BFTRAND: A Secure Runtime Random Number Provider for BFT Blockchain

***Abstract*—Random numbers are crucial to decentralized applications in the blockchain ecosystem. Random number providers (RNPs) rely on callback transactions to mitigate MEV attacks while generating and supplying random numbers to smart contracts. However, the cost of the callback function is not negligible, leading to high transaction fees, prolonged processing time, and bloated on-chain storage.**

In this paper, we propose BFTRAND, a novel runtime RNP protocol, which significantly outperforms the existing RNP services by removing the callback mechanism such that the random number can be fulfilled to a smart contract within the same consensus round where the request is initiated. We identify MEV vulnerabilities specifically for a runtime RNP and carefully design our protocol to ensure it is still MEV resistant by utilizing a distributed random beacon protocol. Meanwhile, we discover the malicious reversal, a misbehavior that allows the attacker to reverse the contract operations if the randomness-derived result is undesirable. We investigate the potential forms of the attack and its security implications to a runtime RNP and propose a defensive library for mitigation. The experimental results demonstrate that BFTRAND reduces the fee cost for random number requests by at least 89% while producing 76.4% less on-chain data compared to the callback-based solutions.

1. Introduction

Smart contracts [24], [96] are at the core of the decentralized applications (DApps) ecosystem. Many DApps, including DeFi and gaming, require access to randomness provided by public random number providers. For instance, decentralized games use randomness to determine the winner [66] or shuffle cards [41]; in non-fungible tokens (NFTs), randomness ensures uniqueness [34], rarity [80] and fair distribution [76]; and a legal toolkit relies on random jurors to resolve disputes [67].

The RNP protocol produces random values that are bias-resistant and unpredictable, i.e., no entity can interfere with the randomness generation and foresee the result. In the early development of blockchain RNPs, blockchain data was used as an entropy source. However, the blockchain data is public, the random numbers generated by on-chain RNPs can be computed in advance, leading to the Miner or Maximum Extractable Value (MEV) attacks [38], [55], [92], [94], which allows an attacker to infer the generated randomness before the contract execution and then construct transactions or modify the transactions list accordingly (Appendix B shows two on-chain RNPs). As a result, numerous lottery

(a) Callback RNP (b) Runtime RNP

Figure 1: Comparison between callback RNP and runtime RNP. $\mathcal{T}_{\texttt{request}}$ and $\mathcal{T}_{\texttt{callback}}$ denote transactions for requesting and providing random numbers, respectively. The solid line represents one consensus round. Callback RNP (a) requires two transactions in two separate consensus rounds to provide the requested random number, and runtime RNP (b) completes one request within one consensus round. RNG stands for random number generator.

contracts hosted on the EOS blockchain [42] were compromised, with the attackers profiting from their successful prediction of the winning number [91]. Currently, random number-related attacks are commonly recognized as one of the most prevalent blockchain attacks [92].

In the literature, the existing RNP schemes aim to counter MEV attacks [2], [16], [26], [33], [49], [58], [88], [89], [95] by explicitly leveraging a callback mechanism, as depicted in Figure 1a. In these methods, the user parameters are first recorded in the transaction $\mathcal{T}_{\texttt{request}}$ to initiate the random number request during a block interval before the randomness can be generated to anyone. This strategy prevents any parameter modification while enabling subsequent random number generation. Next, a second transaction $\mathcal{T}_{\texttt{callback}}$ is constructed in the following block time to provision random numbers, keeping the attacker from guessing in advance the random number because $\mathcal{T}_{\texttt{request}}$ precedes $\mathcal{T}_{\texttt{callback}}$ and occurs in different consensus rounds. In this case, the random number can only be created after $\mathcal{T}_{\texttt{request}}$ is stored on chain. Nevertheless, the present callback RNPs suffer from the following problems:

(i) High Transaction Fees: As callback RNPs require two transactions to complete one request for random number generation, the user initiating the request must pay transaction fees for both $\mathcal{T}_{\texttt{request}}$ and $\mathcal{T}_{\texttt{callback}}$. One might consider reducing the amortized cost by using one callback transaction $\mathcal{T}_{\texttt{callback}}$ to serve multiple request transactions $\mathcal{T}_{\texttt{request}}$. However, relying on one $\mathcal{T}_{\texttt{callback}}$ to invoke multiple intended contracts is problematic because we cannot ensure that a smart contract will be invoked without a trusted third party, which defies the decentralized nature of blockchain and must be avoided.

(ii) Latency: $\mathcal{T}_{\texttt{request}}$ and $\mathcal{T}_{\texttt{callback}}$ must exist in different blocks to store user parameters before creating random numbers, i.e., it takes at least two blocks to generate a random number for a smart contract [30], [46], [84], thus causing a delayed contract execution.

(iii) Storage Overhead: Although a requested random number could be just a few bytes, the required transactions are much larger. Except for the random number, a transaction also needs additional information, such as token's address, key hash, fee, and contract code, for correct functioning. For example, $\mathcal{T}_{\texttt{callback}}$ in Chainlink VRF [44][1] is 624 bytes versus a 32-byte random number. More transactions impose an additional storage burden to the blockchain.

In contrast, our runtime RNP protocol for smart contracts can *securely* fulfill random number requests within the *same* consensus round, i.e. only one transaction is required for random number provision, reducing the associated fees, time, and storage costs without sacrificing security. Figure 1b illustrates the runtime RNP concept.

BFTRAND - Runtime Random Number Provider (Runtime RNP). We introduce BFTRAND, a secure runtime RNP protocol for smart contracts on BFT-based blockchains. BFTRAND does not require a callback transaction $\mathcal{T}_{\texttt{callback}}$ [52], as illustrated in Figure 1b. The core concept of BFTRAND is to generate a beacon from a distributed random beacon (*DRB*) protocol during the block' consensus. Then the beacon is fed into a pseudorandom function (PRF) to generate the required random numbers for smart contracts. With BFTRAND, a random number request can be fulfilled at runtime within the same consensus round with the request transaction $\mathcal{T}_{\texttt{request}}$. As a result, users pay less for random number provision and have a much improved experience. In addition, the service providers also benefit from reduced on-chain costs.

Challenges. Due to the more direct request-response information flow, BFTRAND outperforms the current callback RNP protocols. However, it also poses the following security challenges.

MEV Attack. Without a callback function, a naïve runtime RNP design would be vulnerable to MEV attacks. Therefore, we identified four types of MEV attacks that compromise the runtime RNP security (see Section 3.3).

To mitigate such attacks, we rigorously define the *MEV resilience* considering the security requirements of *pseudorandomness*, *uniqueness*, and *liveness* that any runtime RNP must satisfy. *Pseudorandomness* captures the runtime RNP's ability to generate unpredictable and unbiased beacons, whereas *uniqueness* ensures that only one valid beacon is generated for a given input and *liveness* means one beacon will be successfully generated for each block, preventing an attacker from interfering with the underlying consensus by clogging beacon generation.

Malicious Reversal (MR). Blockchain transactions are atomic [9]. In other words, if a transaction fails, the related operations will be rolled back. We discover that the attacker can intentionally revert smart contract functions if the result is undesirable. We call this illegal behavior *malicious reversal* (Details in Section 3.4) The root cause of this attack is that the attacker can validate the state change before the malicious transaction completes its execution. We identified four types of malicious reversals in this paper: *Script MR*, the attacker attaches a malicious script to the transaction that will be executed after the user contract returns to check the state change; *Contract MR*, the attacker deploys an attacker contract, calls the user contract from the attacker contract, and then verifies the state after the user contract returns; *Fallback MR*, when the user contract transfers a token to the account of the attacker contract, the user contract triggers the fallback function of the attacker contract and enables it to validate the state in the fallback function; *Fee MR*, by knowing the fee cost associated with the execution paths in a contract, the attacker can set a maximum transaction fee such that only desired path will be taken.

We formally define the *MR resistance* property and propose a defense library to help developer mitigate malicious reversals (see Section 4.2 for details.).

Implementation and Evaluation. To evaluate the proposed protocol and compare it against existing callback RNP, we instantiate a concrete runtime RNP protocol BFTRAND by leveraging *DRB* scheme that has been used as the core building block for many RNP protocols [2], [17], [40]. We also implement $\text{BFTRAND}_{callback}$ as baseline to simulate the behavior of a callback RNP on neo [75], which was one of the top BFT blockchain project. We discover that BFTRAND saves 76.4% 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, **BFTRAND is the first secure runtime RNP protocol designed for smart contracts of blockchain projects running on BFT protocols**.

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

- We present BFTRAND, a novel runtime random number provider protocol for smart contracts on BFT-based blockchains.
- We formally analyze the security of BFTRAND, identify multiple MEV attacks and malicious reversals, and formally define and prove the basic properties of MEV Resilience and MR Resilience to mitigate MEV attacks and malicious reversals accordingly.
- We implement a BFTRAND prototype from BLS and evaluate its performance. Smart contracts on BFT-based blockchains that use BFTRAND can securely fulfill any random number request at runtime (if not bound by consensus).
- Four applications are evaluated to show the effectiveness and efficiency of BFTRAND. To prove its

1. Chainlink VRF is an off-chain random oracle. It is the most influential smart contract random number provider and is adopted by at least 613 projects [41]

superiority, we also compare BFTRAND to other RNPs.

2. Background

The notations used throughout this paper are reported in Table 1.

Symbol	Description
n	Number of system participants.
t	Maximum Byzantine nodes allowed s.t., $t = \lfloor n/3 \rfloor$.
k	Beacon aggregation threshold s.t., $t < k \leq 2t + 1$ (refer to Section 4.1.2 for detail).
f	Size of actual faulty nodes s.t., $f \leq t < k \leq 2t + 1$.
$\mathcal{N} = \{N_i\}$	Participants set $\mathcal{N}$ and participant N_i with $i \in [0, n)$.
$\mathcal{C}$	Set of corrupted participants.
b, v, l	Consensus round b, view v, and the leader index l.
$\mathcal{L}_{b,v}$	Block proposal in the consensus for view v of round b.
λ	Security parameter.
$\sigma_{b,v,i}$	Partial beacon from node N_i for view v of round b.
$\sigma^i_{b,v}$	Aggregated beacon from node N_i for view v of round b.
σ_b	Final beacon from the protocol for round b.
$\mathcal{T}$	User transaction.
$\mathcal{SC}$	Smart Contract.
$r_{\mathcal{T},i}$	i-th random number for transaction $\mathcal{T}$.

