BCRAND: A Secure Runtime Random Number Generator for Smart Contracts

***Abstract*—A random number generator (RNG) is essential to a blockchain ecosystem by providing a continuous public source of randomness to a wide range of blockchain applications, such as DeFi, NFT, decentralized gaming, and the Metaverse. The state-of-the-art solutions rely on additional callback transactions to generate random numbers for BFT-based blockchains. However, the extra cost associated with the callback function is non-negligible, e.g. increased fees, delayed processing time, and bloated on-chain storage.**

In this work, we propose BCRAND, a secure RNG that produces random numbers within the same consensus round where the request for random numbers is made. Thus, BCRAND provides the random numbers for smart contracts at runtime. BCRAND first locks the transaction list and enables consensus nodes to collectively generate an unbiased, unpredictable, and verifiably unique random seed by distributed threshold BLS signature at the end of the BFT consensus. In addition to being resistant to MEV attacks, we identify a new threat, revert-after-use attack, in this challenging runtime RNG setting and propose effective mitigations. By removing the callback, BCRAND significantly outperforms the existing RNGs. Our experiment shows that the average fee cost of a random number request in BCRAND is reduced by 99.967% compared to callback-based solutions. BCRAND also generates 89% less on-chain data if 0.1% of the total transactions request random numbers.

1. Introduction

Smart contracts [1], [2] are at the core of the decentralized applications (DApps) ecosystem. Many DApps, including DeFi and decentralized games, require access to randomness provided by public random number generators (RNGs). For instance, decentralized games use randomness to determine the winner [3]; in NFTs, randomness is translated to uniqueness [4] and rarity [5]; a legal toolkit relies on random jurors to resolve disputes [6].

A RNG 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 RNGs, blockchain data was used as an entropy source. Since the blockchain data is public, the random numbers generated by on-chain RNGs can be computed in advance, leading to the Miner Extractable Value or Maximum Extractable Value (MEV) attacks [7]–[10]. Many lottery contracts hosted on the EOS blockchain [11] were compromised as a result, where attackers profited by correctly predicting the winning number [12]. Loot [5],

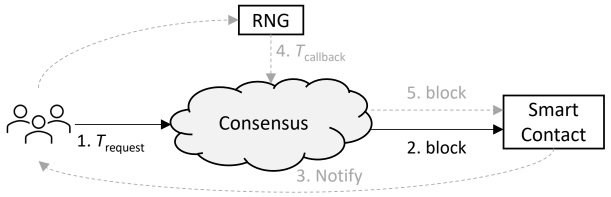


Figure 1: RNG based on callbacks. The solid lines represent the request for a random number, and the dashed lines represent the callback process to generate random numbers for the smart contract.

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 deemed one of the most prevalent types of blockchain attacks [10].

In the literature, callback-based RNG protocols were proposed to mitigate MEV attacks [13]–[21]. 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 1. User parameters are first recorded in the transaction $T_{\texttt{request}}$ to initiate the random number request during a block interval. Later, the second transaction $T_{\texttt{callback}}$ is needed in the following block time to produce the random number. With $T_{\texttt{request}}$ and $T_{\texttt{callback}}$ in different consensus rounds, the attacker cannot predict the random number in advance.

As promising the callback-based RNGs are, they face the following main issues:

(i) Expensiveness: Each transaction on the blockchain incurs a cost in the form of transaction fees. Because callback-based RNGs require two transactions to complete a random number request, the user initiating the request must pay transaction fees for both $T_{\texttt{request}}$ and $T_{\texttt{callback}}$.

(ii) Latency: In order to record user parameters prior to the generation of random numbers, $T_{\texttt{request}}$ and $T_{\texttt{callback}}$ must exist in separate blocks, i.e., it takes at least two blocks to produce a random number for a smart contract [22]–[24].

(iii) Storage Overhead: While a random number can be only a few bytes in size, a transaction used to request the number contains a variety of information, such as the coordinator's address, the token's address, the key hash, the fee, and the function code. When two transactions are needed in the callback RNG, the storage overhead is also doubled. For example, $T_{\texttt{callback}}$ in Chainlink [25] is 624 bytes, while the random number is only 32 bytes.

In this work, we propose a novel RNG that achieves the following goals. *(i)* Low transaction fees: paying a low transaction fee to acquire random numbers; *(ii)* Low latency: random numbers are fulfilled within one round of consensus; *(iii)* Low storage overhead: minimal data needs to be stored on blockchain.

We present BCRAND, a secure runtime RNG for smart contracts for BFT-based blockchain. BCRAND does not require the transaction $T_{\texttt{callback}}$ [26] to work. The main idea of BCRAND is to generate a random number seed for each block during the consensus. Then the seed is fed into a pseudorandom function to generate the 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 further significantly reduces the time required for randomness-related smart contact functions. Meanwhile, BCRAND also reduces the relevant transaction fees and on-chain storage overhead.

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 number seed 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 intentionally revert the transaction if the result is not profitable. How to defend BCRAND against this *revert-after-use* attack (see Section 3.2.1)?

BCRAND adopts the following methods to address the three 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 attack. After locking the transaction list, we generate a verifiable random beacon using the distributed threshold BLS signature [27] for each block. Finally, BCRAND provides a library for developing the revert-after-use defender for smart contracts. To assess BCRAND's effectiveness, we implement a prototype of BCRAND and a baseline BCRAND$_{callback}$ which simulates the behavior of a callback-based RNG on neo [28]. We discover that if 0.1 percent of transactions are $T_{\texttt{request}}$, BCRAND saves 89 percent of on-chain data and reduces the monetary cost to get a random number by 99.967%. To the best of our knowledge, BCRAND is the first *secure runtime RNG* designed for smart contracts of blockchain projects running on BFT consensus. The main contributions of this paper are summarized as follows:

- We design, implement, and evaluate BCRAND, the first secure runtime random number generator for smart contracts. By generating a random seed for each block via consensus, smart contracts that use BCRAND can fulfill a large number of (if not bounded by consensus) random number requests at runtime.
- BCRAND is a novel distributed random number generation protocol based on the BFT consensus. As a result, the protocol by nature stays tolerant to the number of faulty nodes guaranteed by the underlying consensus. Even without the callback function, BCRAND is designed to be resistant to MEV attacks.
- We identify a new revert-after-use (RAU) attack in smart contracts and propose a RAU defense mechanism to mitigate its impact.
- We evaluate four applications to demonstrate BCRAND's ability to efficiently supply random numbers to a diverse range of services. We also compare BCRAND to prior work to show its excellent performance in random number generation.

Note that BCRAND is intended for use with 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 [1] and Bitcoin [29], where nodes are free to work on their own transaction list.

The rest of the paper is structured as follows. Section 2 introduces the background. Section 3 summarizes the system. Section 4 delves into the details of the design. Section 5 analyzes security, while Sections 6, 7, and 8 elaborate on the implementation, evaluation, and related work respectively. Finally, Section 9 concludes the paper.

2. Background

2.1. Blockchain

Blockchain technology creates a tamper-proof source of truth by utilizing distributed consensus algorithms 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 immediately 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 [30], [31]. The Ethereum smart contract [1] is a well-known example. A typical smart contract operates inside a virtual machine [32] in order to provide a separate environment for contract execution, which is segregated from the network, file system, and I/O services. The smart contracts utilize user transactions as input.

2.1.2. Byzantine Fault Tolerance (BFT). BFT is a family of consensus algorithms designed to overcome the Byzantine Generals' Problem in distributed settings [33]. 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 [34]. When a consensus round ends, a new block is formed to represent the network consensus. For a BFT protocol containing a total of n nodes, the protocol 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 [34] and $0 \leq f \leq t = \lfloor n/2 \rfloor$ for synchronous BFT [35].

As one of the most widely used consensus algorithms, at the time of writing, 17 out of the top 80 chain projects adopt BFT-based consensus [36], such as Theta [37], neo [28], and EOS [11].

