they do not support runtime RNP, while BCRAND is a runtime RNP for smart contracts.

Hardware-based Random Number Providers. An ASIC-based VDF [19] is used as a source of randomization in ETH 2.0 [39]. Chainlink VRF [28] enables smart contracts to interact with a TEE-based oracle to obtain randomness and cryptographic proof. Secret [18] uses a TEE-based secret oracle to provide on-chain random service. The random beacon service of oasis [73] provides unbiased randomness on each epoch. All these methods rely on specific hardware and introduces extra trust to the blockchains. BCRAND, in contrast, is a hardware-independent RNP solution.

BFT-based Blockchain Projects. Several existing works employ BFT-family protocols as consensus. Celo [74] is built on the Istanbul BFT. Kadena [75] originally uses the novel ScalableBFT. Solana's [85] Tower Consensus relies on a

Table 3: Comparison of RNPs for blockchain.

Protocol	Platform Consensus	Method(s)	Resistance (t)	# random values (r)	Runtime
Drand [24]	PABFT	Threshold SecretBLS	$t < n/2$	$O(O)$	✗
HERB [30]	∅	Threshold ElGamal	$t < n/3$	$O(O)$	✗
RandChain [45]	Sequential PoW	PoW	$t < n/3$	$O(O)$	✗
RandHerd [87]	BFT	Threshold Schnorr	$t < n/3$	$O(O)$	✗
RandHound [87]	BFT	Client based, PVSS	$t < n/3$	$O(O)$	✗
BRandRiper [14]	BFT	VSS, q-SDH	$t < n/2$	$O(O)$	✗
Dfinity [2]	BFT	Threshold BLS	$t < n/2$	∞	✗
Secret [18]	DPoS	Scrt-RNG,TEE	$t < n/2$	∞	✗
Elrond [15]	Secure PoS	BLS,onchain data	$t < n/3$	∞	✗
Klaytn [17]	Istanbul BFT	VRF	$t < n/3$	$O(O)$	✗
Harmoney [16]	Fast BFT	VRF,VDF	$t < n/3$	$O(O)$	✗
⋆Chainlink VRF [28]	∅	VRF, TEE	$t < n/2$	$O(O)$	✗
⋆Automata [68]	∅	VRF, TEE	$t < n/2$	∞	✓
BCRAND-DBLS$_{callback}$	BFT	∅	$t < n/3$	$O(O)$	✗
BCRAND-DBLS	BFT	BCRAND, DBLS	$t < n/3$	∞	✓§

In the table, n denotes the number of the consensus nodes, t is the maximum number of Byzantine nodes permitted in the system, and O denotes the beacon. **Resistance** refers to the system's tolerance for Byzantine faults. ⋆ is the off-chain RNP Oracle. O is the beacon generated by random beacon protocol. ∞ means number of random numbers is upper bounded by consensus. §BCRAND-DBLS is the first runtime smart contract RNP solution on BFT-based blockchain.

BFT mechanism. Multi-Level BFT is used by Theta [69]. Klaytn [17] uses an optimized version of Istanbul BFT. The Helium [46] is based on a variant of the HoneyBadgerBFT [64]. The EOS [38] utilizes a BFT modeled consensus mechanism. Harmony proposes Fast BFT, which reduces communication costs by using BLS aggregate signature [16]. As BCRAND is a BFT-based runtime RNP framework, it can potentially be deployed on these projects.

Overall, none of the existing works described above support runtime RNP for blockchain.

9 Discussion

Scalability. BCRAND is designed for BFT smart contracts. BFT does not scale well due to its communication complexity $O(n^2)$. Our method was tested in a real-world BFT-based blockchain setting, i.e., the neo platform with 7 consensus nodes and 21 committee nodes. There exists other BFT-based blockchain projects with similar settings, such as EOS with 21 nodes. Note that BCRAND does not introduce any new communication steps to BFT. It only adds a few bytes to the existing BFT messages. As shown in our evaluation in Section 7, the time of BCRAND to aggregate signatures is negligible compared to 15 seconds consensus intervals on neo.