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 [68], [100]. An increasing number of smart contracts require randomness to perform their functions, such as in NFT contracts where randomness is required for fair distribution [80] and in GameFi contracts where randomness is used to determine winners and rarity for entities [53].

2.1.2. Byzantine Fault Tolerance (BFT). BFT is a family of consensus algorithms, such as *PBFT* [29] and its variants [3], [8], [11]–[14], [25], [27], [35], [60], [62], [64], [65], [75], [97]–[99], designed to overcome the Byzantine Generals' Problem in distributed settings and provide instant finality [63]. At the time of writing, 17 out of the top 80 chain projects adopt BFT consensus [36], such as Theta [77], neo [75], Algorand [5] and EOS [42].

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 [29]. 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) attacker $\mathcal{A}$, $0 \leq f \leq t = \lfloor n/3 \rfloor$ for asynchronous BFT [29]. BFT consensus protocol meets the following two properties: *(1) Consistency:* If some two correct nodes deliver any payload, then they deliver the same payload. *(2) Liveness:* It is assured that the nodes will terminate the protocol [27].

Next, we use *PBFT* [29], 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. For simplicity, we define a function **Leader**$(b, v, n) \rightarrow l$ for leader selection, which will return the leader index l at round b of view v. A *PBFT* protocol can be abstracted to the following five phases [29].

- *Initialize.* On input of a view v, round b and the number of participants n, a leader function **Leader** will output the leader index l.
- *Prepare.* The leader accepts requests and transactions from users, 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 $2t + 1$ "respond" messages.
- *Finalize.* If a node obtains more than $2t+1$ "commit" messages, a consensus is reached and the consensus node provides the execution result to users (reply).

2.2. MEV Attacks

An MEV [38], [55], [92], [94] attacker can manipulate a transaction list, such as inserting new transactions, swapping transaction orders, and removing transactions, in her 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 [38]. Most of the blockchains are vulnerable to MEV attacks [85] except for some confidential ledgers [32], [81]. There are many types of MEV attacks, such as front-running, back-running, and sandwich attacks [85]. We target MEV attacks that pose direct threat to runtime RNP, discussed in Section 3.3.

Runtime RNP Definition:

- **Setup**$(1^\lambda, k, n)$: The protocol generates a list of secret keys $\mathcal{SK} = \{sk_0, \cdots, sk_{n-1}\}$ and a list of public keys $\mathcal{PK} = \{pk_0, \cdots, pk_{n-1}\}$, where each secret key sk_i is only known to N_i. For a view v in round b, the leader node is decided by $l \leftarrow$ **Leader**(b, v, n).
- **Prepare**(l, b, v, sk_l): On input the leader index l in round b, view v, and its secret key sk_l, the leader node N_l computes a partial value $\sigma_{b,v,l}$ and outputs $(l, \sigma_{b,v,l})$.
- **Response**$(l, \sigma_{b,v,l}, b, v, sk_i)$: With input the leader's index l, its partial value $\sigma_{b,v,l}$ in view v of round b, and secret key sk_i of N_i, N_i computes and outputs its partial value $\sigma_{b,v,i}$.
- **Commit**$(\mathcal{PK}, \mathcal{E}, b, v, i)$: On input the public key vector $\mathcal{PK}$, a set of partial values $\mathcal{E} = \{(j, \sigma_{b,v,j})\}_{j \in I \wedge |I| \geq k}$, the round number b, and view v. Node $N_i \in \mathcal{N}$ computes a random beacon $\sigma^i_{b,v}$, outputs $(i, \sigma^i_{b,v})$, or $\perp$.
- **Finalize**$(\mathcal{E}^*, b, v)$: On input a set of beacon values $\mathcal{E}^* = \{\sigma^i_{b,v}\}_{i \in I \wedge |I| \geq 2t+1}$ from more than t nodes, a round b, and a view v. checks whether there exists a $\sigma'_{b,v} \in \mathcal{E}^*$ that appears at least $t+1$ times. Output $\sigma'_{b,v}$ or $\perp$.
- **Validator**$(\{\mathcal{SC}, \mathcal{T}\}, addr)$: On input $\{\mathcal{SC}, \mathcal{T}\}$, which contains the user contract $\mathcal{SC}$ and an invoking transaction $\mathcal{T}$, and an address $addr$, this algorithm verifies if $\mathcal{T}$ could revert the execution of $\mathcal{SC}$ and if $addr$ is a contract account, then returns 0 or 1.
- **GetRandom**$(\sigma, \mathcal{T}, m)$: On input the random beacon σ, the transaction $\mathcal{T}$, and the number of required random number m, this algorithm outputs m random number $\{r_{\mathcal{T},0}, r_{\mathcal{T},1}, \ldots, r_{\mathcal{T},m-1}\}$.

Figure 2: Runtime RNP definition for PBFT consensus (and its variants).

3. Runtime RNP Protocol

This section begins with a formal definition of a runtime RNP protocol in Section 3.1, followed by a discussion of the security assumption in Section 3.2. To ease illustration, we consider a *PBFT*-based blockchain system with smart contracts, comprising n nodes $\mathcal{N} = \{N_0, \cdots, N_{n-1}\}$. We use $\mathcal{C}$ to denote the set containing corrupted nodes controlled by the attacker $\mathcal{A}$. Let $a, b : \mathbb{N} \rightarrow \mathbb{N}$ be polynomial time functions where $a(\lambda)$, $b(\lambda)$ both are bounded by a polynomial in λ. Let domain `Dom` and range `Ran` be sets of size $2^{a(\lambda)}$ and $2^{b(\lambda)}$ respectively [47].

3.1. Formalization of Runtime RNP Protocol

The runtime RNP protocol generates a beacon for each block, which is then utilized to supply smart contracts with random numbers at runtime. We present the syntax of a runtime RNP protocol for *PBFT* consensus.

Definition 1 (Runtime RNP Protocol)**.** *Let the underlying PBFT be (f, t, n)-secure, a k-out-of-n runtime RNP based on a secure PBFT consensus with a set of nodes $\mathcal{N} = \{N_0, ..., N_{n-1}\}$ is specified as a tuple $\mathcal{R}$ = (Setup, Prepare, Response, Commit, Finalize, Validator, GetRandom) of polynomial algorithms detailed in Figure 2.*

3.2. Security Assumptions

Let the attacker $\mathcal{A}$ control up to $t = \lfloor n/3 \rfloor$ nodes, which may behave arbitrarily, i.e., being Byzantine faulty. Let $\mathcal{A}$ be static, i.e., it can only choose the nodes to be corrupted before the *PBFT* protocol starts. $\mathcal{A}$ has complete control over a faulty node, including its network connections and operating systems, i.e., it knows all the secret keys of the corrupted node and can forge any message on behalf of the corrupted node. $\mathcal{A}$ can further monitor the blockchain network. Thus, it knows all transactions and consensus messages. Hence, if the leader node is corrupted, it can propose a transaction list that benefits the attacker.

A node that is not corrupted throughout the consensus is considered honest and executes the PBFT protocol as specified. The honest nodes are assumed to have stable network connections, adequate computing power, and sufficient storage space to process blockchain transactions. Moreover, honest and corrupted nodes do not collude. When an honest node receives a valid transaction, it will immediately broadcast the transaction to all other consensus nodes.

Out of Scope. For the smart contract security, we only consider the MEV attacks that pose direct threats to the runtime RNP (see Section 3.3) and the identified malicious reversal issues (see Section 3.4). MEV attacks on other blockchain services, e.g., decentralized exchange [85], are not covered in this work. The existing countermeasures could be used for defense [11].

3.3. MEV Attacks for Runtime RNP

Once the beacon is known, before or 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. However, callback RNP is an expensive approach for supplying random numbers, whereas runtime RNP aims to provide random numbers within the same consensus round that generates $\mathcal{T}_{\texttt{request}}$, necessitating new mechanisms to protect against MEV attacks. Fair ordering of transaction list was proposed to defend against MEV [11], [56], [57]. However, our analysis below reveals that an attacker can perform MEV attacks against runtime RNP without modifying the transaction list; hence, fair ordering is insufficient to protect runtime RNP from MEV. There are four types of concerns associated with MEV attacks during the beacon generation.

- *Type 1: Predictable Beacon Generation.* The primary security risk for a runtime RNP is predictability. If the attacker is able to predict the upcoming beacon with a non-negligible probability before his transaction is broadcast, the attacker could undertake an MEV attack to his advantage. Therefore, the creation of beacons for a particular input should be *unpredictable*.
- *Type 2: Biased Beacon Generation.* Another issue is the probability of beacon generation is not uniform. Conse-

We first define two oracle queries used in the definition: *Partial Beacon Query* and *Beacon Query*. In response to an attacker $\mathcal{A}$'s query, the Partial Beacon Query returns the partial beacon of the selected honest node, whereas the Beacon Query returns the aggregated beacon from the specified honest node. These queries mimic $\mathcal{A}$'s capacity to acquire public messages from the network.

(a) Partial Beacon Query: With $\mathcal{A}$'s input $(\mathsf{Partial}, b, v, i)$ for some honest node $N_i \in \mathcal{N} \setminus \mathcal{C}$ at round b of view v, get the leader index $l \leftarrow$ **Leader**(b, v, n). It computes $(l, \sigma_{b,v,l}) \leftarrow$ Prepare(l, b, v, sk_l). If $i = l$, it returns $(l, \sigma_{b,v,l})$; otherwise, it returns $(i, \sigma_{b,v,i}) \leftarrow$ Response$(l, \sigma_{b,v,l}, b, v, sk_i)$. In any other cases, it returns $\perp$.

(b) Beacon Query: On input $(\mathsf{Beacon}, \mathcal{E}, b, v, i)$ for honest node $N_i \in \mathcal{N} \setminus \mathcal{C}$, where $\mathcal{E} = \{\sigma_{b,v,i}\}_{N_i \in \mathcal{C}}$ contains partial beacons from corrupted nodes. It collects partial beacons generated by honest nodes as $\mathcal{E}'$ and returns $\sigma^i_{b,v} \leftarrow$ Commit$(\mathcal{PK}, \mathcal{E}, \mathcal{E}', b, v, i)$ to the attacker.