2.2. MEV Attacks

An MEV [7]–[10] attacker can manipulate a transaction list, such as inserting new transactions, swapping transaction orders, and removing transactions. While the blockchain consensus guarantees the uniqueness of block orders, the order of transactions within a block is entirely up to the leader node. Therefore, a rationale blockchain node may prioritize the transactions attached with higher fees. Such free manipulation can cause damage to the fairness of order-sensitive applications [8].

2.3. Technical Preliminaries

2.3.1. Distributed Key Generation (DKG). A DKG protocol [38], [39] allows a set of n parties to generate unbiased, random keys. The outcome of a run of a DKG protocol is a secret key sk, but shared among parties using a secret-sharing scheme. 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 shared 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 $k \leq t$.

2.3.2. Distributed Boneh–Lynn–Shacham (BLS) Signature. Let e denote a bilinear map $G_1 \times G_1 \rightarrow G_2$, where G_1 is a cyclic group of prime order q and we assume that CDH problem is hard on G_1. In BLS signature scheme [27], a signer with a secret key $sk \in_R \mathbb{Z}_q^*$ computes a signature $\sigma = \mathsf{H}(m)^{sk} \in G_1$ on message m, where H is a secure hash function from $\{0,1\}^*$ to G_1. Given the public key $pk = g^{sk} \in G$, a verifier checks the signature via $e(H(m), pk) \overset{?}{=} e(\sigma, g)$. Using standard secret sharing technique, we can derive a t-out-of-n threshold signature from the original BLS scheme. It consists of four algorithms as follows:

- *KeyGen(1^λ).* It is performed by a full-trust dealer. Taking as input the security parameter, the dealer generates a random secret key $sk \in_R \mathbb{Z}_q^*$ and public key $pk = g^{sk}$. It then splits sk into $sk_1, sk_2, ..., sk_n$ with the secret sharing technique. Finally, the dealer distributes the shares to the n participants and makes $g^{sk_1}, g^{sk_2}, ..., g^{sk_n}$ public accordingly.
- *Sign(sk_i, m).* To produce a signature on m, a participant i computes a signature share $\sigma_i = \mathsf{H}(m)^{sk_i}$ with his/her secret share sk_i.
- *Aggregate(σ_0, $\sigma_1, \cdots,$ σ_n).* The aggregator verifies each σ_i via $e(H(m), g^{sk_i}) \overset{?}{=} e(\sigma_i, g)$ and finds a valid subset K with size k. Now, the aggregator can compute the final signature as

$$\sigma = \prod_{i \in K} \sigma_i^{\mathcal{L}_i^K(0)} = \mathsf{H}(m)^{\sum_{i \in K} sk_i \mathcal{L}_i^K(0)} = \mathsf{H}(m)^{sk}$$

  , where $\mathcal{L}_i^K(0)$ is a Lagrange polynomial.
- *Verify(pk, m, σ).* This algorithm takes a public key pk, a message m, and a signature σ as input, and outputs '1' or '0' by checking $e(\mathsf{H}(m), pk) \overset{?}{=} e(\sigma, g)$.

This (k, n)-threshold signature assumes the existence of a fully trusted dealer. However, it is unreasonable in the distributed blockchain setting. To achieve a distributed (k, n)-threshold BLS signature, we use the *DKG* protocol to replace *KeyGen(1^λ)* such that the shares can be generated in a distributed manner.

3. Synopsis

We consider a BFT-based blockchain system with smart contracts, $\mathcal{P} := \{p_1, ..., p_n\}$ consisting of n nodes. For the sake of fairness for all parties involved, a smart contract may require the input of random numbers into the underlying applications. When a blockchain user invokes the smart contract, he or she anticipates paying minimal fees to obtain the execution result with a single transaction.

Our protocol 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 by the BFT consensus. We denote a block as b, and the $proposal$ for block b from the consensus leader p at view $v \geq 0$ as $\mathcal{L}_{\langle b,v,p \rangle}$. A *proposal* is a candidate block from the leader p and will be confirmed at the end of consensus if majority of nodes agree on it. For simplicity, we also use b to represent the consensus round to generate block b.

3.1. Assumptions

In the considered blockchain system, we assume that an 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 to be corrupted before the start of the BFT 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 by adding transactions or removing existing transactions from the list.

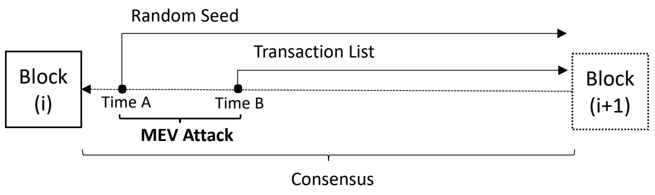
We assume that the underlying BFT consensus works as expected. In the system, a node that is not faulty throughout the consensus is considered to be honest and executes the BFT protocol as specified. The honest nodes are assumed to have stable network connections, adequate computing power, and sufficient storage space to process blockchain transactions. An honest node will not collude with a faulty one. It have 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 RNG for BFT-based blockchains. RNGs for non-BFT settings are out of the scope of this work. For the security of smart contracts, we only consider the identified RAU attack and the MEV attacks that pose threats to the runtime RNG. Other forms of MEV attacks, such as front-running, back-running, and sandwich attacks on decentralized exchange platforms [40], are not covered in this work.

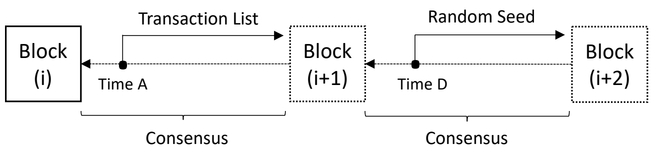
3.2. Problem Statement and Security Goals

3.2.1. Research problem. Our research problem is how to securely generate random values to smart contracts in the BFT-based blockchain at runtime. A runtime RNG is expected to meet the following requirements: *(i)* No adversary should be able to interrupt the RNG operations and predict the random number in advance. It also cannot interfere with the transaction execution according to the generated random number. *(ii)* Smart contracts invoking transactions can request multiple random values during execution. *(iii)* The runtime RNG does not undermine the security and efficiency of the BFT-based blockchain. Namely, runtime RNG should not introduce extra consensus steps to the existing ones and the adversary cannot take advantage of the runtime RNG to break the fairness of smart contract.

MEV attacks for runtime RNG. MEV attacks enable an adversary to monitor transactions in the blockchain network and manipulate transaction orders for its own benefit. Most of the BFT-based blockchains are vulnerable to MEV attacks with the exception of some confidential ledgers [41], [42]. As illustrated in Figure 2a, once the random value is known during the consensus, anyone, including the leader, can construct new transactions to exploit the smart contract with this random number. That is why existing RNGs use the callback mechanism to generate random values in future



(a) Demonstration of how MEV attacks happen, where time A < time B.



(b) The workflow of callback based RNG, where $T_{\texttt{request}}$ is recorded in block i+1 and the $T_{\texttt{callback}}$ is recorded in block $\geq$ i+2.

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

blocks, as shown in Figure 2b. We design BCRand as a runtime RNG that do not require callback transactions for MEV mitigation.

RAU Attack. Blockchain transactions are atomic [43]. In other words, a transaction is considered 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 attacker could craft an attack contract to call the victim contract and check the result against its expectations. If the result is not desired, the attack contract may revert the transaction intentionally. We call this the revert-after-use attack. The workflow of the RAU attack is depicted in Figure 3. We use an example of a "blind box" contract to demonstrate the RAU attack. 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 attack 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 the token; otherwise, $\mathcal{A}$ reverts the execution and get the payment back. The adversary can repeat the RAU attack until the result is desired.

