BFTRAND: Low-latency Random Number Provider for BFT Smart Contracts

Jinghui Liao*†§x, Borui Gong*†‡, Wenhai Sun¶, Fengwei Zhang*†✉,
Zhenyu Ning‖, Man Ho Au‡, and Weisong Shi**
*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
†Department of Computer Science and Engineering, Southern University of Science and Technology, China
‡Department of Computing, The Hong Kong Polytechnic University, China
§Computer Science, Wayne State University, USA
¶Computer and Information Technology, Purdue University, USA
‖Hunan University, China
**Computer and Information Sciences, University of Delaware, USA
xNeo Foundation
{liaojh2021, gongbr}@mail.sustech.edu.cn, whsun@purdue.edu
zhangfw@sustech.edu.cn, zning@hnu.edu.cn, mhaau@polyu.edu.hk, weisong@udel.edu

***Abstract*—Random numbers play a crucial role in decentralized applications (dApps) like decentralized finance (DeFi) and non-fungible tokens (NFTs). However, their generation faces challenges due to blolckchain's deterministic and decentralized nature, risking smart contract security and ecosystem stability. Prior solutions, including Oracles, employing commit-execute schemes, suffer from higher transaction fees, extended processing times, and increased on-chain storage, compromising efficiency.**

This paper proposes a novel random number provider (RNP) protocol for smart contracts, eliminating dependencies on traditional commit-execute approaches. Furthermore, we systematically identify potential random number-related attacks on smart contracts, particularly Post-reveal Undo Attacks (PUAs), where attackers may reverse contract operations when randomness is unfavorable, and discuss the security requirements. Our protocol addresses these attacks by (1) incorporating distributed random beacons (DRBs) with consensus processes, bridging the semantic gap between DRB and consensus, and (2) thoroughly analyzing and classifying four types of PUA and offering robust mitigations, alongside presenting a security proof.

Our experiments show the protocol significantly enhances response times and security for random number queries in smart contracts, slashing request fees by at least 89% and reducing on-chain data by 76.4% versus current methods. This work advances the integration of DRB protocols and consensus mechanisms, securing and optimizing random number applications in dApps, thus fostering the creation of more dependable, robust systems.

I. INTRODUCTION

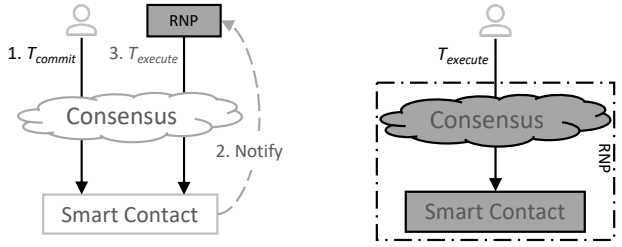
Decentralized applications (dApps), based on smart contracts, such as decentralized finance (DeFi) and non-fungible tokens (NFTs) [46], rely heavily on public random number providers (RNPs) for determining the winners, shuffle cards [46], and ensuring the uniqueness, rarity, and equitable distribution of NFTs [37], [76], [81]. However, generating secure random numbers within smart contracts faces challenges. Therefore, some legal toolkits use random jurors to facilitate efficient dispute resolution [72]. The difficulty lies in the deterministic and decentralized nature of blockchain systems, which exposes vulnerabilities to adversaries aiming to predict or bias randomness, leading to various attacks [41], [60], [90], [92]. For example, a dApp called EOSBet encountered a nonce attack in 2018, where attackers influenced the RNP to control game outcomes, resulting in token theft [42].