Integration of BCRAND to existing BFT-based blockchain. It is straightforward to apply BCRAND to existing BFT-based blockchain projects to support runtime RNP without incurring hard forks. See Appendix H for more details.

Future Work. Note that BCRAND is intended for BFT-based blockchain systems where a leader node can propose a list of transactions and the rest of the network can lock to that list for consensus. BCRAND is not applicable to blockchain platforms that adopt alternative consensus, such as Ethereum [22] and Bitcoin [65], where nodes are free to work on their own transaction list. Adding runtime RNP support to permissionless consensus will be our future work.

10 Conclusion

Providing reliable randomness to smart contracts is critical to various blockchain functions. Many DApps depend on randomness for their utility and security. Though current callback-based random number providers improve security, they suffer from many other issues, such as expensive onchain costs and delayed processing time. In this work, we explore and design BCRAND, a novel runtime RNP framework for smart contracts on BFT-based blockchains. We call it runtime because it can securely provision random numbers using one round of consensus instead of two in callback-based protocols. To demonstrate the performance, we implement an instance of BCRAND with distributed threshold BLS signature and incorporate it into the consensus layer without affecting the efficiency. BCRAND has been proven to be secure against MEV attacks on RNPs. In addition, we also identify the Malicious Revert and design an effective mitigation to enhance the security of BCRAND. As a result, BCRAND is the first of its kind for blockchains that need secure, reliable, and runtime random number provision.

References

[1] Random in xrp. https://xrpl.org/random.html.

[2] ABRAHAM, I., MALKHI, D., NAYAK, K., AND REN, L. Dfinity consensus, explored. *Cryptology ePrint Archive* (2018).

[3] ABRAHAM, I., MALKHI, D., NAYAK, K., REN, L., AND YIN, M. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 106–118.

[4] AHEAD, A. B. Poa 2.0: Vechain's verifiable random function library in golang, Mar 2020.

[5] AMD. AMD ESE/AMD SEV. https://github.com/AMDESE/AMDSEV. Accessed: 2020-04-27.

[6] AMIR, Y., COAN, B., KIRSCH, J., AND LANE, J. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing 8*, 4 (2010), 564–577.

[7] ANTONOPOULOS, A. M., AND WOOD, G. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[8] ARM. Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone, 2019-12-13.

[9] ASAYAG, A., COHEN, G., GRAYEVSKY, I., LESHKOWITZ, M., ROTTENSTREICH, O., TAMARI, R., AND YAKIRA, D. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), IEEE, pp. 55–65.

[10] AUBLIN, P.-L., MOKHTAR, S. B., AND QUÉMA, V. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013), IEEE, pp. 297–306.

[11] BANO, S., SONNINO, A., AL-BASSAM, M., AZOUVI, S., MCCORRY, P., MEIKLEJOHN, S., AND DANEZIS, G. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936* (2017).

[12] BESSANI, A., SOUSA, J., AND ALCHIERI, E. E. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), IEEE, pp. 355–362.

[13] BHAT, A., KATE, A., NAYAK, K., AND SHRESTHA, N. Optrand: Optimistically responsive distributed random beacons. *Cryptology ePrint Archive* (2022).

[14] BHAT, A., SHRESTHA, N., LUO, Z., KATE, A., AND NAYAK, K. Randpiper–reconfiguration-friendly random beacons with quadratic communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 3502–3524.

[15] BLOCKCHAIN, E. Random numbers in smart contracts. https://docs.elrond.com/developers/developer-reference/random-numbers-in-smart-contracts/, 2022.

[16] BLOCKCHAIN, H. Harmony randomness. https://docs.harmony.one/home/general/technology/randomness, 2022.

[17] BLOCKCHAIN, K. Consensus randomness. https://docs.klaytn.com/klaytn/design/consensus-mechanism, 2022.