3.2.2. Security Goals. Here we summarize the security goals of BCRand as below.

Goal 1. *Anti-MEV for RNGs.* An anti-MEV protocol ensures fair ordering by prioritizing transactions in a first-in-first-out fashion [44]. 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

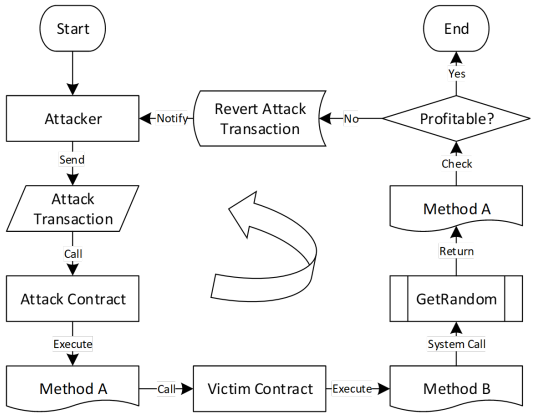


Figure 3: The RAU attack against runtime RNGs. Adversary repeatedly constructs malicious transactions to call the victim contract from an attack contract and checks the execution result of the victim contract against its expectation.

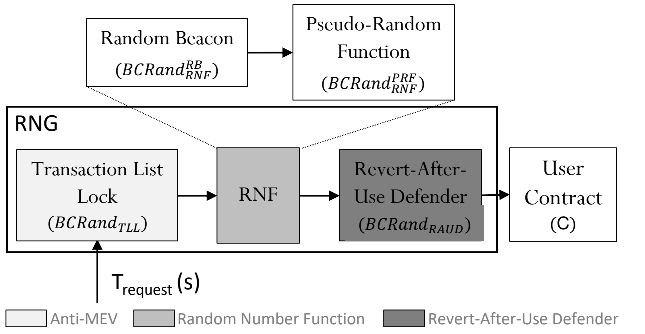


Figure 4: High level architecture of BCRAND. It contains three phases, the anti-MEV phase (denoted as BCRAND$_{TLL}$), two modules in the second phase, BCRAND$^{RB}_{RNF}$ and BCRAND$^{PRF}_{RNF}$, to generate fair random numbers, and the last phase RAU defender (denoted as BCRAND$_{\mathtt{RAUD}}$).

network at time i, T_j be a transaction that an adversary node broadcasts at time j. $j \geq i + \Delta$. An anti-MEV protocol ensures that T_i is always executed earlier than T_j. However, existing fair ordering anti-MEV implementations [44] are complicated and unnecessary for RNG protocol, because the adversary $\mathcal{A}$ cannot perform MEV attacks against RNG 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 specification for RNGs.

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

Goal 2. *Secure Random Number Function (RNF).* BCRAND should generate random values that are bias-resistant and unpredictable. Further, the seed for each block should be verifiably unique. Specifically, we want to realize the following defined secure random number functions.

Definition 3.2 (Secure Random Number Function). Let O be the seed output for some block b, $\mathtt{C}$ be a smart contract that requests m random numbers, T and T' be two transactions in the *proposal* $\mathcal{L}_{\langle b,v,p\rangle}$ for block b at view v from leader p that invoke $\mathtt{C}$, and $R = \{r_1, r_2 ... r_m\}$ and $R' = \{r'_1, r'_2 ... r'_m\}$ be the results of C executing T and T' respectively. A random number function is secure if it satisfies:

- Bias-resistance. No adversary $\mathcal{A}$ can bias any bit of O but with negligible probability $\mathsf{negl}(\kappa)$, where $\mathsf{negl}(\cdot)$ is a negligible function and κ is a security parameter.
- Unpredictability. For any two values $\{r_i, r_j\}$ from $R \cup R'$, the probability of r_i and r_j at some position k are equal is $1/2 \pm \mathsf{negl}(\kappa)$.
- Deterministic. Any time C executing T and T' from $\mathcal{L}_{\langle b,v,p\rangle}$ always yields R and R' correspondingly.
- Verifiably Unique. There is only one legitimate O for each block $b > 1$, and its authenticity can be publicly verified after it is generated.
- Liveness. For every block $b > 1$, the protocol generates a seed O.

Goal 3. *Irrevocable Execution.* To defend against the RAU attack, transactions that trigger $\mathtt{C}$ 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. We say that C's execution is irrevocable if T is incapable of checking the value of res and deliberately reverting when res is not expected.

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.

3.3. Our Approach

Overview. BCRAND contains three major components: a) the anti-MEV attack BCRAND$_{TLL}$, which locks the transaction list during the consensus; b) the random number function (RNF), including BCRAND$^{RB}_{RNF}$ and BCRAND$^{PRF}_{RNF}$; and c) the RAU defender BCRAND$_{\mathtt{RAUD}}$ that prevents adversaries from reverting $T_{\mathtt{request}}$. $\mathtt{C}$ is the smart contract that uses runtime random numbers.

(a) Transaction List Lock BCRAND$_{TLL}$. The BCRAND$_{TLL}$ is a distributed lock system that prevents anyone from modifying the transaction list.

(b) Random Number Function BCRAND$_{RNF}$. BCRAND$^{RB}_{RNF}$ generates a random seed during each consensus round. The seed is expected to be bias-resistance, unpredictable, and

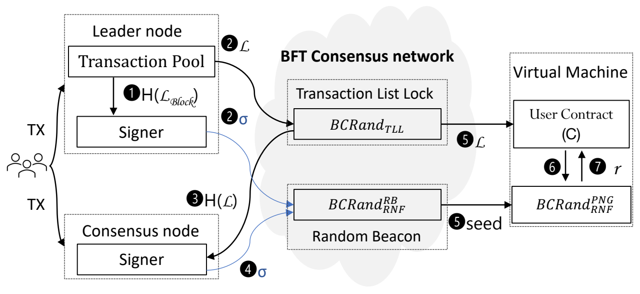


Figure 5: The data flow of BCRAND. $\mathcal{L}$ represents the *proposal* from the leader. σ is the BLS signature. r is the generated random number.

verifiably-unique so that no one can predict or manipulate it. BCRAND$^{RB}_{RNF}$ is intended to aggregate multiple input signatures into a single one. Since BCRAND is based on the BFT consensus, BCRAND$^{RB}_{RNF}$ employs threshold BLS [27], [39] to accommodate faulty input. This is the interface that provides random number calls for a smart contact during runtime. For each random number call made by the smart contract, BCRAND$^{PRF}_{RNF}$ returns a random number. A smart contract may call the interface as many times as it wants, and the generated random number is bound to the transaction that triggers the smart contract, ensuring that no transaction receives the same random numbers.

(c) RAU Defender BCRAND$_{RAUD}$. BCRAND$_{\mathrm{RAUD}}$ is a smart contract programming library that includes built-in functions to prevent transactions from performing RAU attacks on contracts.

Workflow. We sketch the workflow of a typical random number reqeust and generation depicted in Figure 5. The detailed formal protocol is presented in Section 4.1. For simplicity, we assume that honest nodes are in the consensus round b at view v with leader p. We also assume that there is no view change.

Prior to the start of the BCRAND, the leader monitors network transactions and generates a *proposal* $\mathcal{L}$. In ❶, the leader obtains a BLS signature σ. In ❷, the leader broadcasts both $\mathcal{L}$ and σ to the consensus network, indicating the beginning of a new round of consensus. Once a consensus node receives the message from the leader in ❸, it verifies both the transactions in $\mathcal{L}$ and the structure of $\mathcal{L}$ in order to lock the transaction orders. After $\mathcal{L}$ is locked, any subsequent changes to $\mathcal{L}$ are not permitted. In ❹, each consensus node $p_i \in \mathcal{P}$ creates a BLS signature and broadcasts it to the consensus network. Given all σs from $p_i \in \mathcal{P}$, the consensus network produces a random number seed for the new block. In ❺, the seed is transferred to the BCRAND$^{PRF}_{RNF}$ and $\mathcal{L}$ is loaded into the virtual machine. When a smart contract C calls BCRAND$^{PRF}_{RNF}$ for a random number in ❻, BCRAND$^{PRF}_{RNF}$ returns a random number r and promptly provides it to the smart contract in ❼.

4. Detail of BCRAND Protocol

In this section, we present BCRAND protocol while tolerating $f \leq t < n/3$ Byzantine faults.

Setup. Run the DKG protocol to generate a BLS key pair for each consensus node.