Therefore, secure RNP protocols are required to produce random values resistant to possible previous attacks [26]. Schemes like distributed random beacon (DRB) [2], [19], [31], [36], [43], [56], [62], [87], [88], [93], the verifiable delay function (VDF) [6], [51], and verifiable random function (VRF) [4], [35] offer promising solutions for generating secure random numbers. However, the integration of these generated numbers into smart contracts necessitates a commit-execute RNP mechanism, as illustrated in Figure 1a. Specifically, it requires recording user parameters in the transaction $\mathcal{T}_{\texttt{commit}}$ during a block interval and generating universally accessible randomness. Then a second transaction $\mathcal{T}_{\texttt{execute}}$ provides random numbers in the subsequent block time, therefore preventing attackers from predicting or bias the random numbers. Although this approach realizes a secure RNP, it results in high transaction fees, extended processing time, and increased on-chain storage. Moreover, existing schemes do not consider blockchain-specific security needs and operational limitations, such as the risk of post-reveal undo attacks (PUAs) (§IV), where attackers can intentionally revert the execution of $\mathcal{T}_{\texttt{execute}}$ if the result is not profitable. Taking these into account, current RNP approaches reveal their inadequacy for application in the context of smart contract execution, causing the need for a new approach.

BFTRAND Protocol: We present a new low-latency RNP protocol, called BFTRAND, which is specifically designed to *deliver random numbers to smart contracts within a single consensus round*, as illustrated in Figure 1b. BFTRAND addresses the challenges associated with existing commit-execute RNP schemes, such as Chainlink VRF [35] and Drand [45], while offering a secure and efficient approach to generate random numbers for smart contracts.

BFTRAND generates low-latency random numbers during

Jinghui Liao and Borui Gong contributed equally.
✉ Fengwei Zhang is the corresponding author.



(a) Commit-execute RNP (b) BFTRAND RNP

Figure 1: Comparison between commit-execute RNP and BFTRAND RNP. $\mathcal{T}_{\texttt{commit}}$ and $\mathcal{T}_{\texttt{execute}}$ represent transactions to commit function parameters and execute the function, respectively. Solid lines indicate one consensus round. Commit-execute RNP (a) requires two transactions in two distinct consensus rounds to supply the requested random number, while BFTRAND (b) fulfills one request within one consensus round. RNP components are shaded in gray

the execution of smart contracts, resulting in reduced response times for random number requests. More specifically, By incorporating DRBs into the consensus process, BFTRAND introduces a beacon for each block and quickly generates random numbers for smart contracts upon receiving requests. Furthermore, BFTRAND includes robust security measures to protect against possible random number attacks, including post-reveal undo attacks (PUA)(§IV).

Challenges: Our proposed protocol tackles two main challenges: seamlessly integrating DRB without compromising consensus efficiency and mitigating possible PUA risks.

Insufficiency of DRB. Although the direct request-response paradigm introduced by BFTRAND offers a substantial improvement in delivering randomness for dApps, integrating DRBs with blockchain consensus mechanisms reveals potential security mismatches. To address this problem, we make a thorough analysis to harmonize these two components. Specifically, DRBs inherently possess security properties such as pseudo-randomness and uniqueness [53], [57], [85]. However, when embedding DRBs into blockchain systems to provide random numbers for smart contracts, we encounter unique security and performance challenges stemming from divergent security premises and blockchain-specific constraints. A prime example is the inherent operational difference between DRBs and Byzantine Fault Tolerance (BFT), a foundational consensus framework. DRBs function through sequential stages, contrasting with the BFT's round-based consensus, where each round is further divided into multiple views under a common round context. Therefore, we derive careful management of views within BFT rounds to prevent adversaries from predicting random numbers across views (§V-A).

Post-reveal Undo Attack (PUA). The inherent characteristics of blockchain present unique challenges in developing a secure Random Number Provider (RNP) protocol for smart contracts. Our analysis reveals that including random numbers in smart contracts at runtime may expose a previously unexplored attack surface, termed the Post-reveal Undo Attack (PUA), which exploits blockchain transactions' atomic nature. Specifically, if a transaction in a blockchain is unsuccessful, its associated operations are undone [10]. An attacker could exploit this by intentionally failing smart contract transactions when outcomes are adverse, leading to wasted operations and potential system manipulation. This attack leverages the attacker's capacity to assess state changes before finalizing a malicious transaction. We categorize PUAs into four types: *Contract PUA*, *Fallback PUA*, *Fee PUA*, and *Script PUA* (§IV).