[18] BLOCKCHAIN, S. Secret randomness. https://docs.scrt.network/dev/developing-secret-contracts.html#randomness, 2022.

[19] BONEH, D., BONNEAU, J., BÜNZ, B., AND FISCH, B. Verifiable delay functions. In *Annual international cryptology conference* (2018), Springer, pp. 757–788.

[20] BONEH, D., LYNN, B., AND SHACHAM, H. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security* (2001), Springer, pp. 514–532.

[21] BSCSCAN. Vrfcoordinator. https://bscscan.com/address/0x747973a5A2a4Ae1D3a8fDF5479f1514F65Db9C31#analytics, 2022.

[22] BUTERIN, V., AND ETHEREUM.ORG. A next-generation smart contract and decentralized application platform. *white paper 3*, 37 (2014).

[23] CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference* (2001), Springer, pp. 524–541.

[24] CACHIN, C., KURSAWE, K., AND SHOUP, V. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology 18*, 3 (2005), 219–246.

[25] CACHIN, C., AND VUKOLIĆ, M. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[26] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OsDI* (1999), vol. 99, pp. 173–186.

[27] CHAIN.LINK. https://chain.link/. https://chain.link/. accessed: 2020-08-18.

[28] CHAINLINK. Get a random number. https://docs.chain.link/docs/get-a-random-number/, 2021.

[29] CHENG, R., ZHANG, F., KOS, J., HE, W., HYNES, N., JOHNSON, N., JUELS, A., MILLER, A., AND SONG, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), IEEE, pp. 185–200.

[30] CHERNIAEVA, A., SHIROBOKOV, I., AND SHLOMOVITS, O. Homomorphic encryption random beacon. *Cryptology ePrint Archive* (2019).

[31] CHERNIAEVA, A., SHIROBOKOV, I., AND SHLOMOVITS, O. Homomorphic encryption random beacon. Cryptology ePrint Archive, Report 2019/1320, 2019. https://ia.cr/2019/1320.

[32] CHOHAN, U. W. Non-fungible tokens: Blockchains, scarcity, and value. *Critical Blockchain Research Initiative (CBRI) Working Papers* (2021).

[33] CLEMENT, A., WONG, E. L., ALVISI, L., DAHLIN, M., AND MARCHETTI, M. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI* (2009), vol. 9, pp. 153–168.

[34] COINMARKETCAP.COM. Today's cryptocurrency prices by market cap. https://coinmarketcap.com/, Mar. 2022.

[35] DAIAN, P., GOLDFEDER, S., KELL, T., LI, Y., ZHAO, X., BENTOV, I., BREIDENBACH, L., AND JUELS, A. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234* (2019).

[36] DEVS, T. C. Safe practice of tron solidity smart contracts: Implement random numbers in the contracts, Mar 2020.

[37] DRAND. Drand/drand: a distributed randomness beacon daemon - go implementation.

[38] EOSIO. Eosio website. https://eos.io/, Nov. 2022.

[39] ETHEREUM.ORG. Ethereum upgrades (formerly 'eth2').

[40] ETHERSCAN. Chainlink vrf. shorturl.at/klrzL, 2022.

[41] FETCH.AI. Launching our random number beacon on binance smart chain, Oct 2020.

[42] FILECOIN. Collaboration with the ethereum foundation on vdfs. https://filecoin.io/blog/posts/collaboration-with-the-ethereum-foundation-on-vdfs/, 2019.

[43] GALINDO, D., LIU, J., ORDEAN, M., AND WONG, J.-M. Fully distributed verifiable random functions and their application to decentralised random beacons. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)* (2021), IEEE, pp. 88–102.

[44] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), Springer, pp. 295–310.

[45] HAN, R., YU, J., AND LIN, H. Randchain: Decentralised randomness beacon from sequential proof-of-work. *IACR Cryptol. ePrint Arch. 2020* (2020), 1033.