BCRAND$_{TLL}$. MEV attacks are the primary reason why other random number generation algorithms use a callback mechanism to feed random values to smart contracts. To defend against MEV attacks without callback, BCRAND$_{TLL}$ must ensure that the *proposal* $\mathcal{L}_{\langle b,v,p\rangle}$ from the leader p for block b at view v is immutable during and after the random number (seed) generation. As a result, when the leader proposes $\mathcal{L}_{\langle b,v,p\rangle}$ in BCRAND, BCRAND$_{TLL}$ obtains the hash value of the internal transaction list using a Merkle tree [29], and then signs it using BLS: $\sigma_p \leftarrow \mathit{BLS.Sign}(sk_p, \mathcal{L}_{\langle b,v,p\rangle})$. When a non-leader node p_i receives σ_p, it verifies the signature by $\mathit{BLS.Verify}(pk_p, \mathcal{L}_{\langle b,v,p\rangle}, \sigma_p)$. p_i creates a BLS signature σ_{p_i} if the verification succeeds.

BCRAND$_{RNF}$. The random number function consists of two steps: BCRAND$^{RB}_{RNF}$ and BCRAND$^{PRF}_{RNF}$. BCRAND$^{RB}_{RNF}$ is a random beacon protocol that generates a random seed, denoted as O, for every block b. As shown in Algorithm 1, with a (k, n)-threshold BLS signature BLS, BCRAND$^{RB}_{RNF}$ aggregates the input signatures σ_i into a single one σ' when the number of signatures is equal to or greater than k. Next, a random seed can be calculated as $O \leftarrow \mathsf{H}(\sigma')$.

Algorithm 1: Random Beacon BCRAND$^{RB}_{RNF}$

```
Input: $\{\sigma_i | \sigma_i$ is the $BLS$ signature of $p_i\}$
Output: Random Seed ($O$)
Data: $t$
1 if $| \{\sigma_i\} | < k$ then
2     return
3 $\sigma' \leftarrow BLS.Aggregate(\{\sigma_i\})$
4 Assign $\mathsf{H}(\sigma')$ to $O$
5 return $O$
```

Since one transaction may trigger multiple requests for random numbers, BCRAND$^{PRF}_{RNF}$ is used to generate the requested random numbers based on the seed O from BCRAND$^{RB}_{RNF}$. BCRAND$^{PRF}_{RNF}$ has an internal variable *salt* which is a singleton instance that can hold its assigned value across the execution of all transactions in block b. When a transaction T invokes a random number call for the first time, BCRAND$^{PRF}_{RNF}$ saves its hash value to h_T and initializes *salt*, such that no transaction could generate the identical random numbers. Further random number calls from T are fulfilled with the hash value of the *salt*. More detail of BCRAND$^{PRF}_{RNF}$ is illustrated in Algorithm 2.

BCRAND$_{\mathrm{RAUD}}$. To defend against the revert-after-use attack, the smart contract C must restrict the invoking transaction T from doing the following actions: (a) calling C from a *delegate contact*; 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 BCRAND$_{\mathrm{RAUD}}$. When C is invoked by

Algorithm 2: Pseudo-Random Function BCRand$^{PRF}_{RNF}$

```
Input: Random Seed ($O$), Transaction ($T$)
Output: Random Number ($r$)
Data: Transaction hash ($h_T$), Global variable ($salt$)
1 if $h_T$ is null or $h_T \neq T.hash$ then
2     $h_T \leftarrow T.hash$  // Cache the transaction hash
3     $salt \leftarrow \mathsf{H}(h_T \oplus O)$
4 else
5     $salt \leftarrow \mathsf{H}(salt)$
6 Assign $salt$ to $r$
7 return $r$
```

transaction T, $\texttt{BCRand}_{\texttt{RAUD}}$ first verifies whether C is the first smart contract invoked by T. If not, abort the execution. Then $\texttt{BCRand}_{\texttt{RAUD}}$ 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, $\texttt{BCRand}_{\texttt{RAUD}}$ requires a developer-supplied parameter specifying the length of the script.

4.1. BCRand Protocol

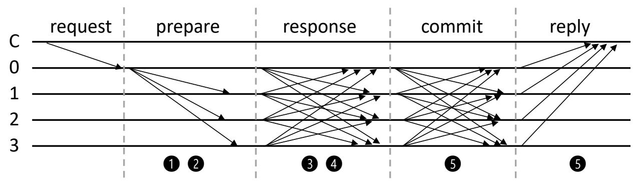


Figure 6: BCRand workflow in the view of BFT consensus. Steps correspond to Figure 5.

BCRand focuses on producing random numbers for smart contracts at runtime and leverages BFT as the consensus layer for block generation. We incorporate BCRand into the classic BFT consensus without adding additional steps, as shown in Figure 6. Figure 7 details BCRand based on BFT.

(1) **Prepare. Prepare** starts at the beginning of each view v in BFT. In this step, leader p constructs a transaction list and generates a *proposal* $\mathcal{L}_{\langle b,v,p\rangle}$ with the list for block b. Transactions in the *proposal* are organized with a Merkle Tree, such that the order of transactions can be verified. Then p runs BCRand$_{TLL}$ and passes $\mathcal{L}_{\langle b,v,p\rangle}$ as a parameter. BCRand$_{TLL}$ yields a BLS signature σ_p for $\mathcal{L}_{\langle b,v,p\rangle}$ and broadcasts σ_p to the consensus network along with the *proposal*.

(2) **Response.** After $p_i \in \mathcal{P}$ receives the *proposal* $\mathcal{L}_{\langle b,v,p\rangle}$ and the BLS signature σ_p from the leader p, it verifies the validity of the proposal and authenticity of the signature by calling BCRand$_{TLL}$. If verification passes, BCRand$_{TLL}$ returns a BLS signature of p_i for $\mathcal{L}_{\langle b,v,p\rangle}$: $\sigma_{\langle b,v,p_i\rangle} \leftarrow$ BCRand$_{TLL}(\mathcal{L}_{\langle b,v,p\rangle})$.

(3) **Commit.** In the **Commit** phase of BFT, p_i collects and verifies the response messages from other nodes. If p_i obtains over $2t$ valid responses, it issues a commit message indicating that p_i is ready to construct block b from $\mathcal{L}_{\langle b,v,p\rangle}$. In BCRand, we add BCRand$^{RB}_{RNF}$ to this phase. When p_i receives the response messages, it also receives the corresponding BLS signatures σ. Before p_i sends out the commit message, it calls BCRand$^{RB}_{RNF}$ to produce the block seed O and then broadcasts the seed along with the commit message to the network.

(4) **Validate.** Recall that C is a smart contract that implements $\texttt{BCRand}_{\texttt{RAUD}}$. When p_i sends transaction T to call smart contract C, $\texttt{BCRand}_{\texttt{RAUD}}$ will verify if the calling script of the transaction has a valid format.

(5) **GetRandom().** `GetRandom()` is the interface through which smart contact C requests random values. For a transaction T that triggers C, BCRand$^{PRF}_{RNF}$ is called to return the resulting random number r to the smart contract. Different r can be returned by calling `GetRandom()` multiple times during the execution of T.

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 RNGs (**Goal 1**), (2) secure random number function (**Goal 2**), and (3) irrevocable execution (**Goal 3**). For simplicity, we use BLS to represent a (k, n)-threshold BLS signature scheme.