To address these problems, we define *random number security* by identifying essential security criteria our protocol must meet when integrating DRBs into the consensus process. These criteria are *pseudorandomness*, *uniqueness*, *availability*, and *irreversibility*. The *pseudorandomness* evaluates the ability to generate random numbers that are unpredictable and unbiased. The *uniqueness* ensures that each random number generated is deterministically unique and cannot be biased. The *availability* guarantees that the RNP can respond consistently to requests for random numbers without allowing attackers to disrupt the consensus process. Lastly, irreversibility ensures that outcomes of contract functions using random numbers, once recorded on the blockchain, are final and non-reversible.

By addressing these aspects and carefully designing the integration, our protocol aims to provide secure random numbers while maintaining the performance of the blockchain consensus without additional consensus phases (§III-C).

Implementation and Evaluation: To assess BFTRAND against existing commit-execute RNPs, we instantiate a prototype relying on the DRB scheme proposed in [53], [57]. Additionally, we implement $Instance_{commit-execute}$ as a baseline to simulate the behavior of commit-execute RNPs. Our results demonstrate that BFTRAND reduces on-chain data by 76.4% and decreases the cost of fulfilling a random number request by at least 89%, while introducing a minimal 0.002% increase in consensus time (assuming a 15-second block time). To the best of our knowledge, our protocol is the first secure and low-latency RNP protocol specifically designed for smart contracts.

Contributions: Our contributions are summarized as follows:

1) We introduce a novel RNP protocol, BFTRAND, which securely integrates distributed random beacons into the consensus, enabling efficient and secure provisioning of random numbers for decentralized applications.
2) We provide the first comprehensive and systematic analysis of random number security in smart contracts and rigorously define the concept of *random number security*. This definition includes the requirements for *pseudo-randomness*, *uniqueness*, *availability*, and *irreversibility*, which must be fulfilled by RNP protocols.
3) We identify a new risk on random numbers, named *Post-reveal Undo Attack*, and propose the security property of *irreversibility* for RNPs to address this problem. This property ensures the security and integrity of RNPs in smart contracts by preventing such attacks.

4) We prove the security of BFTRAND under the definition of *Random Number Security*.
5) We evaluate the performance of BFTRAND, illustrating its efficiency and scalability. Our evaluation shows that it outperforms existing commit-execute RNP schemes in terms of transaction throughput and computational overhead while maintaining a high level of security.

II. BACKGROUND

This work focuses on a blockchain system based on Practical Byzantine Fault Tolerant (PBFT) with smart contracts consisting of n nodes, $\mathcal{N} = N_0, \cdots, N_{n-1}$. The set of corrupted nodes controlled by the adversary $\mathcal{A}$, is represented as $\mathcal{C}$. Let $D, R : \mathbb{N} \to \mathbb{N}$ denote polynomial-time functions, with both $D(\lambda)$ and $R(\lambda)$ bounded by a polynomial in λ. Furthermore, let $F : \mathtt{Dom} \to \mathtt{Ran}$ be a function with domain $\mathtt{Dom}$ and range $\mathtt{Ran}$, where $\mathtt{Dom}$ and $\mathtt{Ran}$ are sets with sizes $2^{D(\lambda)}$ and $2^{R(\lambda)}$, respectively. We also assume the existence of a pseudorandom function (denoted as *PRF*).

A. Blockchain

This section provides a brief overview of fundamental concepts and technologies relevant to our work.

1) Smart Contracts: Smart contracts are self-executing agreements with terms encoded in the program. They are used primarily in decentralized applications (dApps) built on platforms such as Ethereum [29] and NEO [75]. These contracts automate complex business logic and enforce contractual terms without intermediaries. They are immutable and transparent, ensuring a high level of trust and security.