Pseudorandomness:

A runtime RNP protocol is (f, k, n)-pseudorandomness if for all PPT adversaries $\mathcal{A}$, it holds that

$$\left|\Pr[\text{PRand}_{\mathcal{R},\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\text{PRand}_{\mathcal{R},\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \textit{negl}(\lambda) \tag{1}$$

where $\textit{negl}(\cdot)$ is a negligible function and the experiment $\text{PRand}_{\mathcal{R},\mathcal{A}}(1^\lambda, \mathbf{b})$ with $\mathbf{b} \in \{0, 1\}$ is defined as follows:

1. An attacker $\mathcal{A}$ chooses a collection $\mathcal{C}$ of nodes to be corrupted with $\mathcal{C} \subseteq \mathcal{N}$ and $|\mathcal{C}| \leq f$. Attacker $\mathcal{A}$ acts on behalf of corrupted nodes, while the challenger acts on behalf of the remaining nodes, behaving honestly.
2. Challenger and attacker engage in running the setup protocol Setup$(1^\lambda, k, n)$. Every honest node $N_i \in \mathcal{N} \setminus \mathcal{C}$ obtains a key pair (sk_i, pk_i). In contrast, corrupted nodes $N_j \in \mathcal{C}$ end up with key pairs (sk_j, pk_j) in which one of keys may be undefined (i.e. either $sk_j = \perp$ or $pk_j = \perp$). At the end of this phase, the public keys vector $\mathcal{PK}$ is publicly known by both sides.
3. The challenger receives from the attacker $\mathcal{A}$ a set of partial beacons $\{(i, \sigma_{b,v,i})\}_{N_i \in \mathcal{I}}$ with $\mathcal{I} \subseteq \mathcal{N}$ and $|\mathcal{I}| \geq k$, such that $\{(\mathsf{Partial}, b, v, i)\}_{N_i \in \mathcal{N} \setminus \mathcal{C}}$ are queried for different honest nodes. Let $\sigma_{b,v,j}$ be the output by querying $(\mathsf{Partial}, b, v, j)$ for $N_j \in \mathcal{I} \setminus \mathcal{C}$ and $\sigma^p_{b,v} \leftarrow$ Commit$(\mathcal{PK}, \{(j, \sigma_{b,v,i})\}_{N_i \in \mathcal{I}}, b, v, p)$ where p is randomly specified and $N_p \in \mathcal{N}$. If $\sigma^p_{b,v} = \perp$ the experiment outputs $\perp$. Otherwise, if $\mathbf{b} = 0$ the attacker receives $\sigma^p_{b,v}$; if $\mathbf{b} = 1$ the attacker receives a uniform random value in $\mathtt{Ran}$.
4. Finally $\mathcal{A}$ returns a guess $\mathbf{b}'$.

Uniqueness:

A distributed runtime RNP protocol achieves uniqueness if for all PPT adversaries $\mathcal{A}$,

$$\Pr[\text{URand}_{\mathcal{R},\mathcal{A}}(1^\lambda) = 1] \leq \textit{negl}(\lambda), \tag{2}$$

where the experiment $\text{URand}_{\mathcal{R},\mathcal{A}}(1^\lambda)$ is defined as follows:

1. 2. The same as step 1 and step 2 of pseudorandomness property correspondingly.
3. The challenger receives from $\mathcal{A}$ two different node sets $\mathcal{I}$, $\mathcal{I}'$, with $\mathcal{I}$ (resp.$\mathcal{I}'$) $\subseteq \mathcal{N}$ and $|\mathcal{I}|$ (resp. $|\mathcal{I}'|$) $\geq 2t + 1$, the round and view (b, v), and corresponding beacons $\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I}\text{ (resp.}\mathcal{I}')}$ from different nodes with beacons $\{\sigma^j_{b,v}\}_{N_j \in \mathcal{I} \setminus \mathcal{C}\text{ (resp.}\mathcal{I}')}$ from honest nodes acquired from beacon query. Then the challenger runs, $\sigma_{b,v} \leftarrow$ Finalize$(\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I}}, b, v)$ and $\sigma'_{b,v} \leftarrow$ Finalize$(\{\sigma^j_{b,v})\}_{N_j \in \mathcal{I}'}, b, v)$. If $\sigma_{b,v} \neq \perp$ and $\sigma'_{b,v} \neq \perp$ and $\sigma_{b,v} \neq \sigma'_{b,v}$, the challenger outputs 1; otherwise, it outputs 0.

Liveness:

A runtime RNP satisfies liveness if for all PPT adversaries $\mathcal{A}$,

$$\Pr[\text{LRand}_{\mathcal{R},\mathcal{A}}(1^\lambda) = 1] \leq \textit{negl}(\lambda), \tag{3}$$

where the experiment $\text{LRand}_{\mathcal{R},\mathcal{A}}(1^\lambda)$ is defined as follows:

1. 2. The same as step 1 and step 2 of pseudorandomness property correspondingly.
3. $\mathcal{A}$ sends a set $\mathcal{I} \subseteq \mathcal{N}$ of size at least $2t + 1$, and their corresponding beacons $\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I}}$ to the challenger with beacons $\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I} \setminus \mathcal{C}\text{ (resp.}\mathcal{I}')}$ from honest nodes acquired from beacon query. The challenger runs $\sigma_{b,v} \leftarrow$ Finalize$(\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I}}, b, v)$ and returns 0 if $\sigma_{b,v} \neq \perp$; otherwise, returns 1.

Figure 3: Formal definition of *Pseudorandomness*, *Uniqueness*, and *Liveness*.

quently, an attacker might manipulate the generation in favor of his attacker transaction. Thus, the RNP must be *bias-resistance*.

• *Type 3: Uniqueness of Beacon Generation.* Only one beacon will be produced for a given input and the result should be unique. Otherwise, the attacker may deceive the system into generating multiple beacons to disrupt the system functions.

• *Type 4: Liveness of Beacon Generation.* The attacker may prevent the random number generation from completing during a consensus round by not collaborating with other honest nodes, for example the corrupted nodes do not respond and wait for timeout. This will result in a view change. The attacker can keep interrupting until a controlled node becomes the leader and injects malicious transaction list. A runtime RNP protocol should tolerate corrupted nodes to defend against the resulting MEV attack, hence ensuring *liveness*.

3.3.1. MEV Resilience. As analyzed above, we define that an MEV-resilience runtime RNP protocol should satisfy the following properties:

• *Pseudorandomness.* Pseudorandomness guarantees that the beacon outputs are indistinguishable from uniformly random values in the presence of an active attacker who can receive beacons from other views or rounds, which implies

unpredictability (*Type 1*) and *bias-resistance* (*Type 2*) [47], [89], [95].

• *Uniqueness.* Uniqueness (*Type 3*) [47] requires that for a given input x a unique beacon σ is generated. It is infeasible for the attacker to compute two different output beacons, σ, σ', with the same input $x \in \mathtt{Dom}$.

• *Liveness.* Liveness (*Type 4*) [28], [59] ensures that all honest nodes will always generate a valid beacon on any admissible round b in an adversarial environment.

Those properties are formally defined in Figure 3. The formal definition of *MEV Resilience* for runtime RNP on *PBFT* platforms is given below.

Definition 2 (MEV Resilience)**.** *Assume the underlying PBFT is (f, t, n)-secure, then an runtime $\mathcal{R}$ = (Setup, Prepare, Response, Commit, Finalize, Validator, GetRandom) with a set of nodes $\mathcal{N} = \{N_0, ..., N_{n-1}\}$ is (f, k, n)-resilient to MEV attacks with $f \leq t \leq \lfloor n/3 \rfloor < k \leq 2t + 1 < n$ (refer to Section 4.1.2 for detail) if for any PPT attacker $\mathcal{A}$, it satisfies pseudorandomness, uniqueness, and liveness.*

With MEV Resilience we only consider MEV attacks on runtime RNP related to the discussed four types of scenarios defend against MEV attacks, which is consistent with the security requirements.

3.4. Threat of Malicious Reversal

Blockchain transactions are atomic [9]. 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 conducted actions. Atomicy is critical to ensuring the security of the blockchain state.

However, the attacker can abuse the reverting function and intentionally creates *malicious reversal* to compromise the fairness of the smart contract. Similar behaviors have already been observed in the blockchain. For example, FlashLoan attack [92] enables the attacker to revert the transaction to undermine security of a decentralized exchange. We found that this unauthorized reverse also presents great threats to a runtime RNP. For instance, a contract for the game rock-paper-scissors (see Section 6) generates shapes from the runtime random numbers and compares them with user's shapes; in a gambling game, the result is also decided by the generated random numbers. In either case, if the result is not satisfactory, a malicious user can reverse the execution until his/her goal is achieved.

In particular, malicious reversal allows the attacker to actively or passively validate the state changes of the user contract during the execution of the attacker transaction. In this work, we identify and consider four types of malicious reversal. The first two types, i.e. *Script MR* and *Contract MR*, can be performed by the attacker via the attacker transaction to validate the state change of the user contract, while the other two types, *fallback MR* and *Fee MR*, rely on the user contract for state validation.

• *Type 1: Script MR.* Transactions are stored in the format of scripts and will be loaded into the virtual machine (e.g. Neo Virtual Machine [75]) for execution. The attacker can add extra checking logic after the logic of calling the user contract in the transaction script to verify the execution result of the user contract. As demonstrated in Figure 4a, the VM executes the transaction by reading the transaction script. The VM first ❶ loads and executes the user contract, and then ❷ it executes the verification script after the user contract returns. If the result is not desirable, the attacker may revert the transaction intentionally.

• *Type 2: Contract MR.* Though Script MR is effective, it requires the attacker fully understand the processing mechanisms of the VM. In addition, due to the size limit of a transaction [6], [75], complicated logic could not be embedded in the script. On the other hand, Contract MR is more versatile. As shown in Figure 4b, an attacker can craft an attacker contract to involve more computationally intensive computations, from which the attacker can further ❶ call the user contract and ❷ check the result against its expectations. Contract MR can be initiated by deploying attacker contracts. This method is technically easier and provides access to more complex attack logic, such as repeatedly calling the user contract until the expected result is obtained. FlashLoan attack, which ranks second of the most common blockchain attacks [92], is essentially a Contract MR.

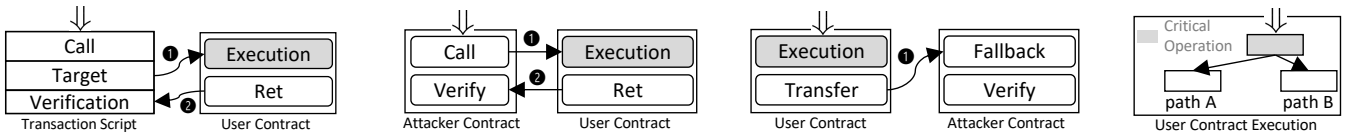
• *Type 3: Fallback MR.* Another stealthy malicious reversal behavior is fallback MR. For example, in the case of value transfer, the attacker can set the destination address to the account of his crafted malicious contract. As a result, the transfer operation will activate the fallback functions (e.g. fallback function in Ethereum [37] and OnNEP17Payment function in neo [73]) in the malicious contract, which allows the attacker to verify the execution result of the target contract, as illustrated in Figure 4c.