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

Theorem 5.1 (Security of BCRand$_{TLL}$). *Assuming that the hash function* H *is collision resistant and that the digital signature scheme used for consensus messages and BLS signature are existentially unforgeable under chosen message attacks (EUF-CMA), and* BFT *is* (f, t, n)*-secure, then* BCRand$_{TLL}$ *achieves anti-MEV security in accordance with Definition 3.1.*

Proof. For a view $v \geq 0$ of consensus round $b > 1$, we show that BCRand$_{TLL}$ is anti-MEV by proving that *(1)* no node $p_i \in \mathcal{P}$ knows the random seed O before the transaction list is locked; and *(2)* no one could change the transaction list. Assuming that the leader p is controlled by adversary $\mathcal{A}$, then the $\mathcal{A}$ can manipulate the transaction list. $\mathcal{A}$ creates a *proposal* $\mathcal{L}$. If $\mathcal{A}$ wishes to know the random seed before sending *proposal* $\mathcal{L}$, $\mathcal{A}$ has to obtain at least k response messages $(response, \sigma_{\langle b,v,p_i\rangle},)_{p_i}$. Since BFT is (f, t, n)-secure, adversary $\mathcal{A}$ can control at most t nodes and $t < k$. However, if adversary $\mathcal{A}$ can forge another $k - t$ response messages, it can compute the random seed O. Then the problem can be reduced to the EUF-CMA property of consensus message signature scheme and BLS signature scheme. Without accessing to the corresponding secret keys, $\mathcal{A}$ cannot forge those signatures. As a result, adversary $\mathcal{A}$ can never know the random seed before BCRand$_{TLL}$ locks $\mathcal{L}$. The transactions in the *proposal* are ordered by the Merkle hash tree. Adversary $\mathcal{A}$ cannot change the list without a different root hash. This is guaranteed by collision resistant hash functions. Because no one can modify

Run DKG protocol. Then every node generates a key pair $(BLS.sk, BLS.pk) \leftarrow DKG.KGen(1^\lambda)$, and all nodes agree on each other's public keys. Let BLS be the (k, n)-threshold BLS scheme used, BFT be the (t, n, κ)-secure BFT consensus and $0 \le f \le t < k \le 2t+1$, $b > 1$ be the current consensus round and $p_{\langle b,v \rangle}$ be the leader of round b at view $v \ge 0$, $Q_{\langle b,v \rangle}$ be the set of BLS signatures for view v of round b, O_b be the seed 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 generates a *proposal* $\mathcal{L}_{\langle b,v,p \rangle}$ and gets the BLS signature: $\sigma_{\langle b,v,p \rangle} \leftarrow$ BCRAND$_{TLL}(\mathcal{L}_{\langle b,v,p \rangle})$, then broadcasts $(prepare, \mathcal{L}_{\langle b,v,p \rangle}, \sigma_{\langle b,v,p \rangle})_p$.

(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 $\mathcal{L}_{\langle b,v,p \rangle}$ and verifies the internal transactions, and gets the proposal's hash value $hash \leftarrow \mathsf{H}(\mathcal{L}_{\langle b,v,p \rangle})$;
- verifies $\sigma_{\langle b,v,p \rangle}$ with $hash$, and then calculates the BLS signature $\sigma_{\langle b,v,p_i \rangle} \leftarrow$ BCRAND$_{TLL}(\mathcal{L}_{\langle b,v,p \rangle})$;
- broadcasts *response* message $(response, \sigma_{\langle b,v,p_i \rangle})_{p_i}$.

(3) **Commit:** If receives *response* message from some $p_j \in \mathcal{P}$, `Add`$(Q_{\langle b,v \rangle}, \sigma_{\langle b,v,p_j \rangle})$. With `Len`$(Q_{\langle b,v \rangle}) > 2t$, get random seed $O_{\langle b,v \rangle} \leftarrow$ BCRAND$^{RB}_{RNF}(Q_{\langle b,v \rangle})$. Broadcast commit message $(commit, \mathcal{L}_{\langle b,v,p \rangle}, O_{\langle b,v \rangle})_{p_i}$.

(4) **Validate:** For a transaction T from block b that triggers $\mathcal{C}$, call BCRAND$_{\text{RAUD}}(T)$ from $\mathcal{C}$.

(5) **GetRandom():** For random seed O and each transaction T that calls for random from block b , get random number $r \leftarrow$BCRAND$^{PRF}_{RNF}(T, O)$, and output r.

Figure 7: BCRAND: secure runtime random number generator protocol description.

the transaction list during or after the random seed generation, BCRAND$_{TLL}$ accomplishes anti-MEV for runtime RNGs. □

BCRAND realizes the defined secure random number function in **Goal 2**.

Theorem 5.2 (Security of BCRAND$_{RNF}$)**.** *Assuming BLS signature and the hash function H in* BCRAND$^{PRF}_{RNF}$ *are secure, and the BFT is (f, t, n)-secure, then* BCRAND$_{RNF}$ *is a secure RNF under Definition 3.2.*

Proof. We briefly prove that BCRAND$_{RNF}$ meets all the security requirements in Definition 3.2.

Bias-resistance. The output of BCRAND$^{RB}_{RNF}$ in any view v of consensus round b is $O_{\langle b,v \rangle}$, which is the hash of the aggregated BLS signature σ. Assume that a PPT-adversary $\mathcal{A}$ wants to bias some bits of $O_{\langle b,v \rangle}$, and controls the leader p at view v. As BFT is secure, $\mathcal{A}$ can control at most t nodes, which is smaller than the threshold k of BLS. As a result, $\mathcal{A}$ cannot know O before the transaction list is locked. p has to construct a *proposal* without knowing O. Therefore, $\mathcal{A}$ cannot fix any bit of O. With a bias-resistant O, random values are generated through BCRAND$^{PRF}_{RNF}$ using hash function H during the execution of a smart contract. This process cannot be interfered with unless BFT is compromised. Thus, adversary $\mathcal{A}$ may fix a bit of the random value generated by BCRAND$_{RNF}$ with probability no more than $\mathsf{negl}(\kappa)$.

Unpredictability. The random seed O is derived by hashing the BLS aggregated signature σ. If a PPT-adversary $\mathcal{A}$ could predict a bit in O, it would imply that the adversary $\mathcal{A}$ could predict σ before the aggregation, which is not possible due to the BLS signature. Therefore, BCRAND$^{RB}_{RNF}$ is unpredictable. The following random values are generated by further hasing the seed. With an unpredictable random seed from BCRAND$^{RB}_{RNF}$, these random values are also unpredictable, i.e. BCRAND$_{RNF}$ achieves unpredictability.

Deterministic. This is because the hash computation is a deterministic process, i.e. the same inputs always generates the same hash output. Given a fixed seed, the same number will be produced through the deterministic hashing.

Verifiably Unique. This is guaranteed by the threshold BLS signature that only one aggregated signature σ will be generated and publicly verifiable by all nodes.

Liveness. For a consensus round $b > 1$, the view v of b starts from 0. Since the threshold of BLS is k, the BFT is (f, t, n)-secure, as long as $0 \le f \le t < k \le 2t+1$, for any PPT-adversary $\mathcal{A}$, $\mathcal{A}$ cannot block BCRAND$^{RB}_{RNF}$ from aggregating signatures and generating a new random seed for block b. If at any moment BFT consensus triggers view change, BCRAND resets and starts to process in view $v+1$. Therefore, as long as BFT consensus is secure, BCRAND$^{RB}_{RNF}$ always generates a random seed for every block $b > 1$. □

BCRAND accomplishes **Goal 3** by implementing BCRAND$_{\text{RAUD}}$ to defend against RAU attacks.

Theorem 5.3 (Security of BCRAND$_{\text{RAUD}}$)**.** BCRAND$_{RAUD}$ *achieves irrevocable execution under Definition 3.3.*

Proof. For a smart contract `C` that triggers runtime random number calls and has BCRand$_{\text{RAUD}}$ implemented. A PPT-adversary $\mathcal{A}$ can perform RAU attack on `C` in two ways: (1). Adversary $\mathcal{A}$ deploys an attack contact `C'` with the intent of launching a RAU attack 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 BCRand$_{\text{RAUD}}$ 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 RAU attack. Therefore BCRand$_{\text{RAUD}}$ achieves irrevocable execution. □

6. Implementation

We implemented a BCRand prototype on the neo blockchain platform or neo. Neo [28] is a well-known BFT-based blockchain project. It integrates non-fungible tokens (NFT), distributed storage, Oracles, and built-in virtual machines. It also supports C# for developing developer-friendly smart contracts. The consensus algorithm of neo is the delegated BFT (dBFT), an optimized version of practical BFT [34]. 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.

6.1. 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 RAU attacks, 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.

6.2. Applications

We described the applications that rely on random numbers in order to show the advantage of BCRand.