[46] HELIUM.COM. Helium documentation, 2022.

[47] HELLIAR, C. V., CRAWFORD, L., ROCCA, L., TEODORI, C., AND VENEZIANI, M. Permissionless and permissioned blockchain diffusion. *International Journal of Information Management 54* (2020), 102136.

[48] HTTPS://BLOKS.IO/. eosluck.bank. https://bloks.io/account/eosluck.bank, 2018.

[49] HTTPS://YCHARTS.COM/. Ethereum transactions per day. https://ycharts.com/indicators/ethereum_transactions_per_day, 2022.

[50] INTEL. Intel® xeon® processor e3 v5 family. https://ark.intel.com/content/www/us/en/ark/products/88177/intel-xeon-processor-e3-1275-v5-8m-cache-3-60-ghz.html, 2019-12-3.

[51] JUDMAYER, A., STIFTER, N., SCHINDLER, P., AND WEIPPL, E. Estimating (miner) extractable value is hard, let's go shopping! *Cryptology ePrint Archive* (2021).

[52] KELKAR, M., ZHANG, F., GOLDFEDER, S., AND JUELS, A. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference* (2020), Springer, pp. 451–480.

[53] KELSEY, J., BRANDÃO, L. T., PERALTA, R., AND BOOTH, H. A reference for randomness beacons: Format and protocol version 2. Tech. rep., National Institute of Standards and Technology, 2019.

[54] KOKORIS-KOGIAS, E., JOVANOVIC, P., GASSER, L., GAILLY, N., SYTA, E., AND FORD, B. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 583–598.

[55] KRASNOSELSKII, M., MELNIKOV, G., AND YANOVICH, Y. Distributed random number generator on hedera hashgraph. In *2020 the 3rd International Conference on Blockchain Technology and Applications* (2020), pp. 7–11.

[56] KWON, J. Tendermint: Consensus without mining. *Draft v. 0.6, fall 1*, 11 (2014).

[57] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[58] LEV-ARI, K., SPIEGELMAN, A., KEIDAR, I., AND MALKHI, D. Fairledger: A fair blockchain protocol for financial institutions. *arXiv preprint arXiv:1906.03819* (2019).

[59] MARTIN, J.-P., AND ALVISI, L. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing 3*, 3 (2006), 202–215.

[60] MAVIS, S. Official axie infinity whitepaper. https://whitepaper.axieinfinity.com/, Nov. 2021.

[61] MAVIS, S. Paid whitepaper. https://docsend.com/view/jdbdpza9d9nehnf2, Jan. 2021.

[62] MCCORRY, P., SHAHANDASHTI, S. F., AND HAO, F. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security* (2017), Springer, pp. 357–375.

[63] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *HASP@ISCA* (2013), p. 10.

[64] MILLER, A., XIA, Y., CROMAN, K., SHI, E., AND SONG, D. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), pp. 31–42.

[65] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2016.

[66] NEO PROJECT, T. Neo native contract.

[67] NEO.ORG. Neo smart economy. https://neo.org/, 2022.

[68] NETWORK, A. Automata network. https://www.ata.network, 2021.

[69] NETWORK, T. Theta network.

[70] NEWECONOLAB. neo-ns. https://github.com/NewEconoLab/neo-ns/blob/master/dapp_nns_register_sell/nns_register_sell.cs, 2018.

[71] NGUYEN-VAN, T., NGUYEN-ANH, T., LE, T.-D., NGUYEN-HO, M.-P., NGUYEN-VAN, T., LE, N.-Q., AND NGUYEN-AN, K. Scalable distributed random number generation based on homomorphic encryption. In *2019 IEEE International Conference on Blockchain (Blockchain)* (2019), IEEE, pp. 572–579.

[72] PROJECT, L. Loot contract source code. https://etherscan.io/address/0xff9c1b15b16263c61d017ee9f65c50e4ae0113d7#code, 2021.