• *Type 4: Fee MR.* For the cases where the attacker cannot directly verify the state change, fee-based side channel could be leveraged to conduct malicious reversals without touching the user contract, we call this Fee MR. For instance, the execution of a target contract may contain two possible paths, path A and path B in Figure 4d. Assume that the attacker knows the gas costs of both paths and wants the execution of the Critical Operation (shaded area in the figure) to be reverted if path A is selected. In this case, the attacker can set the maximum transaction fee greater than the cost of path B but less than that of path A. It is much more challenging to identify Fee MR than other MR types.

3.4.1. Malicious Reversal Resilience.

Malicious reversals depends on the atomicity of transaction execution, which means that if a malicious reversal is triggered at any time point during transaction execution, the execution is immediately halted and then reverted [9]. Therefore, to protect a contract from malicious reversals, we must ensure that transactions cannot commence any reverting operations. In this work, we consider the defense against the above four malicious reversals and define the *Malicious Reversal Resilience* as follows:

Definition 3 (MR Resilience)**.** *A runtime RNP $\mathcal{R}$ = (Setup,*



(a) Script MR (b) Contract MR. (c) Fallback MR. (d) Fee MR.

Figure 4: Different types of malicious reversals. ⇓ represents the execution entry point. The Critical Operation s highlighted in gray are those that the attacker intends to undo. 4a *Script MR*: ❶ user contract is activated and ❷ after it returns, malicious verification script is executed. 4b *Contract MR*: Attacker deploys an attacker contract and ❶ calls the user contract from the attacker contract, then ❷ verifies the state change after the user contract returns. 4c *Fallback MR*: user contract ❶ transfers assets to the attacker contract which triggers the fallback function of the attacker contract. 4d *Fee MR*: Attacker sets the transaction fee to only be able to cover the desirable execution path B. Taking path A will reverse the critical operation.

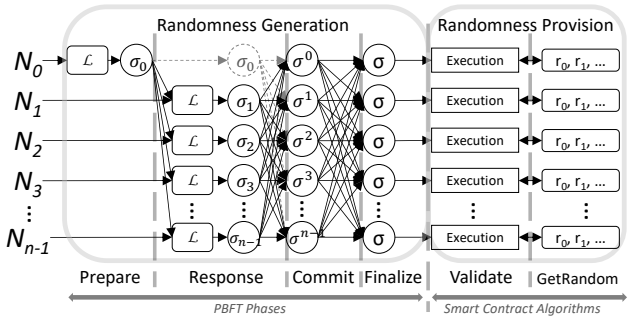


Figure 5: High-level BFTRAND description mapping to *PBFT* phases for randomness generation and smart contract execution for randomness provision. $\mathcal{L}$ is the *proposal* from the leader for some view v of round b. $\{\sigma_i\}$ are the partial beacons, whereas $\{\sigma^i\}$ are the beacons aggregated by nodes, and σ is the final beacon. $\{r_i\}$ are random numbers. Execution entails running smart contracts with transaction inputs on a virtual machine. In the figure, we omit the corresponding subscripts relating to the view v and round b in $\mathcal{L}$, σ_i, σ^i and σ.

Prepare, Response, Commit, Finalize, Validator, GetRandom) with a set of nodes $\mathcal{N} = \{N_0, \cdots, N_{n-1}\}$ is resilient to malicious reversal if the attacker cannot launch the identified four types of malicious reversals.

4. BFTRAND Protocol

We present BFTRAND, a runtime RNP protocol, for PBFT consensus. It may also be adapted to other BFT style consensus blockchain platforms. The overall protocol is illustrated in Figure 5, where randomness generation occurs in the `Prepare`, `Response`, `Commit`, and `Finalize` phases to produce beacons for blocks while randomness provision is described in the `Validator` and `GetRandom` algorithms to supply random numbers to smart contracts. A *proposal* is a candidate block containing the transaction list from the leader which will be confirmed at the end of consensus if over $2t$ nodes agree on it. The *proposal* for consensus round $b > 0$ at view $v \geq 0$ is denoted as $\mathcal{L}_{b,v}$. Following this, we slightly abuse the notation b to also indicate the block b.

DRB Definition:

- *Setup*$(1^\lambda, k, n)$. This is an interactive protocol run by the nodes in P to set up a random beacon committee. The protocol outputs a list of secret keys $\{sk_0, ..., sk_{n-1}\}$ and their corresponding public keys $\{pk_0, ..., pk_{n-1}\}$.
- *Partial*$(\mathsf{st}_{rn-1}, sk_i)$. On input a state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$ from round $rn-1$, a secret key sk_i, the algorithm computes a partial beacon σ_i. Output is (i, σ_i).
- *Comb*$(\mathsf{st}_{rn-1}, \mathcal{E})$. On input a state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$, a set $\mathcal{E} = \{\sigma_i\}_{i \in I}$ of partial values from $|I| \geq k$ different nodes, this algorithm outputs a beacon σ_i, or $\perp$.
- *Verify*$(\mathsf{st}_{rn-1}, \sigma_{rn})$. On input a state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$ and a beacon σ_{rn}, this algorithm verifies if σ_{rn} is valid by outputting 0 or 1.
- *UpdState*$(\mathsf{st}_{rn-1}, \sigma_{rn})$. On input the current state st_{rn-1}, a beacon $\sigma_{rn} \in \mathtt{Ran}$ generated at the end of round $rn-1$, the algorithm outputs the updated state st_{rn} for round rn, or $\perp$.

Security Requirements:

- *Pseudorandomness*. This ensures that the generated beacon cannot be distinguished from uniformly random value in the presence of active adversaries, denoting unpredictability and bias-resistance.
- *Uniqueness*. This requires the generated random beacons are unique even if the attacker can access the secret keys of honest parties.

Figure 6: Definition of our DRB protocol.

Distributed Random Beacon: The feasibility of constructing a DRB protocol has been demonstrated in the literature [47], [50], [86]. *DRB* provides a way to collaboratively agree on a (pseudo)random value without involving a central party. To create a beacon, each node generates a partial beacon, along 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 full beacon. Here we adapt the *DRB* scheme in [47] and give the formal definition in Figure 6.

We also assume there is an efficient pseudorandom function $\mathcal{F}$. Let $\mathcal{F}$ be an efficient keyed function family, $\mathcal{F} : \{\mathcal{K} \times \mathcal{X} \to \mathcal{Y}\}$, where $\mathcal{K}, \mathcal{X}$ and $\mathcal{Y}$ are indexed with a security parameter λ. For a function F_k in $\mathcal{F}$ where its key k is uniformly chosen from $\mathcal{K}$, for any PPT attacker $\mathcal{A}$, the following holds,

$$\left|\Pr[\mathcal{A}^{F_k(\cdot)} = 1] - \Pr[\mathcal{A}^{f(\cdot)} = 1]\right| \leq negl(\lambda), \qquad (4)$$

where f is a function uniformly chosen from the set of functions mapping from $\mathcal{X}$ to $\mathcal{Y}$.

Randomness Generation Protocol *(b, v, l)*

• **Setup:** Let *DRB* be a (k, n)-threshold decentralized random beacon protocol. A set $\mathcal{N}$ of n nodes jointly run *DRB.Setup*$(1^\lambda, k, n)$ to generate a pair of secret key sk_i and corresponding public key pk_i for each node. Public key vector $\mathcal{PK}$ that contains all public keys is available to all nodes. Let *PBFT* be the (f, t, n)-secure BFT consensus and $0 \leq f \leq t < k \leq 2t + 1$.

• ***PBFT* Prepare Phase:** After generating a *proposal* $\mathcal{L}_{b,v}$, the leader N_l gets the current state $\mathsf{st}_{b,v}$ from *DRB.UpdState*, then compute a partial value $(l, \sigma_{b,v,l}) \leftarrow$ *DRB.Partial*$(\mathsf{st}_{b,v}, sk_i)$ for current consensus. Broadcasts $(prepare, l, \mathcal{L}_{b,v}, b, v, \sigma_{b,v,l})$.

• ***PBFT* Response Phase:** When $N_i \in \mathcal{N}$ receives a valid *prepare* message from N_l for view v of round b for the first time, the node gets the state $\mathsf{st}_{b,v}$ from *DRB.UpdState*, then computes a partial value $(i, \sigma_{b,v,i}) \leftarrow$ *DRB.Partial*$(\mathsf{st}_{b,v}, sk_i)$. Broadcast *response* message $(response, i, \mathcal{L}_{b,v}, b, v, \sigma_{b,v,i})$.

• ***PBFT* Commit Phase:** When a node N_i receives valid *response* messages from over $2t - 1$ different nodes for view v of round b, it gets $\mathsf{st}_{b,v}$ from *DRB.UpdState*, then computes the random beacon $\sigma^i_{b,v} \leftarrow$ *DRB.Comb*$(\mathsf{st}_{b,v}, \mathcal{PK}, \{j, \sigma_{b,v,j}\})$ where $\sigma_{b,v,j}$ is parsed from the *prepare* and *response* message. If $\sigma^i_{b,v} = \perp$, then broadcast *view change* message $(viewchange, i, b, v)$; otherwise, broadcast commit message $(commit, i, \mathcal{L}_{b,v}, b, v, \sigma^i_{b,v})$.

• ***PBFT* Finalize Phase:** When a node N_i receives over $2t$ valid *commit* messages from different nodes for view v of round b, if there exists a $\sigma \in \{\sigma^j_{b,v}\}$, where $\sigma^j_{b,v}$ is parsed from *commit* message, that appears at least $t + 1$ times, node N_i constructs a new block for round b and broadcast $(block, \mathcal{L}_{b,v}, b, \sigma)$.

Figure 7: Secure randomness generation protocol for BFTRAND based on *PBFT* consensus. We assume that all honest nodes are at the same view v of round b.

4.1. Randomness Generation from *DRB*

We adapt the threshold *DRB* scheme in [47] to design our randomness generation protocol for BFTRAND, as illustrated in Figure 7.