Fair NFT Distribution. Loot [5] is one of the most successful NFTs deployed on Ethereum, with $8,000$ OG tokens originally accessible by public for claim. 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 contents 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 Request. Neoverse is a "blind box" smart contract based on the neo blockchain 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 truly 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 the user 2 GAS; otherwise, the contract retains the user's stake.

Comparison. To compare BCRand with callback-based RNG solutions, we implemented a comparison application that contains three functions: `Runtime()` which calls `GetRandom()` directly to work as BCRand, and two other functions `Callback()` and `Fulfill()` to simulate the workflow of callback-based RNGs. `Callback()` issues a random number request and `Fulfill()` accepts the random number from a simulated random Oracle. 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 crypto function provided by CryptoLib [45], a native contract, "CryptoLib.Sha256" to expand the received random number. For simplicity, we denote `Callback()` and `Fulfill()` together as BCRand$_{callback}$.

7. Evaluation

In this section, we show the performance of BCRand and demonstrate the advantage of BCRand by comparing BCRand with other RNGs. We implemented BCRand with a quad-core 3.6 GHz Intel(R) E3-1275 v5 CPU [46] 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.

7.1. Lines of Code

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

Table 1: Sample neo smart contracts developed with BCRAND. 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

7.2. Application Transaction Cost

Table 2 summarizes the transaction costs related with 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	0.20661505
Neoverse:: UnBoxing	0.00119552	0.07280720
Neoverse:: BulkUnBoxing	0.00125752	0.36020228
RPC:: OnNEP17Payment	0.00616260	0.06555925

7.3. Transaction Fee Cost

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

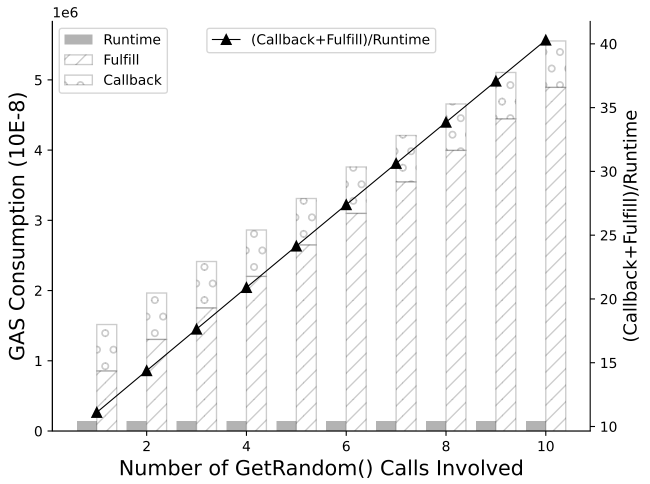


Figure 8: The GAS consumption when calling `GetRandom()`. The left y axis is the total GAS consumption, while the right y axis is the GAS increment when more `GetRandom()` calls involved.

implemented in C# on the neo platform according to Chainlink[1]. The evaluation result is shown in Figure 8. We can see that the average cost to request a random number with BCRAND is 0.00000144 GAS and it costs 0.00448629 GAS on average to get a random number with BCRAND$_{callback}$, which is $3,115$ times higher than BCRAND (i.e. BCRAND saves 99.967% fees). If we consider that one transaction only requests one random number, the fee to request a random number with BCRAND is 0.00104907 GAS, which is only 20% of the cost of issuing a callback transaction for BCRAND$_{\texttt{callback}}$.

7.4. Blockchain Overhead

To evaluate the impact of RNGs on the blockchain ledger, we define the size of data other than $T_{\texttt{request}}$ has to be written onto the blockchain to complete a random number request as blockchain overhead. Denote the transaction size as l, n the number of $T_{\texttt{request}}$ transactions, h the average number of $T_{\texttt{request}}$ in each block. While processing n $T_{\texttt{request}}$, the blockchain overhead of BCRAND$_{callback}$ 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 is $32n/h$, because the random number seed in every block header is 32 bytes. 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 [25]). As shown in Figure 9, if 0.04% of total transactions are $T_{\texttt{request}}$, i.e., one $T_{\texttt{request}}$ for every 5 blocks, then BCRAND saves about 17% blockchain overhead versus BCRAND$_{\texttt{callback}}$

1. Chainlink [47] has the most influential RNG solution, which has processed about 7 million random number requests for multiple chain projects.

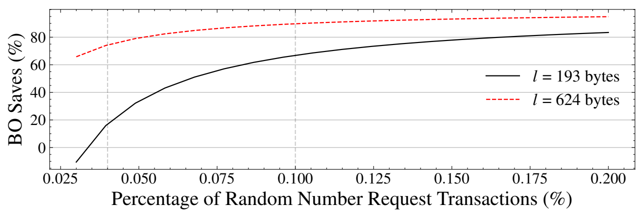


Figure 9: The blockchain overhead that BCRAND saves compared to the callback-based solution. BO stands for Blockchain Overhead.

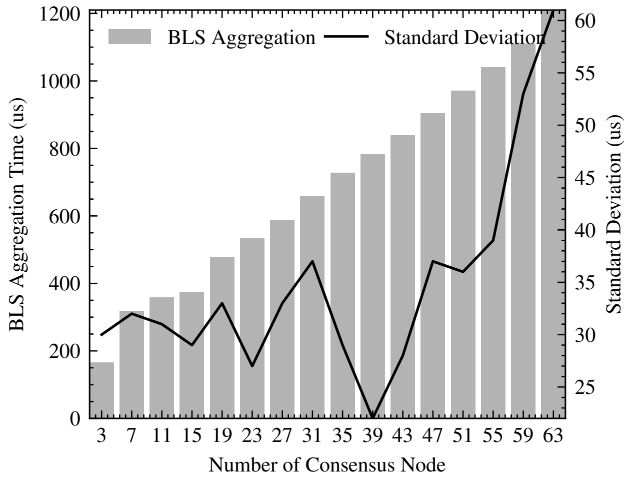


Figure 10: The BLS signature aggregation time.

for $l = 193$ bytes and 72% for $l = 624$ bytes; if 0.1% transactions are $T_{\texttt{request}}$, BCRAND saves at least 67% blockchain overhead for $l = 193$ bytes and 89% for $l = 624$ bytes.

7.5. Evaluation on BLS signature

We evaluate the overhead of BLS signature aggregation. Since we use BLS to generate the random seed, we evaluate the aggregation time of σ. Figure 10 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 $318\mu s$; when the number of consensus nodes is increased to 23 (comparable to 21 nodes in EOS), the time cost is about $533\mu s$. Adding BCRAND 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 57ms. The evaluation result is shown in Figure 11 with 2,000 DKG setups. Most of them can finish in about $1s$. But there are cases where the setup takes longer (e.g. 13% needs $2s$). This is mainly affected by the communication cost of

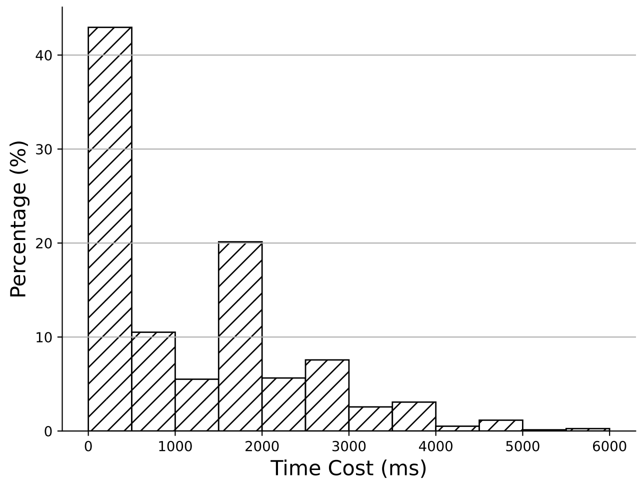


Figure 11: BLS setup time cost in a testnet with 7 consensus nodes.

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.