2) Byzantine Fault Tolerance: Byzantine Fault Tolerance (BFT) represents a class of consensus algorithms, including PBFT [33] and its variants [3], [8], [12], [15]–[17], [30], [32], [38], [63], [66], [68], [71], [75], [95], [96], [98], which address the Byzantine General Problem in distributed systems and provide instant finality [67]. Currently, 17 of the top 80 blockchain projects use BFT consensus [39], including Theta [77], Neo [75], Algorand [5], and EOS [47].

BFT consensus enables distributed systems to achieve consensus despite faulty nodes exhibiting unpredictable behavior. It operates in rounds, with each round having one or more views [33]. At the end of each round, a new block is formed that represents the consensus of the network. An (f, t, n) secure BFT protocol with n nodes can tolerate up to t faulty nodes and is secure for any probabilistic polynomial-time (PPT) attacker $\mathcal{A}$ if $0 \leq f \leq t < n/3$ for asynchronous BFT [33]. A BFT consensus protocol adheres to consistency and liveness properties [32].

For clarity, we will now provide a concise explanation of PBFT [33], which is a commonly used variant of BFT. PBFT operates through consensus rounds, with each round led by a designated leader. The leaders are chosen in a round-robin fashion. Specifically, the function **Leader**$(b, v, n) \to l$ is defined to select the leader index l for round b of view v, where n represents the number of participants. In other words, given the values of view v, round b and the total number of participants n, the function **Leader** outputs the leader index l. The PBFT algorithm consists of the following phases [33]: *Prepare.* The leader receives user requests and transactions and starts the consensus process by sending a "prepare" message to the consensus nodes that are not the leader (replicas). *Response.* Upon receiving the "prepare" message, a node sends a "respond" message to all other nodes, including the leader. *Commit.* A consensus node sends a "commit" message after collecting more than $2t+1$ "respond" messages. *Finalize.* Once a node has obtained over $2t+1$ "commit" messages, consensus is achieved and the node provides the execution result to users. **Viewchange:** This mechanism is a critical aspect of the PBFT algorithm to ensure fault tolerance, which is triggered by a "viewchange" message. Specifically, if a node encounters a timeout during any of the aforementioned phases, it constructs and broadcasts a "viewchange" message to request for a fresh view. If more than $2t$ view change messages are received, a new leader will restart the entire process with a new view.

B. Distributed Random Beacons

Distributed Random Beacons (DRBs) are a well-established class to generate random numbers, which are called beacons within the DRB context. This algorithm ensures that no single entity can manipulate or predict the generated values. Various DRB protocols have been proposed, including Drand [45], DFINITY [2], RandChain [56], RandPiper [19], DDH-DVRF and GLOW-DVRF [53], RandHound and RandHerd [93]. The feasibility of constructing a DRB protocol has been demonstrated in [53], [57], [85]. Looking ahead, we will utilize the *DRB* scheme proposed in [53] in our instantiation, and we provide a formal definition of DRB as follows.

1) ***Setup***$(1^\lambda, k, n)$. The nodes in $\mathcal{N}$ execute this interactive protocol to establish a random beacon committee, where λ is a security parameter. The protocol produces a global public key pk, a list of public keys $\{pk_0, \cdots, pk_{n-1}\}$, and a list of secret keys $\{sk_0, \cdots, sk_{n-1}\}$.
2) ***Partial***$(\mathsf{st}_{rn-1}, sk_i, pk)$. Given a state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$ of round $rn-1$, a secret key sk_i, and a global public key pk, the algorithm computes a partial beacon σ_i and a proof π_i. The output is (i, σ_i, π_i).
3) ***Comb***$(\mathsf{st}_{rn-1}, \mathcal{E}, pk)$. Given a state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$, a set $\mathcal{E} = \{\sigma_i, \pi_i\}_{i \in I}$ of partial values and proofs from $|I| \geq k$ different nodes and a global public key pk, this algorithm produces a beacon σ^{rn} and its proof $\pi^r n$, or $\perp$.
4) ***Verify***$(\mathsf{st}_{rn-1}, \sigma^{rn}, \pi^{rn}, pk)$. Given an input state $\mathsf{st}_{rn-1} \in \mathtt{Dom}$, a pair consisting of a beacon σ^{rn} and proof π^{rn}, and a public key pk, this algorithm verifies the validity of σ^{rn}. It outputs 0 if the beacon is valid and 1 otherwise.
5) ***UpdateState***$(\mathsf{st}_{rn-1}, \sigma^{rn}, \pi^{rn}, pk)$. Given the current state st_{rn-1}, a beacon $\sigma^{rn} \in \mathtt{Ran}$ and its proof π^{rn} generated at the end of the round $rn-1$ (σ^{rn} and π^{rn} can be $\perp$), and a global public key pk, the algorithm outputs the updated state st_{rn} for the round rn, or $\perp$.