4.1.1. Integration of DRB and PBFT. BFTRAND is based on *PBFT*. It is important to seamlessly incorporate the DRB into PBFT. To this end, the beacon generation of *DRB* needs to end before the Finalize phase of *PBFT* in order to not interrupt the consensus. In addition, the aggregated beacon should be broadcast with the commit message. The *PBFT* leader should also leverage *DRB* to generate the partial beacon during the *PBFT* Prepare phase and broadcast it along with the prepare message. The other consensus nodes construct their partial beacons during the *PBFT* Response phase and send them along with the response message.

4.1.2. The Range of *DRB* Threshold k. The *DRB* derives the random beacon from the threshold aggregation of k or more authenticated partial beacons. To ensure that the *DRB* threshold k is consistent with the Byzantine assumption of *PBFT*, we discuss the valid range of k. First, k should be bigger than the maximum *PBFT* Byzantine tolerance t. If $k \leq t$, the attacker $\mathcal{A}$ would be able to aggregate the beacon in advance, since it controls at most t nodes. In addition, k should be equal or less than $2t + 1$. Otherwise *DRB.Comp* would require over $2t + 1$ partial beacons to aggregate the final beacon, which means the aggregation needs coorperation from Byzantine nodes. As a result, k must fall in $(t, 2t + 1]$.

4.1.3. The Gap Between rn and (b, v). The *DRB* operates in rounds, with each round rn generating a random beacon. Meanwhile, *PBFT* operates in accordance with the consensus round b, and runtime RNP requires only one random beacon per consensus round. Thus, the intuitive design utilizes the consensus round b as the *DRB* round rn, i.e., $rn \leftarrow b$. However, combining *DRB* with *PBFT* may raise a security concern as the *PBFT* consensus may produce multiple views. *PBFT* will switch views if the consensus fails in current view, and restart the consensus in a new view. If *DRB* and *PBFT* use the same number of rounds, different views, v and $v+1$, of the same consensus round will have the same input state, which is $\mathsf{st}^{b,v}_{rn} = \mathsf{st}^{b,v+1}_{rn}$, and therefore get the same random beacon. Henceforth, if the prior consensus changes view at or after the *PBFT* Commit phase, the attacker $\mathcal{A}$ may compute the random beacon even before the new view starts. To ensure the random beacon remains unknown during the whole consensus round, we append v to the *DRB* round. Specially, we define rn as a combination of round b and view v : $rn \leftarrow (b, v)$. As a result, the state update algorithm *DRB.UpdState* works as follows: If the consensus is a new round, then $\mathsf{st}_{b,v} \leftarrow \sigma_{b-1} \parallel b - 1$; if the consensus is a new view of the same consensus round, then $\mathsf{st}_{b,v} \leftarrow \mathsf{st}_{b,v-1} \parallel b \parallel v - 1$.

4.2. Randomness Provision Library

The existing RNP efforts, such as VRF [31], [70], VDF [21], and DRB [16], [47], concentrate on how to generate random numbers for blockchains in a secure and efficient manner. If the random numbers (or beacons) generated by these RNPs are used in contracts at runtime, smart contracts would be vulnerable to malicious reversals, as analyzed in Section 3.4. Nonetheless, as the first secure runtime RNP protocol, BFTRAND has designed the protocol with the security of randomness provision to smart contracts in mind. The randomness provision of BFTRAND is structured as a smart contract programming library, which contains two functions: *Validator* function, which detects malicious reversals, and *GetRandom* function, which generates the required random numbers.

4.2.1. Transaction Validator Function. Without mitigating the malicious reversals, BFTRAND is impossible to achieve secure runtime RNP. An intuitive but effective defense against malicious reversals can be described as below. To defend against the Script MR (*Type 1*), we check the transaction script size, making sure that the invoking transaction is not attached with extra verification script; check the invoking script from the user contract, making sure the user contract is invoked directly from a transaction, not from another contract (*Type 2*); and check if the user contract interacts with an attacker contract (*Type 3*); then verify the

Randomness Provision Lib ($\mathcal{SC}, \mathcal{T}, \sigma$)

Function Validator:

input : A smart contract $\mathcal{SC}$ and a transaction $\mathcal{T}$; or an address $addr$.
output: A bit B that is either 1 or 0.

1 *For simplicity, we use the name of the contract as the contract address and abstract invoking function of $\mathcal{SC}$ as invoking $\mathcal{SC}$:*
2 **begin**
3 $B \longleftarrow 0$
4 **if** $\{\mathcal{SC}, \mathcal{T}\} \neq \perp$ **then**
5 $(\$G_{min}, S_{max}) \longleftarrow \mathcal{SC}$ /* Minimum transaction fee and maximum script size from $\mathcal{SC}$. */
6 $(\$g_{max}, S) \longleftarrow \mathcal{T}$ /* Maximum transaction fee and script of $\mathcal{T}$. */
7 **if** $\$G_{min} \leq \$g_{max}$ **and** $S_{max} \geq |S|$ **then**
8 $\mathcal{SC}_{entry} \longleftarrow EntryScriptFromVM$ /* Acquire the entry script from the virtual machine. */
9 **if** $\mathcal{SC}_{entry} = S$ **then**
10 $B \longleftarrow 0$ /* No malicious reversal. */
11 **else**
12 $B \longleftarrow 1$
13 **else**
14 $B \longleftarrow 1$
15 **else**
16 $S \longleftarrow GetContract(Addr)$ /* Get the script from the blockchain ledger. */
17 **if** $S \neq \perp$ **then**
18 $B \longleftarrow 1$
19 **else**
20 $B \longleftarrow 0$ /* No malicious reversal. */
21 **return** B

Function GetRandom:

input : The number of random numbers m; A beacon σ; A transaction $\mathcal{T}$.
output: A set of distinct random numbers R of size m.

22 **begin**
23 $R \longleftarrow \emptyset$ /* Set of random numbers. */
24 $I \longleftarrow 0$
25 $H \longleftarrow \mathcal{T}$ /* Acquire the Transaction hash */
26 **while** $|S| \neq m$ **do** /* Need m random numbers */
27 $r_I \longleftarrow PRF(\sigma \parallel H \parallel I)$;
28 **if** $r_I \neq \perp$ **then**
29 $AppendToR(r_I)$
30 $I \longleftarrow I + 1$
31 **return** R

Figure 8: Smart contract programming library for randomness provision of BFTRAND.

maximum available fee of the invoking transaction and make sure it covers the all execution paths (*Type 4*).

Function Validator of the randomness provision library, as depicted in Figure 8, models the malicious reversal checking mechanism to defend against Script MR, Contract MR, Fallback MR, and Fee MR. The Validator function accepts two parameters, one is $\{\mathcal{SC}, \mathcal{T}\}$ that contains the user contract $\mathcal{SC}$ and the invoking transaction $\mathcal{T}$, another is $addr$ which is an address the user contract intends to interact. If $\{\mathcal{SC}, \mathcal{T}\}$ is not empty at *Line 4*, Validator goes into the Script MR, Contract MR, and Fee MR detecting code. *Line 5* gets two contract variables, $\$G_{min}$ and S_{max}, specified when the contract is developed. $\$G_{min}$ indicates that the minimum transaction fee required to cover all execution paths and S_{max} is the maximum transaction script size allowed to ensure that no verification script is appended. *Line 6* extracts the maximum transaction fee $\$g_{max}$ and transaction script S from the transaction parameters. Then, at *Line 7*, Validator confirms that there are no Fee MR concerns by ensuring the transaction has sufficient transaction fee that is greater than the required minimum fee, and that there is no Script MR threat by comparing the size of the transaction to the maximum script size provided in the contract. If any check at *Line 7* fails, the Validator returns a bit 1 to alert potential malicious reversal issues. In other cases, the Validator further verifies whether the contract execution is directly initiated by a transaction by acquiring the entry script of the execution from the underlying virtual machine at *Line 8*. Validator returns 0 if the acquired entry script and the transaction script are identical; otherwise, it returns 1. *Line 16* is where Validator attempts to retrieve the contract script for $addr$ from the blockchain ledger. If the loaded script is not empty, suggesting that the address is a contract account, then a Fallback MR may exist and Validator returns 1; otherwise, it returns 0.

This work aims to raise awareness of the relevant malicious reversal vulnerabilities. Our solutions are effective and are ready for implementation. Meanwhile, it still has a room for further optimization for better usability by considering additional factors, such as access control/authorization functions, which we leave as future work.

4.2.2. Random Number Generating Function. The GetRandom function in Figure 8 shows the algorithm that turns the beacon into multiple random numbers. Each block only contains a single random beacon. However, transactions within the same block need different random numbers for security. Therefore, we apply a *PRF* to expand the beacon into multiple random numbers for the transactions. To ensure that the transactions receive unique random numbers, the *PRF* must be able to distinguish across transactions; thus, at *Line 25*, we use the hash of the transaction as the transaction identifier and feed it into *PRF*. We also skip empty random values at *Line 28*.

5. Security Analysis of BFTRAND Protocol

In this section, we analyze the security of BFTRAND. BFTRAND aims to achieve the following security requirements described in Section 3: (1) MEV resilience for runtime RNP, and (2) Malicious reversal resilience. The formal proofs of the following lemmas and theorems are given in Appendix D.

5.1. MEV Resilience

We show that BFTRAND can defend against MEV attacks on runtime RNP by achieving *pseudorandomness*, *uniqueness* (Lemma 5.1), and *liveness* (Lemma 5.2), hence achieving MEV resilience.

Lemma 5.1. *If PBFT is (f, t, n)-secure and DRB is (f, k, n)-secure for any f with $f \leq t < k \leq 2t + 1$, then* BFTRAND *satisfies pseudorandomness and uniqueness.*

Sketch. This follows directly from the *pseudorandomness* and *uniqueness* of the underlying *DRB* protocol and the property of *PBFT* such that $\mathcal{A}$ can only collude at most t nodes. See the full proof in Appendix D.1.1.

Lemma 5.2. BFTRAND *satisfies liveness if PBFT is (f, t, n)-secure and DRB satisfies uniqueness.*

Sketch. As BFTRAND receives at least $2t+1$ beacons in the Finalize phase that are generated by different nodes and the beacon can be confirmed only if there exists a value that appears over t times. Following the uniqueness of *DRB*, honest nodes generates the same beacons; hence, honest nodes of BFTRAND can always output a valid beacon, even in the presence of faulty nodes. Therefore, BFTRAND achieves *liveness*. See the full proof in Appendix D.1.2.