7.6. RNG Solutions Comparison

BCRAND provides an efficient runtime random number generator for BFT-based blockchain smart contracts. To demonstrate the advantages of BCRAND, we compare BCRAND with a host of current random number projects. Table 3 shows the comparison result in terms of the following categories: Platform Consensus (the consensus of the platform that the RNG is built on), Method (s) (the technique used to design the RNG protocol), Resistance (the number of Byzantine faults tolerated in the system), # random values (number of random numbers RNG can generate), Runtime (the support for runtime random number generation). In the table, n denotes the size of the consensus nodes, t is the maximum number of Byzantine nodes permitted in the system, and O denotes the random seed.

We compare the features among random beacons (Drand [14], HERB [16], RandChain [18], RandHerd [19], RandHound [19], BRandRiper [21]), RNG-enabled blockchain projects (Dfinity [15], Secret [48], Elrond [49], Klaytn [50], Harmoney [51]), and random oracles (Chainlink [47], Automata [52]).

7.6.1. Platform Consensus. Since we do not allow one consensus node to generate random numbers, most of the RNG projects, including BCRAND, are designed for or based on BFT consensus, as shown in Table 3. 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, as an oracle, is platform independent and provides randomness for both

Table 3: Comparison of random number generator protocols for blockchain

Protocol	**Platform Consensus**	**Method(s)**	**Resistance (t)**	**# random values (r)**	**Runtime**
Drand [14]	PABFT	Threshold SecretBLS	$t < n/2$	$\mathcal{O}(O)$	✗
HERB [16]	∅	Threshold ElGamal	$t < n/3$	$\mathcal{O}(O)$	✗
RandChain [18]	Sequential PoW	PoW	$t < n/3$	$\mathcal{O}(O)$	✗
RandHerd [19]	BFT	Threshold Schnorr	$t < n/3$	$\mathcal{O}(O)$	✗
RandHound [19]	BFT	Client based, PVSS	$t < n/3$	$\mathcal{O}(O)$	✗
BRandRiper [21]	BFT	VSS, q-SDH	$t < n/2$	$\mathcal{O}(O)$	✗
Dfinity [15]	BFT	Threshold BLS	$t < n/2$	∞	✗
Secret [48]	DPoS	Scrt-RNG,TEE	$t < n/2$	∞	✗
Elrond [49]	Secure PoS	BLS,onchain data	$t < n/3$	∞	✗
Klaytn [50]	Istanbul BFT	VRF	$t < n/3$	$\mathcal{O}(O)$	✗
Harmoney [51]	Fast BFT	VRF,VDF	$t < n/3$	$\mathcal{O}(O)$	✗
⋆Chainlink [47]	∅	VRF, TEE	$t < n/2$	$\mathcal{O}(O)$	✗
⋆Automata [52]	∅	VRF, TEE	$t < n/2$	∞	✓
BCRAND$_{callback}$	BFT	∅	$t < n/3$	$\mathcal{O}(O)$	✗
BCRAND	BFT	BCRAND$_{\{TLL,\ RNF,\ RAUD\}}$	$t < n/3^{\dagger}$	∞	✓[§]

κ is the security parameter that specifies maximum of sizes of signatures, hashes, and other components. **Resistance** refers to the system's tolerance for Byzantine faults. ⋆ is the off-chain RNG Oracle. $^{\dagger}t < n/2$ for synchronous BFT, $t < n/3$ for asynchronous BFT consensus. O is the random seed generated by random beacon. ∞ means number of random values is upper bounded by consensus. [§]BCRAND is the first runtime smart contract RNG solution on BFT-based blockchain.

BFT-based [25] and PoW [53] blockchain platforms. Automata [52] is a blockchain middleware, specifically built for Ethereum. As BCRAND is built on top of BFT, it can be easily ported to any BFT-based blockchain project.

7.6.2. Method. To avoid the faulty node from blocking the process of generating random numbers or exploiting the random numbers, HDrand, RandRiper, Dfinity, Elrond, and BCRAND use the threshold BLS signature scheme 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, and Automata, make use of the trusted execution environment (TEE) [54]–[56], 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 a RNG to operate on TEE-enabled devices.

7.6.3. Random Values. We compare the ability of RNGs to generate a large number of random values. 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 seeds O. While Secret and Elrond provide built-in pseudorandom functions that can extend O into numerous random values, the random value they can supply is limited by the consensus's maximum transaction cost and transaction size, which we represent as ∞. Because BCRAND uses BCRAND$^{PRF}_{RNF}$ to safely create random values from the random seed provided by BCRAND$^{RB}_{RNF}$, the number of random values can be generated by BCRAND is likewise constrained by the consensus.

7.6.4. Runtime. Due to MEV and RAU attacks, none of the RNGs 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 projects we compare. It is the only one that can deliver runtime random numbers to smart contracts. Automata obtains a runtime random number by requesting transactions from users and then creates random values with a verified random function (VRF) in the TEE. Following that, Automata creates EIP712 [57] transactions and sends them to an Automata smart contract deployed on Ethereum, where the Automata contract then invokes the user contract. As a result of its close association with Ethereum, Automata is now unavailable on BFT-based blockchains. In summary, BCRAND is currently the only safe runtime random number generator for BFT-based blockchains.

8. Related Work

Random Number Generators for Smart Contract. Existing works on random beacons adopt different cryptographic primitives and threat models [13], [14], [17], [18], [38], [58], [59]. Some projects [16], [60], [61] are built with homomorphic encryption, but homomorphic encryption introduces heavy overhead to the system. Hedera Hashgraph [62] is a fault-tolerant RNG algorithm for a public directed acyclic graph, which can be used as RNG service. There are also projects runs in a permissionless setting [47]–[49], [62]–[67]; unfortunately, they do not support runtime RNG, while BCRAND is a runtime RNG for smart contracts.

Hardware based Random Number Generators. The Verifiable Delay Function (VDF) [68] is used as a source of randomization in ETH 2.0 [69]. To make VDF secure, ETH2.0 has to develop Application-Specific Integrated Circuit (ASIC). Chainlink [47] enables smart contracts to interact with a TEE-based oracle to obtain randomness and cryptographic proof. Secret [48] uses crowd source and price entropy inside a TEE-based secret oracle to provide on-chain random service. The random beacon service of oasis [41] provides unbiased randomness on each epoch. It employs a commit-reveal strategy in conjunction with Publicly Verifiable Secret Sharing (PVSS). All these methods rely on specific hardware and introduces extra trust to the blockchains. BCRand, in contrast, is a hardware-independent RNG solution.

BFT-based Chain Projects. Several existing works employ BFT-family protocol as consensus. Celo [70] is built on the Istanbul BFT, which is carried out by a set of validator nodes. Kadena [71] originally uses the novel ScalableBFT consensus protocol. Solana's [72] Proof of Stake (PoS) system relies on a BFT mechanism called Tower Consensus. Multi-Level BFT is used by Theta [37] to reach a consensus that allows thousands of nodes to participate in the consensus process. Klaytn [50] uses an optimized version of Istanbul BFT, which has better scalability. The Helium [73] Consensus Protocol is based on a variant of the HoneyBadgerBFT (HBBFT) [74] protocol. The EOS [11] blockchain utilizes a Delegate Proof of Stake (DPoS) consensus mechanism, with a BFT model. Harmony proposes Fast BFT, which reduces communication costs by using BLS aggregate signature [51]. As BCRand is a BFT-based runtime RNG solution, it can potentially be deployed on these projects.

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

9. Discussion and Conclusion

Scalability. BCRand is designed for BFT-based smart contracts. BFT does not scale well due to its communication complexity $\mathcal{O}(n^2)$. Our method was tested in a real-world BFT blockchain setting, i.e., the neo platform with 7 consensus nodes and 21 committee nodes. There exists other BFT 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.

Hard fork. Since BCRand needs to put the random seed in the block header, blockchain projects that intend to adopt it might need to go a hard fork. However, this is not the only method to make the seed available to the virtual machine; in the future, we will consider additional options, such as keeping the random seed in a transaction to prevent a hard fork.

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 generators improve security, they suffer from many other issues, such as expensive on-chain costs and delayed processing time. In this work, we explore and design BCRand, a novel runtime RNG 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. Under the hood, we adopt the 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 RNGs. In addition, we also identify the revert-after-use attack 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] V. Buterin and ethereum.org, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[2] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[3] S. Mavis, "Official axie infinity whitepaper," https://whitepaper.axieinfinity.com/, Nov. 2021.