Security Requirements. Pseudorandomness: ensures that the generated beacon cannot be distinguished from a uniformly

chosen value in the presence of active adversaries, denoting unpredictability and bias-resistance; *Uniqueness*: requires the ***UpdateState*** function to generate a unique and non-repeating state for each round rn, and the generated random beacon is deterministically unique with the fixed input, that is, the output will always be the same with the same input and state, even if the attacker can access the secret keys of honest parties.

Furthermore, we would like to clarify that the uniqueness property guarantees that no matter which set of partial values is used in *Comb*, as long as the valid partial beacon exceeds the threshold, they will all result in the same beacon σ.

III. Problem Formulation

In this section, we provide a comprehensive overview of our BFTRAND protocol (§III-A) and examine the security assumptions (§III-B). Subsequently, we briefly explain the required security properties (§III-C). In the full version [9], we provide an explanation of random number attacks.

A. Overview of BFTRAND

BFTRAND aims to provide a low-latency secure random number service for smart contracts. It generates a random beacon for each block and derives multiple random numbers for smart contracts from the generated beacon.

In BFTRAND, a *proposal* refers to a candidate block that contains a list of transactions. This proposal will be confirmed at the end of the consensus process if more than $2t$ nodes agree. For the consensus round $b > 0$ in view $v \geq 0$, the proposal is denoted as $\mathcal{L}_{b,v}$. To simplify our notation, we use b to represent both the consensus round and the corresponding block. BFTRAND is divided into three main sections: Initialization, Beacon Request, and Randomness Request, each describing the necessary steps to achieve the desired outcomes.

1) Initialization: Before executing BFTRAND with DRB, it is crucial to initialize the context of BFTRAND to establish the execution environment. During this phase, the DRB.Setup function generates public and private keys for participants. Additionally, an empty dictionary of random beacons, denoted by $RB = \{\}$, is initialized to store random beacons generated indexed by unique identifiers.

2) Beacon Request: BFTRAND generates a random beacon for each block after the point at which new transactions cannot be created or the block proposal cannot be altered. This beacon acts as a random seed to produce random numbers within the corresponding block. A beacon request begins when the consensus leader sends $(\texttt{Beacon}, b, v)$ to others, where b and v represent the block and view numbers respectively. Then a (deterministic) random beacon, rb, is generated by DRB after BFT's commit phrase. The beacon is then stored in the dictionary with the index b as $RB[b] = rb$.

3) Randomness Request: A smart contract requests randomness by sending a tuple $(\texttt{Randomness}, b, m, \mathcal{T})$. The protocol first retrieves the corresponding random beacon from the dictionary using the index b. Then m random numbers, $(r_1, \ldots, r_m)$, are calculated and returned to the smart contract.

B. Security Assumptions and Thread Model

This section presents the security assumptions and threat model of our protocol. We consider a static attacker $\mathcal{A}$, who has control over up to $t < n/3$ nodes in the blockchain network, and the nodes under control are determined before the consensus protocol is initiated. Furthermore, these controlled nodes may exhibit Byzantine behavior and the attacker has complete control over the faulty nodes, including their network connections, operating systems, and their secret keys. The attacker is also able to observe the blockchain network, including all transactions and consensus messages. If the leader node is corrupted, the attacker can propose a transaction list that benefits itself. We assume that the un-corrupted nodes have enough storage capacity to process blockchain transactions.