Theorem 5.1 (MEV Resilience)**.** *Assume the underlying DRB is (f, k, n)-secure under definition 4, PBFT is (f, t, n)-secure, and* BFTRAND *achieves pseudorandomness, uniqueness, and liveness,* BFTRAND *achieves MEV Resilience in Definition 2.*

Proof. This is guaranteed from Lemma 5.1 and Lemma 5.2. □

5.2. MR Resilience

We prove that BFTRAND protocol is resilient to malicious reversal by proving that transaction invoking the user contract in BFTRAND cannot perform malicious reversals identified in Section 3.4.

Theorem 5.2 (MR Resilience)**.** *If the user contract of* BFTRAND *calls the Validator function with all required parameters correctly specified for functions that issue runtime random number requests, then* BFTRAND *achieves MR Resilience under Definition 3.*

Sketch. Validator function verifies whether there are additional transaction scripts (*Type 1*), whether the contract are directly invoked by a transaction (*Type 2*), whether the interacting address is a contract account (*Type 3*), and whether there are adequate fees to cover all execution flows (*Type 4*). By eliminating and checking all conceivable methods for state detection and execution reverting at runtime, it ensures that user contracts execution cannot be maliciously reverted. Thus, *malicious reversal resilience* is achieved. See the full proof in Appendix D.2.

6. Implementation

We implemented a BFTRAND prototype on the neo blockchain platform [75], one of the most popular BFT projects at the time of writing. The neo 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* [29] with 15 seconds for consensus. The dBFT is a typical variation of the PBFT, and protocols that run on the dBFT can be easily adapted to other BFT platforms. We implemented BFTRAND into dBFT by adding a few more fields to the messages of the classic phases, i.e. `Prepare`, `Response`, `Commit`, and `Finalize`, without altering its original consensus function. To make the beacon accessible by the neo virtual machine, we change the "nonce" field in the neo header to 32 bytes to store the beacon[2]). The DRB protocol that we adopt to implement the BFTRAND prototype is the threshold BLS based Dfinity-DRB [47], [50].

Applications. We describe the deployed applications that rely on random numbers in order to show the advantage of BFTRAND:

• *Fair NFT Distribution.* Loot [80] 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. This is a platform-independent issue that will be encountered by similar NFTs on any platform. 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 BFTRAND 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 randomly. We constructed a rock-paper-scissors (RPS) contract that allows participants to play this game with the contract by sending a transaction containing the user's shape. When the RPS 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 RPS contract; if the user wins, the RPS contract pays back to the user 2 GAS; otherwise, the contract retains the user's stake.

• *Comparison.* To compare BFTRAND with callback RNP solutions, we implemented a comparison application

2. This step is not necessary. We make the change for ease of demonstration. See more discussions in Section 4.1 and Appendix F

Table 2: Applications Transaction Fee (GAS/$).

Method	Network Fee	System Fee
Loot::tokenURI	0.00593250/0.013	0.20694257/0.459
Neoverse:: UnBoxing	0.00119552/0.002	0.07313472/0.162
Neoverse:: BulkUnBoxing	0.00125752/0.002	0.36183988/0.803
RPS:: OnNEP17Payment	0.00616260/0.013	0.06588677/0.146

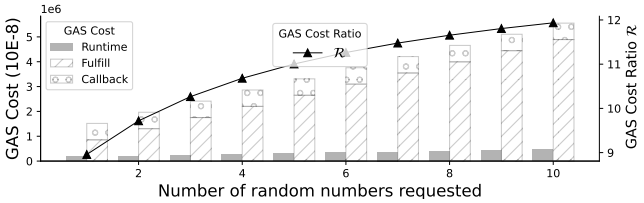


Figure 9: GAS cost when calling `GetRandom`. The left y axis is the total GAS consumption, while the right y axis is the GAS cost ratio $\mathcal{R} = (\texttt{Callback} + \texttt{Fulfill})/\texttt{Runtime}$.

that contains three functions: `Runtime()` to work as BFTRAND, 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. `Fulfill()` takes multiple parameters, including a random value, proof of the random value, and a request ID. When multiple random numbers are requested, `Fulfill()` calls "CryptoLib.Sha256" [74] to expand the received random number. We denote `Callback()` and `Fulfill()` together as BFTRAND$_{callback}$.

7. Evaluation

In this section, we show the performance of BFTRAND and demonstrate the advantage of BFTRAND by comparing BFTRAND with other RNPs. We evaluated BFTRAND with a quad-core 3.6 GHz Intel(R) E3-1275 v5 CPU [54] and 32 GB of memory. The operating system is Ubuntu 20.04 LTS with the Linux kernel version 5.4.0-91-generic, and the SDK is .NET 6.0.1. Lines of code is in the Appendix E.

Application Transaction Cost. Table 2 summarizes the transaction costs related to running the developed applications. In neo, the network fee is proportionate to the length of the transaction script, while the system fee is determined by the OpCodes that a transaction executes. GAS is the token in neo that pays transaction fee; currently, one GAS costs $2.22, and $1 buys 0.45 GAS. Loot::tokenURI provides a 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. RPS::OnNEP17Payment (OnNEP17Payment is the payable function defined in NEP-17 proposal of neo [73]) 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.

Transaction Fee Cost. We compare the fees to request multiple random numbers between BFTRAND and BFTRAND$_{\texttt{callback}}$. Since BFTRAND$_{\texttt{callback}}$ needs two transactions to complete a random number request, we compare the cost of BFTRAND with the sum of two transactions of BFTRAND$_{\texttt{callback}}$. Generating multiple random numbers with BFTRAND$_{\texttt{callback}}$ is implemented in C# on the neo platform based on Chainlink VRF[3]. The evaluation result is shown in Figure 9. We can see that if one transaction only requests one random number, the fee to request one random number with BFTRAND is 0.00169291 GAS ($0.037), which is only 11% of the cost of issuing a random number request for BFTRAND$_{\texttt{callback}}$ (0.01516372 GAS, $0.033). If every transaction requests 10 random numbers, then BFTRAND saves 91.6% GAS (0.05088786 GAS, $0.113). BFTRAND saves a considerable amount of transaction fees because BFTRAND$_{\texttt{callback}}$ pays transaction fee for two transactions, `Callback` and `Fulfill`. `Fulfill` needs to verify the fairness of the beacon and extend the beacon into random numbers in smart contract via the virtual machine, which are expensive and inefficient, whereas BFTRAND generates random numbers in the native environment and guarantees the beacon fairness through consensus.

GAS Cost of Transaction Validity Check. The cost to run the MR defender is less than 0.00001 GAS ($0.00002), which is negligible in comparison with the transaction cost.

Blockchain Overhead. To evaluate the impact of RNPs on the blockchain ledger, we define the size of data that has to be written onto the blockchain to complete a random number request as blockchain overhead (BO). Since Chainlink VRF [30] is one of the most influential RNP solutions, we use it as the baseline. Let the transaction size of $\mathcal{T}_{\texttt{callback}}$ be len. We evaluate the BO with the $\mathcal{T}_{\texttt{callback}}$ transaction size $len = 193$ bytes (the size of a typical transfer transaction in neo) and $len = 624$ bytes[4] while $\mathcal{T}_{\texttt{request}}$ size is set as the neo typical transaction size to 193 bytes. The BO for BFTRAND is 193 bytes, and the overhead for BFTRAND$_{callback}$ is $193 + len$. If $len = 193$ bytes, then BFTRAND saves about 50% BO versus BFTRAND$_{\texttt{callback}}$; if $len = 624$ bytes BFTRAND saves around 76.4% BO.

Evaluation on BLS Signature. Since the adopted DRB exploits BLS signature to generate the random beacon, 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 (consistent with the 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

3. Chainlink VRF [31] is the most influential RNP solution, which has processed about 8 million random number requests for over 656 projects.

4. The size of a callback transaction in Chainlink VRF [44], which includes the random number, proof, contract addresses, accounts, and other required parameters that cannot be ignored to verify the security properties of the random number generated off the blockchain.

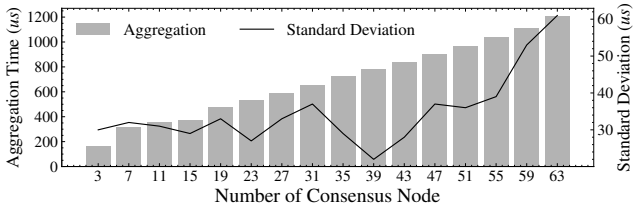


Figure 10: The BLS signature aggregation time.

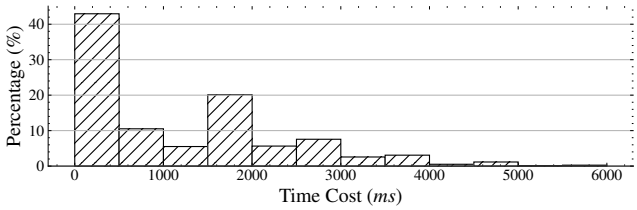


Figure 11: BLS setup time cost in a testnet with 7 consensus nodes.

BFTRAND to the neo blockchain only incurs a negligible 0.002% more consensus time.

To evaluate the setup, we run it 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 LTS. The ttl between nodes is about $57\,ms$. The evaluation result is shown in Figure 11 with $2,000$ 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 of collecting messages from all participants. Since the setup is separate from the consensus and runs only when the consensus committee needs to update, the execution of setup will not impose significant overhead to the consensus layer.

RNP Solutions Comparison. In Table 3, We compare BFTRAND with a host of random number-related projects, such as beacon protocols [16], [26], [33], [49], [95], [95], RNP-enabled blockchain projects [2], [17]–[20], and oracles [31], [76]).

• *Platform Consensus.* According to the table, the majority of 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 [23] and PoX-based [44] blockchain platforms. Automata [76] is a blockchain middleware, specifically built for Ethereum. As BFTRAND is built on top of *PBFT*, it can be easily ported to any *PBFT* variants based blockchain project.

• *Method.* To avoid a faulty node from blocking random number generation/utilization, HDrand, RandRiper, Dfinity, Elrond, and BFTRAND 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) [7], [10], [69], a hardware-protected isolated execution environment, to generate random values in a discreet and unbiased manner within the TEE. TEE is a hardware solution and requires 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 σ. While Secret and Elrond provide built-in pseudorandom functions that can extend σ into numerous random numbers, the random value they can supply is only upper bounded by the consensus (GAS limit for example), which we represent as ∞. Because BFTRAND uses PRF to safely generate random numbers from the beacon constructed in the consensus, the number of random numbers can be generated by BFTRAND is likewise constrained by the consensus, such as transaction fee.