[73] PROJECT, O. P. The oasis blockchain platform. https://docsend.com/view/aq86q2pckrut2yvq, June 2020.

[74] PROJECT, T. C. Celo randomness: Celo docs. https://docs.celo.org/celo-codebase/protocol/identity/randomness.

[75] PROJECT, T. K. Kadena whitepaper. https://docs.kadena.io/basics/whitepapers/overview.

[76] QIAN, Y. Randao: Verifiable random number generation, 2017.

[77] QIN, K., ZHOU, L., AND GERVAIS, A. Quantifying blockchain extractable value: How dark is the forest? *arXiv preprint arXiv:2101.05511* (2021).

[78] REMCO BLOEMEN, L. L. Eip-712: Ethereum typed structured data hashing and signing. https://eips.ethereum.org/EIPS/eip-712.

[79] SCHINDLER, P., JUDMAYER, A., HITTMEIR, M., STIFTER, N., AND WEIPPL, E. Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness.

[80] SCHINDLER, P., JUDMAYER, A., STIFTER, N., AND WEIPPL, E. Hydrand: Efficient continuous distributed randomness. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 73–89.

[81] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[82] SIMIĆ, S. D., ŠAJINA, R., TANKOVIĆ, N., AND ETINGER, D. A review on generating random numbers in decentralised environments. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)* (2020), IEEE, pp. 1668–1673.

[83] SLOWMIST. Slowmist hacked eos ecosystem. https://hacked.slowmist.io/en/?c=EOS&page=5, 2021.

[84] SLOWMIST. Slowmist hacked statistics. https://hacked.slowmist.io/en/statistics/?c=all&d=all, 2021.

[85] SOLANA.COM. Solana documentation, 2022.

[86] STANKOVIC, S. What is mev? ethereum's invisible tax explained. https://cryptobriefing.com/what-is-mev-ethereums-invisible-tax-explained/, 2021.

[87] SYTA, E., JOVANOVIC, P., KOGIAS, E. K., GAILLY, N., GASSER, L., KHOFFI, I., FISCHER, M. J., AND FORD, B. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), Ieee, pp. 444–460.

[88] SZABO, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[89] TOMESCU, A., CHEN, R., ZHENG, Y., ABRAHAM, I., PINKAS, B., GUETA, G. G., AND DEVADAS, S. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 877–893.

[90] VERONESE, G. S., CORREIA, M., BESSANI, A. N., AND LUNG, L. C. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems* (2009), IEEE, pp. 135–144.

[91] VERONESE, G. S., CORREIA, M., BESSANI, A. N., LUNG, L. C., AND VERISSIMO, P. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers 62*, 1 (2011), 16–30.

[92] YIN, M., MALKHI, D., REITER, M. K., GUETA, G. G., AND ABRAHAM, I. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 347–356.

[93] ZHANG, Y., KASAHARA, S., SHEN, Y., JIANG, X., AND WAN, J. Smart contract-based access control for the internet of things. *IEEE Internet of Things Journal 6*, 2 (2018), 1594–1605.

A Hardness Assumptions

A.1 Computational Diffie-Hellman (CDH) Problem

Let $\mathbb{G}$ be a cyclic group with prime order q with generator g.For $a,b \in Z_q^*$, given a tuple (g, g^a, g^b), find the element g^{ab}.

A.2 Decisional Diffie-Hellman (DDH) Problem

Let $\mathbb{G}$ be a cyclic group with prime order q with generator g.For $a,b \in Z_q^*$, given (g, g^a, g^b, g^z), decide whether $g^z = g^{ab}$.

A.3 Gap Diffie-Hellman (GDH) Group

A GDH group is a group where CDH problem is hard but DDH problem is easy.

B Collision-resistant Hash Function

For defining collision-resistant hash functions, we first define an experiment.