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

[5] L. Project, "Loot contract source code," https://etherscan.io/address/0xff9c1b15b16263c61d017ee9f65c50e4ae0113d7#code, 2021.

[6] S. Mavis, "Paid whitepaper," https://docsend.com/view/jdbdpza9d9nehnf2, Jan. 2021.

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

[8] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," *arXiv preprint arXiv:1904.05234*, 2019.

[9] A. Judmayer, N. Stifter, P. Schindler, and E. Weippl, "Estimating (miner) extractable value is hard, let's go shopping!" *Cryptology ePrint Archive*, 2021.

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

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

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

[13] J. Kelsey, L. T. Brandão, R. Peralta, and H. Booth, "A reference for randomness beacons: Format and protocol version 2," National Institute of Standards and Technology, Tech. Rep., 2019.

[14] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[15] I. Abraham, D. Malkhi, K. Nayak, and L. Ren, "Dfinity consensus, explored," *Cryptology ePrint Archive*, 2018.

[16] A. Cherniaeva, I. Shirobokov, and O. Shlomovits, "Homomorphic encryption random beacon," *Cryptology ePrint Archive*, 2019.

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

[18] R. Han, J. Yu, and H. Lin, "Randchain: Decentralised randomness beacon from sequential proof-of-work." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1033, 2020.

[19] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2017, pp. 444–460.

[20] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. Weippl, "Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness," 2021.

[21] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "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.

[22] chain.link, "https://chain.link/," https://chain.link/, accessed: 2020-08-18.

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

[24] Y. Qian, "Randao: Verifiable random number generation," 2017.

[25] BscScan, "Vrfcoordinator," https://bscscan.com/address/0x747973a5A2a4Ae1D3a8fDF5479f1514F65Db9C31#analytics, 2022.

[26] C. V. Helliar, L. Crawford, L. Rocca, C. Teodori, and M. Veneziani, "Permissionless and permissioned blockchain diffusion," *International Journal of Information Management*, vol. 54, p. 102136, 2020.

[27] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International conference on the theory and application of cryptology and information security*. Springer, 2001, pp. 514–532.

[28] neo.org, "Neo smart economy," https://neo.org/, 2022.

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

[30] P. McCorry, S. F. Shahandashti, and F. Hao, "A smart contract for boardroom voting with maximum voter privacy," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 357–375.

[31] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the internet of things," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1594–1605, 2018.

[32] ethereum, "Ethereum virtual machine (evm) awesome list," https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List, 2020-05-02.

[33] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.

[34] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[35] L. Ren, K. Nayak, I. Abraham, and S. Devadas, "Practical synchronous byzantine consensus," *arXiv preprint arXiv:1704.02397*, 2017.

[36] coinmarketcap.com, "Today's cryptocurrency prices by market cap," https://coinmarketcap.com/, Mar. 2022.

[37] T. NETWORK, "Theta network." [Online]. Available: https://docs.thetatoken.org/docs/whitepapers

[38] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 295–310.

[39] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. G. Gueta, and S. Devadas, "Towards scalable threshold cryptosystems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 877–893.

[40] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" *arXiv preprint arXiv:2101.05511*, 2021.

[41] O. P. Project, "The oasis blockchain platform," https://docsend.com/view/aq86q2pckrut2yvq, Jun. 2020.

[42] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.

[43] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[44] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *Annual International Cryptology Conference*. Springer, 2020, pp. 451–480.

[45] T. neo Project, "Neo native contract." [Online]. Available: https://github.com/neo-project/neo/blob/master/src/neo/SmartContract/Native/CryptoLib.cs

[46] 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.

[47] Chainlink, "Get a random number," https://docs.chain.link/docs/get-a-random-number/, 2021.

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

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

[50] K. Blockchain, "Consensus randomness," https://docs.klaytn.com/klaytn/design/consensus-mechanism, 2022.

[51] H. Blockchain, "Harmony randomness," https://docs.harmony.one/home/general/technology/randomness, 2022.

[52] A. Network, "Automata network," https://www.ata.network, 2021.

[53] Etherscan, "Vrfcoordinator," https://etherscan.io/exportData?type=address&a=0xf0d54349aDdcf704F77AE15b96510dEA15cb7952/, 2022.

[54] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ISCA*, 2013, p. 10.

[55] AMD, "AMD ESE/AMD SEV," https://github.com/AMDESE/AMDSEV, accessed: 2020-04-27.

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

[57] L. L. Remco Bloemen, "Eip-712: Ethereum typed structured data hashing and signing," https://eips.ethereum.org/EIPS/eip-712. [Online]. Available: https://eips.ethereum.org/EIPS/eip-712

[58] S. D. Simić, R. Šajina, N. Tanković, and D. Etinger, "A review on generating random numbers in decentralised environments," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2020, pp. 1668–1673.

[59] A. Bhat, A. Kate, K. Nayak, and N. Shrestha, "Optrand: Optimistically responsive distributed random beacons," *Cryptology ePrint Archive*, 2022.

[60] T. Nguyen-Van, T. Nguyen-Anh, T.-D. Le, M.-P. Nguyen-Ho, T. Nguyen-Van, N.-Q. Le, and K. Nguyen-An, "Scalable distributed random number generation based on homomorphic encryption," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 572–579.

[61] A. Cherniaeva, I. Shirobokov, and O. Shlomovits, "Homomorphic encryption random beacon," Cryptology ePrint Archive, Report 2019/1320, 2019, https://ia.cr/2019/1320.

[62] M. Krasnoselskii, G. Melnikov, and Y. Yanovich, "Distributed random number generator on hedera hashgraph," in *2020 the 3rd International Conference on Blockchain Technology and Applications*, 2020, pp. 7–11.

[63] Fetch.ai, “Launching our random number beacon on binance smart chain,” Oct 2020. [Online]. Available: https://medium.com/fetch-ai/launching-our-random-number-beacon-on-binance-smart-chain-8e3b7aa52be6

[64] “Random in xrp,” https://xrpl.org/random.html. [Online]. Available: https://xrpl.org/random.html

[65] Drand, “Drand/drand: a distributed randomness beacon daemon - go implementation.” [Online]. Available: https://github.com/drand/drand

[66] T. C. Devs, “Safe practice of tron solidity smart contracts: Implement random numbers in the contracts,” Mar 2020. [Online]. Available: https://coredevs.medium.com/safe-practice-of-tron-solidity-smart-contracts-implement-random-numbers-in-the-contracts-9c7ad8f6f9b0

[67] A. B. Ahead, “Poa 2.0: Vechain’s verifiable random function library in golang,” Mar 2020. [Online]. Available: https://abyteahead.medium.com/poa-2-0-vechains-verifiable-random-function-library-in-golang-5582268d073b

[68] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, “Verifiable delay functions,” in *Annual international cryptology conference*. Springer, 2018, pp. 757–788.

[69] ethereum.org, “Ethereum upgrades (formerly ’eth2’).” [Online]. Available: https://ethereum.org/en/upgrades/

[70] T. C. Project, “Celo randomness: Celo docs,” https://docs.celo.org/celo-codebase/protocol/identity/randomness. [Online]. Available: https://docs.celo.org/celo-codebase/protocol/identity/randomness

[71] T. K. Project, “Kadena whitepaper,” https://docs.kadena.io/basics/whitepapers/overview. [Online]. Available: https://docs.kadena.io/basics/whitepapers/overview

[72] 2022. [Online]. Available: https://docs.solana.com/introduction

[73] 2022. [Online]. Available: https://docs.helium.com/blockchain/consensus-protocol/

[74] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of bft protocols,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[75] NewEconoLab, “neo-ns,” https://github.com/NewEconoLab/neo-ns/blob/master/dapp_nns_register_sell/nns_register_sell.cs, 2018.

Appendix

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 number from Loot [5].

```
/// 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 [75]. It is vulnerable to MEV attacks.

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 [47]. This is costly to use.

3. Prototype of BCRAND$_{RAUD}$

```
[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)”.