• *Runtime.* Due to MEV and MR, none of the RNPs used by random beacons or blockchain projects can generate random values within a single consensus round, or at 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 [76] is a trusted third party middleware solution for EVM platform. It generates random numbers in TEE via VRF and uses Ethereum EIP712 [87] to encapsulate a user transaction and its requested random numbers inside an EIP712 transaction. Although Automata is theoretically vulnerable to MR, it is unlikely in practice since it is a closed solution that only operates on their own services, such as NFTFair. Compared with Automata, BFTRAND is a runtime RNP solution that focuses on BFT-based open blockchain platforms and does not rely on special hardware and trusted third party. In summary, BFTRAND is currently the only safe runtime random number provider for BFT-based blockchains.

8. Related Work

Random Number Providers for Smart Contracts. Existing works on random beacons adopt different cryptographic primitives and threat models [15], [26], [48], [49], [58], [89], [90]. Some projects [33], [79] are built with homomorphic encryption, which introduces heavy overhead to the system. Hedera Hashgraph [61] is a fault-tolerant RNP algorithm for a public directed acyclic graph. There are also projects running in a permissionless setting [1], [4], [17], [20], [31], [39], [40], [45], [61], but they do not support runtime random number provision, while BFTRAND is a runtime RNP protocol for smart contracts.

Hardware-based Random Number Providers. An ASIC-based VDF [21] is used as a source of randomization in ETH 2.0 [43]. Chainlink VRF [31] enables smart contracts to interact with a TEE-based oracle to obtain randomness and cryptographic proof. Secret [20] uses a TEE-based secret oracle to provide on-chain random service. The random

Table 3: Comparison of RNPs for blockchain.

Protocol	Platform Consensus	Method(s)	Resistance (t)	# random values (r)	Runtime
Drand [26]	PABFT	Threshold SecretBLS	$t < n/2$	$\mathcal{O}(\sigma)$	✗
HERB [33]	∅	Threshold ElGamal	$t < n/3$	$\mathcal{O}(\sigma)$	✗
RandChain [49]	Sequential PoW	PoW	$t < n/3$	$\mathcal{O}(\sigma)$	✗
RandHerd [95]	BFT	Threshold Schnorr	$t < n/3$	$\mathcal{O}(\sigma)$	✗
RandHound [95]	BFT	Client based, PVSS	$t < n/3$	$\mathcal{O}(\sigma)$	✗
BRandRiper [16]	BFT	VSS, q-SDH	$t < n/2$	$\mathcal{O}(\sigma)$	✗
Dfinity [2]	BFT	Threshold BLS	$t < n/2$	∞	✗
Secret [20]	DPoS	Scrt-RNG,TEE	$t < n/2$	∞	✗
Elrond [17]	Secure PoS	BLS,onchain data	$t < n/3$	∞	✗
Klaytn [19]	Istanbul BFT	VRF	$t < n/3$	$\mathcal{O}(\sigma)$	✗
Harmoney [18]	Fast BFT	VRF,VDF	$t < n/3$	$\mathcal{O}(\sigma)$	✗
⋆Chainlink VRF [31]	∅	VRF, TEE	$t < n/2$	$\mathcal{O}(\sigma)$	✗
⋆Automata [76]	∅	VRF, TEE	$t < n/2$	∞	✓
BFTRAND$_{callback}$	BFT	∅	$t < n/3$	$\mathcal{O}(\sigma)$	✗
BFTRAND	BFT	Threshold BLS	$t < n/3$	∞	✓§

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

beacon service of oasis [81] provides unbiased randomness on each epoch. All these methods rely on specific hardware and introduces extra trust to the blockchains. BFTRAND, in contrast, is a hardware-independent RNP solution.

BFT-based Blockchain Projects. Several existing works employ BFT-family protocols as consensus. Celo [82] is built on the Istanbul BFT. Kadena [83] originally uses the novel ScalableBFT. Solana's [93] Tower Consensus relies on a BFT mechanism. Multi-Level BFT is used by Theta [77]. Klaytn [19] uses an optimized version of Istanbul BFT. The Helium [51] is based on a variant of the HoneyBadgerBFT [71]. The EOS [42] utilizes a BFT modeled consensus mechanism. Algorand [5] adopts a PoS and BFT mixed consensus. Harmony proposes Fast BFT, which reduces communication costs by using BLS aggregate signature [18]. As BFTRAND is a BFT-based runtime RNP protocol, it can potentially be deployed on these projects.

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

9. Discussion

Scalability. BFTRAND is designed for BFT 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-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 [42]. Note that BFTRAND 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 BFTRAND to aggregate signatures is negligible compared to 15 seconds consensus intervals on neo.

BFTRAND *Integration.* It is straightforward to apply BFTRAND to existing BFT-based blockchain projects to support runtime RNP without a hard forks. See Appendix F for more details.

Future Work. Note that BFTRAND 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. And because permissioned and permissionless consensuses are fundamentally different, BFTRAND does not apply to blockchain platforms that adopt alternative consensus, such as Ethereum [24] and Bitcoin [72], 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 on-chain costs and delayed processing time. In this work, we explore and design BFTRAND, a novel runtime RNP protocol 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 a prototype of BFTRAND with a distributed random beacon protocol and incorporate it into the BFT consensus layer without affecting the efficiency. BFTRAND has been proven to be secure against MEV attacks. In addition, we also identify multiple types of malicious reversals and propose effective mitigation to enhance the security of BFTRAND. As a result, BFTRAND is the first of its kind for blockchains that need secure, reliable, and runtime random number provision function.

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] ALGORAND. Algorand consensus. https://developer.algorand.org/docs/get-details/algorand_consensus/.

[6] ALLISON, I. Ethereum's vitalik buterin explains how state channels address privacy and scalability, 2016.

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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).

[14] 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.

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

[16] 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.

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

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

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

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] CASCUDO, I., AND DAVID, B. Scrape: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security* (2017), Springer, pp. 537–556.

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

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

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

[32] 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.

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

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

[35] 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.

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

[37] CONSENSYS. Fallback functions. https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/fallback-functions/, Mar. 2022.

[38] 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).

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

[40] DRAND. Drand/drand: a distributed randomness beacon daemon - go implementation. https://github.com/drand/drand.

[41] ECOSYSTEM, C. Chainlink ecosystem. https://www.chainlinkecosystem.com/ecosystem, 2022.

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

[43] ETHEREUM.ORG. Ethereum upgrades (formerly 'eth2'). https://ethereum.org/en/upgrades/.

[44] ETHERSCAN. Chainlink vrf. https://etherscan.io/address/0x271682DEB8C4E0901D1a1550aD2e64D568E69909/, 2022.

[45] FETCH.AI. Launching our random number beacon on binance smart chain. https://medium.com/fetch-ai/launching-our-random-number-beacon-on-binance-smart-chain-8e3b7aa52be6, Oct 2020.

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

[47] 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.

[48] 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.

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

[50] HANKE, T., MOVAHEDI, M., AND WILLIAMS, D. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548* (2018).

[51] HELIUM.COM. Helium documentation. https://docs.helium.com/blockchain/consensus-protocol/, 2022.

[52] 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.

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

[54] 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.

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

[56] KELKAR, M., DEB, S., LONG, S., JUELS, A., AND KANNAN, S. Themis: Fast, strong order-fairness in byzantine consensus. *Cryptology ePrint Archive* (2021).

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

[58] 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.

[59] KIAYIAS, A., RUSSELL, A., DAVID, B., AND OLIYNYKOV, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference* (2017), Springer, pp. 357–388.

[60] 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.

[61] 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.

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

[63] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. In *Concurrency: the works of leslie lamport*. 2019, pp. 203–226.

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

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

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

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

[68] 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.

[69] 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.

[70] MICALI, S., RABIN, M., AND VADHAN, S. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)* (1999), IEEE, pp. 120–130.

[71] 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.

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

[73] NEO. Nep-17. https://docs.neo.org/docs/en-us/develop/write/nep17.html, Nov. 2022.

[74] NEO PROJECT, T. Neo native contract. https://github.com/neo-project/neo/blob/master/src/neo/SmartContract/Native/CryptoLib.cs.

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

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

[77] NETWORK, T. Theta network. https://docs.thetatoken.org/docs/whitepapers.

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

[79] 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.

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

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

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

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

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

[85] QIN, K., ZHOU, L., AND GERVAIS, A. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 198–214.

[86] RAIKWAR, M., AND GLIGOROSKI, D. Sok: Decentralized randomness beacon protocols. *arXiv preprint arXiv:2205.13333* (2022).

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

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

[89] 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.

[90] 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.

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

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

[93] SOLANA.COM. Solana documentation. https://docs.solana.com/introduction, 2022.

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

[95] 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.

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

[97] 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.

[98] 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.

[99] 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.

[100] 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.

Appendix A. Abbreviations

Abbreviation	Name
(P)BFT	(Practical) Byzantine Fault Tolerance.
DRB	Distributed Beacon Generator.
MEV	Miner or Maximum Extractable Value.
MR	Malicious Revert.
(D)BLS	Distributed Boneh–Lynn–Shacham.
PRF	Pseudorandom Function.