Collision Finding Experiment Hash-collision$_{\mathcal{A},H}$. For a hash function $H : \{0,1\}^* \to \{0,1\}^{l(\lambda)}$ with security parameter λ, a PPT adversary $\mathcal{A}$ is asked to output a pair m and m', $\mathcal{A}(H,\lambda) \to (m,m')$[3]. The experiment outputs 1 if the output pair satisfies, $m \neq m'$ and $H(m) = H(m')$.

Collision-resistant hash function. We say that a hash function H is collision-resistant if for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that,

$$\Pr[\text{Hash-collision}_{\mathcal{A},H} = 1] \leq \text{negl}(\lambda).$$

C Distributed Key Generation (DKG)

A DKG protocol [44, 89] allows a set of n parties to generate unbiased, random keys. The outcome of an execution of a DKG protocol is a secret key sk, shared among parties using a secret-sharing scheme. Assume we have a group G under modular p and g is its generator with order q. Formally: p_i chooses a random polynomial $f_i(z)$ over $\mathbb{Z}_q$ of degree t: $f_i(z) = a_{i0} + a_{i1}z + \cdots + a_{it}z^t$. p_i broadcasts $A_{ik} = g^{a_{ik}} \; mod \; p$ for $k \leq t$. Denote a_{i0} by z_i and A_{i0} by y_i. p_i computes the shares $x_{ij} = f_i(j) \; mod \; q$ for $j \leq n$ and sends x_{ij} secretly to party p_j. p_j verifies the shares it received $i \leq n : g^{x_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} \; mod \; p$ and broadcasts a complaint against p_i if the check fails for an index i. Then, p_i reveals the share x_{ij} for each complaining party p_j. If any of the revealed shares fails the verification, p_i is disqualified. T is the set of non-disqualified parties. p_j sets his share of the secret as

$$s_j = \prod_{i \in T} x_{ij} \; mod \; q$$

The secret value sk itself is not computed by any party, but it is equal to $sk = \prod_{i \in T} z_i \; mod \; q$. The public value $y = \prod_{i \in T} y_i \; mod \; p$. The public verification values are computed as $A_k = \prod_{i \in T} A_{ik} \; mod \; p$ for $k \leq t$.

Boneh-Lynn-Shacham (BLS) signature scheme [20]. Let e denote a bilinear map $G_1 \times G_1 \to G_2$, where G_1 is a GDH group of prime order q with generator g. A signer with a secret key $sk \in_R \mathbb{Z}_q^*$ computes a signature $\sigma = \mathsf{H}_1(m)^{sk} \in G_1$ on message m, where $\mathsf{H}_1 \colon \{0,1\}^* \to G_1$ is modeled as a random oracle in its security analysis. Given the public key $pk = g^{sk} \in G_1$ with a message-signature pair, (m,σ), a verifier checks the validity via $e(\mathsf{H}_1(m), pk) \stackrel{?}{=} e(\sigma, g)$.

D Distributed BLS (DBLS)

A k-out-of-n threshold BLS is described as follows where we include a dealer for ease of writing only. We use $\mathcal{P} =$

[3] Sometimes we omit security parameter in $\mathcal{A}$'s input as it can be implicitly addressed in the description of H.

$\{p_0, \ldots, p_{n-1}\}$ to denote the set of n parties involved in this scheme.

- $KeyGen(1^\lambda) \rightarrow (pk, sk, pk_1, sk_1, \ldots, pk_n, sk_n)$. It is performed by a fully-trust dealer (Looking ahead, our protocol deploys a *DKG* algorithm to distribute a secret key instead of a real dealer.). Taking security parameter λ as input, the dealer generates a random secret key $sk \in_R \mathbb{Z}_q^*$ and public key $pk = g^{sk}$. It then splits sk into $sk_1, sk_2, ..., sk_n$ using Sharmir's secret sharing [81] and distributes them to n participants. Their corresponding public keys are $pk_i = g^{sk_i}$, which are made public
- $Sign(sk_i, m) \rightarrow (\sigma_i, m)$. To produce a signature on m, a participant i computes a signature share $\sigma_i = \mathsf{H}_1(m)^{sk_i}$ with his/her secret share sk_i.
- $Aggregate(\sigma_1, \sigma_2, \cdots, \sigma_k) \rightarrow \sigma$. The aggregator finds a valid subset K of shared signatures, s.t., $K = \{\sigma_1, \ldots, \sigma_k\}$ and $|K| = k$, by verifying $e(\mathsf{H}_1(m), pk_i) \stackrel{?}{=} e(\sigma_i, g)$. Then, the aggregator can compute the full signature as

$$\sigma = \prod_{i \in K} \sigma_i^{w_i},$$

  where $w_i = \prod_{j, j \neq i} \frac{j}{j-i}$.
- $Verify(pk, m, \sigma) \rightarrow 0/1$. This algorithm takes a public key pk, a message m, and a signature σ as input, and outputs 0 or 1 by checking $e(\mathsf{H}_1(m), pk) \stackrel{?}{=} e(\sigma, g)$.

This (k, n)-threshold signature assumes the existence of a fully trusted dealer, which is unreasonable in a distributed blockchain setting. To achieve fully distributed, we use *DKG* protocol to replace $KeyGen(1^\lambda)$ in our system.

E Security Requirements of *DBLS*

A secure *DBLS* should satisfy correctness and unforgeability under chosen-message attack.

Correctness. If for $(pk, sk, pk_1, sk_1, \ldots, pk_n, sk_n) \leftarrow KeyGen(1^\lambda)$, any message m in message space and any subset of secret shares $S = \{s_i\}$ satisfying $|S| >= k$, the aggregated signature σ, s.t., $\sigma \leftarrow Aggregate(\sigma_1, \cdots, \sigma_k)$, $(\sigma_i, m) \leftarrow Sign(sk_i, m)$, satisfies $Verify(m, \sigma, pk) = 1$, we say that this scheme satisfies correctness.

Existential Unforgeability Experiment Sig-forge$^{eu\text{-}cma}_{\mathcal{A},\pi}$. Given a distributed signature scheme π, any PPT adversary $\mathcal{A}$ that can control a subset K' of signers $\mathcal{S}_i$ where $|K'| < k$, consider following experiment.

- Running $KeyGen(1^\lambda)$ to generate (pk, sk) and $\{(pk_i, sk_i)\}_{i \in [1,n]}$ for each signer $\mathcal{S}_i$.
- $\mathcal{A}$ is given $(pk, \{pk_i\}_{i \in [1,n]}, \{sk_i : \mathcal{S}_i \in K'\})$.
- $\mathcal{A}$ is given oracle access and get partial signature shares for message m from honest party i and we denote it as $\sigma_{m,i}$. Suppose $\mathcal{A}$ can choose up to q_s messages for this oracle and we denote this set of messages as Q.
- $\mathcal{A}$ outputs a pair (m^*, σ^*).
- The experiment outputs 1 if $Verify(pk, m^*, \sigma^*) = 1$ and $m^* \notin Q$.

Unforgeable under chosen-message attack. A distributed signature scheme π is unforeable under chosen-message attack of for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl$(\cdot)$ such that, Pr[Sig-forge$^{eu\text{-}cma}_{\mathcal{A},\pi} = 1] \leq$ negl(λ).

Algorithm 2: BCRAND$^{PNG}_{RPF}$ used in our instantiation BCRAND-DBLS

Input: Beacon (O), Transaction (T)
Output: Random Number (r)
Data: Transaction hash (h_T), Global variable ($salt, index$)

```
hash ← H(T)                    // Calculate the transaction hash
if h_T is null or h_T ≠ hash then
    h_T ← SHA256(T)            // Cache the transaction hash
    index = 0
else
    salt ← SHA256(h_T ⊕ O ⊕ index)
    index = index + 1
Assign salt to r
return r
```

F Lines of Code

We implemented a BCRAND-DBLS prototype in C# on neo v3.1. In summary, we added 383 lines of code to the neo-core to implement the BCRAND-DBLS$^{PNG}_{RPF}$, 3,679 lines of code to the consensus module to implement the BCRAND-DBLS$_{TTL}$ and BCRAND-DBLS$^{RBP}_{RPF}$, and 47 lines of code to the smart contract compiler to add the smart contract programming interface. Table 4 summarizes the LoCs of applications.