Appendix B. 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 [80].

```
/// 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 [78]. It is vulnerable to MEV attacks.

Appendix C. Primitives

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}.

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 $g^z = g^{ab}$.

Gap Diffie-Hellman (GDH) Group. A GDH group is a group where CDH problem is hard but DDH problem is easy.

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 attacker $\mathcal{A}$ is asked to output a pair m and m', $\mathcal{A}(H, \lambda) \to (m, m')$[5]. 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,

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

Boneh-Lynn-Shacham (BLS) signature scheme [22]. 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 H_1: $\{0,1\}^* \to G_1$ 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 $e(\mathsf{H}_1(m), pk) \stackrel{?}{=} e(\sigma, g)$.

Appendix D. Full Security Analysis of BFTRAND Protocol

D.1. Proof of MEV Resilience

D.1.1. Proof of Pseudorandomness and Uniqueness.

Lemma 5.1. *If PBFT is (f, t, n)-secure and DRB is (f, k, n)-secure for any f with $f < t < k \leq 2t + 1$, then* BFTRAND *satisfies pseudorandomness and uniqueness.*

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

Proof. Pseudorandomness. Given a PPT attacker $\mathcal{A}$ that breaks the pseudorandomness experiment $\text{PRand}_{\text{BFTRAND},\mathcal{A}}$ of BFTRAND with non-negligible probability, we shall construct a PPT attacker $\mathcal{B}$ that breaks the pseudorandomness of *DRB* using $\mathcal{A}$ as a subroutine.

$\mathcal{A}$ chooses to corrupt a collection $\mathcal{C}$ of nodes in $\mathcal{N}$ with $|\mathcal{C}| \leq f$. $\mathcal{B}$ forwards $\mathcal{C}$ to its *DRB* who will start the interactive Setup protocol. The *DRB* runs on behalf of the honest nodes in $\mathcal{N}$, and $\mathcal{A}$ runs on behalf of the corrupted nodes in $\mathcal{C}$. $\mathcal{B}$ acts as a message forwarder between the *DRB* and the attacker $\mathcal{A}$ in the Setup protocol. At the end of the protocol, $\mathcal{B}$ can learn the public keys $\mathcal{PK}$. $\mathcal{B}$ obtains the beacon and partial beacon by by querying $\mathcal{A}$'s queries.

In the challenge query of $\text{PRand}_{\text{BFTRAND},\mathcal{A}}$, $\mathcal{B}$ receives set of partial beacons $\mathcal{E} = \{(i, \sigma_{b,v,i})\}_{i \in \mathcal{I} \cap \mathcal{C}}$, round b, and view v from $\mathcal{A}$. $\mathcal{B}$ answers the challenge query as below:

- Let $\mathsf{st} = x^*$ and $rn = (b, v)$. $\mathcal{B}$ runs the Comb protocol of *DRB* with input $(x^*, \mathcal{E})$ and receives a beacon value $\sigma_{b,v}$.
- $\mathcal{B}$ then uses *UpdState* protocol to update the state and get a new round number from $\mathcal{A}$.

Since the state for the $(b, v) - th$ round is of the form

$$x^* \leftarrow \begin{cases} \sigma_{b-1} \parallel b - 1, & \text{if } v = 0 \\ \mathsf{st}_{b,v-1} \parallel b \parallel v - 1, & \text{otherwise} \end{cases}$$

which is an unique value and has never been used before. After the challenge round, the state is updated to a new value either $\mathsf{st} = \sigma_b \parallel b$ or $\mathsf{st} = \mathsf{st}_{b,v} \parallel b \parallel v$ which is also a unique value and never repeats. Therefore, the attacker $\mathcal{A}$ does not query partial beacon query on state x^* before or after the round (b, v). Within the round (b, v), $\mathcal{A}$ is allowed to query partial beacon query for any times for distinct i. Finally $\mathcal{A}$ outputs b' and $\mathcal{B}$ outputs b'. This concludes the proof of pseudorandomness.

Uniqueness. Then, we show uniqueness of BFTRAND. The proof relies on the uniqueness property of *DRB*. Suppose there exists an attacker $\mathcal{A}$ leading the challenger of uniqueness experiment $\text{URand}_{\text{BFTRAND},\mathcal{A}}$ to output 1 with non-negligible probability. We will show that this leads to a contradiction. If the challenger of $\text{URand}_{\text{BFTRAND},\mathcal{A}}$ outputs 1, then $\mathcal{I} \neq \mathcal{I}'$, with $\mathcal{I}$ (resp.$\mathcal{I}'$) $\subseteq \mathcal{N}$ and $|\mathcal{I}|$ (resp. $|\mathcal{I}'|$) $\geq 2t + 1$, $\sigma_{b,v} \leftarrow \text{Finalize}(\{\sigma^i_{b,v}\}_{N_i \in \mathcal{I}}, b, v)$ and $\sigma'_{b,v} \leftarrow \text{Finalize}(\{\sigma^j_{b,v})\}_{N_j \in \mathcal{I}'}, b, v)$, but $\sigma_{b,v} \neq \sigma'_{b,v}$. The Finalize algorithm checks whether there exists a beacon that appears over t times. Since $\sigma_{b,v} \neq \sigma'_{b,v}$, then there exists two beacons that appears over t times in $\mathcal{I} \cup \mathcal{I}'$, due to the uniqueness of *DRB*, honest nodes in $\mathcal{N}$ will always generate the same beacon, therefore, there should be over t node that is corrupted to generate another beacon. Thus the size of corrupted node is $\mathcal{C} > t$, which is contradicts to the hypothesis. This concludes the proof of uniqueness. □

D.1.2. Proof of Liveness.

Lemma 5.2. BFTRAND *satisfies liveness if PBFT is* (f, t, n)*-secure and DRB satisfies uniqueness.*

Proof. For liveness, suppose there is a PPT attacker $\mathcal{A}$ that breaks the BFTRAND liveness experiment $\text{LRand}_{\text{BFTRAND},\mathcal{A}}$ with nonnegligible probability. We shall construct a PPT attacker $\mathcal{B}$ which breaks the *PBFT* liveness and *DRB* uniqueness. $\mathcal{A}$ chooses to corrupt a set of nodes $\mathcal{C}$ with $\mathcal{C} \subseteq \mathcal{N}$ and $|\mathcal{C}| \leq t$. $\mathcal{B}$ forwards $\mathcal{C}$ to its BFTRAND challenger to run the interactive Setup algorithm. $\mathcal{B}$'s BFTRAND challenger runs on behalf of the honest nodes in $\mathcal{N}$ and $\mathcal{A}$ runs on behalf of the corrupted nodes $\mathcal{C}$. $\mathcal{B}$ acts as a message forwarder between $\mathcal{B}$'s challenger and $\mathcal{A}$. After that, $\mathcal{A}$ can continue to query the oracles. Any query by $\mathcal{A}$ can be answered by $\mathcal{B}$'s beacon query and partial beacon query. $\mathcal{A}$ outputs $(\mathcal{I}, \mathcal{E} = (i, \sigma_i)_{i \in \mathcal{I} \cap \mathcal{C}})$. Let the current state be $st = x^*$. $\mathcal{B}$ issues a query $(x^*, \mathcal{I}, \mathcal{E})$ to its BFTRAND challenger. From the definitions of uniqueness of DRB and BFTRAND, we can easily see that when $\mathcal{A}$'s challenger outputs 1, $\mathcal{B}$'s challenger will also output 1. This concludes the proof. □

Theorem 5.1 (MEV Resilience)**.** *Assume underlying DRB is* (f, k, n)*-secure under definition 4, PBFT is* (f, t, n)*-secure, and* BFTRAND *achieves pseudorandomness, uniqueness, and liveness,* BFTRAND *achieves MEV Resilience in Def. 2.*

Proof. This can be guaranteed from Lemma 5.1 and Lemma 5.2. □

D.2. Proof of MR Resilience

Theorem 5.2 (MR Resilience)**.** BFTRAND *achieves MR Resilience under Definition 3.*

Proof. For a smart contract $\mathcal{SC}$ that issues runtime random number requests and has Validator algorithm implemented. An $\mathcal{A}$ can perform MR on $\mathcal{SC}$ in three ways (Fallback MR is excluded as explained in Section 4.2.1):

Transaction script verification scheme: $\mathcal{A}$ verifies the state of $\mathcal{SC}'$s execution in the transaction script logic after calling $\mathcal{SC}$. It must add extra logic to the transaction script, which will be detected by Validator and causes the execution to fail. Due to $\mathcal{T}$'s inability to verify the execution result of $\mathcal{SC}$, $\mathcal{A}$ is not able to exploit $\mathcal{SC}$ with MR. Checking the script format ensures the attacker cannot know the user contract execution result after user contract execution. The format check can be very complex to make the transaction more flexible and extendable, for example, byte by byte checking or regular expression checking. However, again, our work focuses on the security of runtime RNP. Therefore, we only provide a straightforward but efficient mitigation solution, which is the length check.

Invoking method verification scheme: Attacker $\mathcal{A}$ deploys an attacker contract $\mathcal{SC}'$ with the intent of launching a Contract MR on $\mathcal{SC}$ from $\mathcal{SC}'$. For an attacker transaction $\mathcal{T}$ from $\mathcal{A}$ that calls $\mathcal{SC}'$, the execution of $\mathcal{T}$ will fail. This is because when $\mathcal{SC}'$ calls $\mathcal{SC}$, $\mathcal{SC}$ checks whether it is the first smart contract triggered in Validator. $\mathcal{A}$ controls at most t Byzantine nodes and cannot change the smart contract execution logic. Therefore, $\mathcal{SC}$ can directly revert the transaction without calling any random number. It could

contain blacklist contracts, whitelist contracts, group contracts, permission management, access control, and various other methods to prevent this malicious reversal. We only provide one of the efficient solution here.

Transaction fee verification scheme: Attacker $\mathcal{A}$ learns the execution fee cost of all paths of the user contract $\mathcal{SC}$. Then specifically set the fee of attacker transaction $\mathcal{T}$ that is only sufficient to execute the profitable path to perform Fee MR. Then, the contract $\mathcal{SC}$ that has Validator function embedded can detect this behavior since $\mathcal{SC}$ has a minimum transaction fee requirement for $\mathcal{T}$, if the fee is not sufficient to cover all paths, $\mathcal{SC}$ could revert the execution without calling the random number. Fee MR is a side-channel MR, it dose not interact with $\mathcal{SC}$ directly; hence, setting a minimum transaction fee requirement would be the best option, unless all paths are purposely programmed to have the same transaction fee, which is exceedingly difficult.

Fallback prevention scheme: A Fallback attack requires the attacker $\mathcal{A}$ to somehow induce the user contract $\mathcal{SC}$ to interact with the attacker contract $\mathcal{SC}'$ after executing critical operations, hence activating the fallback function of $\mathcal{SC}'$. However, with $\mathcal{SC}$ calling Validator from the randomness provision library to check whether the address of the interacting address is a contract account, $\mathcal{SC}$ shall never interact with $\mathcal{SC}'$, thus the fallback function of $\mathcal{SC}'$ will never be activated by $\mathcal{SC}$.

With four of the identified malicious reversals mitigated, the proof of malicious reversal resilience is concluded. □

Appendix E. Lines of Code

We implemented a BFTRAND prototype in C# on neo v3.1. In summary, we added 383 lines of code to the neo-core to implement the PRF, 3,679 lines of code to the consensus module to implement the *DRB*, 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 BFTRAND. 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
RPS	C#	369	4
Neoverse	C#	596	8
Comparison	C#	100	2

Appendix F. Instruction for BFTRAND Integration

Developers can pick any eligible *DRB* protocol and PRF 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 below, where setup process is not included:

1. Implement cryptographic *DRB* and PRF libraries.
2. 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.
3. 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.
4. 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.
5. Extend the "Commit" message to contain the beacon. Since the beacon is generated via consensus, no need to contain the proof of the beacon.
6. 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.
7. Implement an interface for smart contract to provide random numbers.
8. Implement an MR defender library with routines to validate the transaction script format.