Table 4: Sample neo smart contracts developed with BCRAND-DBLS. For each, we specify the implementation language, development effort in Line-of-Code (LoC), and number of files.

Application	Language	LoC	Files
Loot	C#	621	6
RPC	C#	369	4
Neoverse	C#	596	8
Comparison	C#	100	2

G Code

G.1 Code Samples that are vulnerable to MEV attacks

```
/**
 * Requests randomness
 */
function random() private view returns(uint){
    return uint(keccak256(abi.encodePacked(block.
        difficulty, now, players)));
}
```

Listing 1: Solidity sample code that uses the Blockchain data to generate random numbers from Loot [72].

```
/// Get random number
public static byte[] GetRandom()
{
    var header = Blockchain.GetHeader(Blockchain.
        GetHeight());
    return Sha256(header.ConsensusData)
}
```

Listing 2: C# sample code that uses the Blockchain data to generate random number [70]. It is vulnerable to MEV attacks.

G.2 Code Sample of Callback Based Random Oracle

```
/**
 * Requests randomness
 */
function getRandomNumber() public returns (bytes32
     requestId) {
    require(LINK.balanceOf(address(this)) >= fee,
        "Not enough LINK - fill contract with
        faucet");
    return requestRandomness(keyHash, fee);
}
/**
 * Callback function used by VRF Coordinator
 */
function fulfillRandomness(bytes32 requestId,
    uint256 randomness) internal override {
    randomResult = randomness;
}
```

Listing 3: Solidity sample code that calls random number from Chainlink with callback [28]. This is costly to use.

G.3 Prototype of BCRAND$_{MRD}$

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
internal static void FAUDRequire(int size)
{
    // Disable call from another contract
    ExecutionEngine.Assert(Runtime.EntryScriptHash
         == Runtime.CallingScriptHash, "FAUD:
        Contract call is not allowed.");
    // Prevent malicious transaction script
    ExecutionEngine.Assert((Transaction)Runtime.
        ScriptContainer).Script.Length <= size), "
        FAUD:Transaction script length error.");
}
```

Listing 4: The internal logic of “FAUDRequire(int size)".

H Instruction for BCRAND Integration

Developers can pick any eligible DRB protocol and PNG algorithm and then add a few fields to existing consensus messages. An interface for smart contract needs to be added to request random numbers and calculate random numbers upon user calls. Detailed steps are shown bellow, where setup process is not included:

- Implement cryptographic DRB and PNG libraries.
- After generating the transaction list, calculate a pair of partial beacon and a proof. Extend the consensus message “Prepare” to include the partial beacon and proof.
- Add partial beacon verify logic in the “Prepare” message verification sector. And calculate a partial beacon and its proof if the partial beacon from the leader is authentic. Extend the “Respond” message to contain the partial beacon and proof.
- Add partial beacon verify logic in the “Response” message verification sector. Check valid number of partial beacons and generates a beacon if valid number of partial beacon is collected.
- Extend the “Commit” message to contain the beacon. Since the beacon is generated via consensus, no need to contain the proof of the beacon.
- Save the beacon to the Block using either hardfork or no hardfork method, for instance:
  - *No-hardfork:* Save the beacon to the block state while persisting the block.
  - *Hardfork:* Add a field in the header to save the beacon.
- Implement an interface for smart contract to provide random numbers.
- Implement a Malicious Revert defender library with routines to validate the transaction script format.