C. Security Properties

The random number security analysis can be found in the full version [9]. To ensure the security of our RNP protocol can effectively counteract random number attacks on smart contracts, we have established a set of properties to which BFTRAND must adhere. Let Π be the RNP protocol, $\mathcal{A}$ be an adversary, and $\mathcal{U}$ be a uniform random function. Let $\mathcal{D}$ be a distinguisher that interacts with $\mathcal{A}$ and a protocol/function, then returns 1, if it believes that it interacted with Π, or 0.

1) Pseudorandomness: Pseudo-randomness requires that generated random numbers are indistinguishable from truly random numbers within practical time and computation resource constraints. It guarantees that an attacker cannot predict random numbers and launch pre-computation attacks.

2) Uniqueness: Uniqueness stipulates that only one valid and deterministic random number can be generated for any given input. Enforcing uniqueness prevents potential adversaries from manipulating multiple valid random numbers for identical inputs or exploiting race conditions. Consequently, uniqueness effectively mitigates replay attacks and obstructs validator collusion attacks, thus enhancing RNP security.

3) Availability: Availability ensures the continuous generation of random numbers by the RNP for each smart contract request. Protect against potential DoS attacks or other disruptions that aim to compromise the functionality of the blockchain system and its applications, enhancing the overall security and reliability of the system.

4) Irreversibility: The property of *irreversibility* ensures that the execution of a transaction does not fail unintentionally. This security prerequisite significantly reduces the likelihood that adversaries exploit PUAs to compromise "critical operations" and reverse the transaction outcome. Consequently, it provides a solid foundation for the secure execution of smart contracts that require random numbers.

IV. Post-reveal Undo Attack

In this section, we discuss PUA in further detail. We identify four classes of PUAs: **Contract PUAs**, **Fallback PUAs**, **Fee PUAs** (which arise from smart contract designs), and **Script PUAs** (specific to script-based blockchains). To demonstrate

```
1  function MintNFT(from, target, amount)
2      CheckWitness(from)
3      Require(amount == 1)
4  ❶   Transfer(from, this, amount)
5      rarity = GetRandom(1) % 2
6      if (rarity == 1){
7          // Cost 0.5 GAS
8          blindBox = RareBlindBox(target, rarity)
9  ❷   }else{
10         // Cost 0.6 GAS
11         blindBox = CommonBlindBox(target, rarity)
12     }
13     Mint(target, blindBox)
14 ❸   Transfer(this, target, 1, {blindBox})
15     StoreBlindBox(blindBox, rarity)
16     return blindBox
17 end function
18
19 function GetRarity(blindBox)
20     return StoreBlindBox[blindBox]
21 end function
```

Figure 2: Pseudocode representation of the two core functions within the vulnerable BlindBox, the victim contract, susceptible to PUA attacks. ❶ The"MintNFT" function accepts parameters (from, target, amount), where 'from' denotes the user address, 'target' represents the address receiving the NFT token, and 'amount' specifies the number of tokens transferred."MintNFT" processes a transfer of one token from a user, ❷ employs random numbers to ascertain the rarity of the newly minted NFT, and ❸ transfers the NFT to the intended recipient. The "GetRarity" function retrieves the rarity of a specific NFT. Note that the GAS in the code is a dummy token utilized solely for contract demonstration, and it is also used as the unit of transaction fee.

how these PUAs exploit vulnerable contracts, we create a representative BlindBox contract as an illustrative example.

A. A Victim Contract - BlindBox Contract.

Figure 2 presents the pseudocode of a BlindBox contract designed to demonstrate the effects of various PUA approaches. This contract allows users to deposit a token, mint a corresponding NFT, and transfer it to a specified recipient. The rarity of the NFT depends on a random number acquired during run-time through the "GetRandom" function. The "Critical Operation" in the code block ❶ represents a user who submits a one token bid for an NFT. The random number in ❷ determines the rarity of the minted NFT, while ❸ transfers the minted NFT to the designated address. Adversaries exploit this contract by reversing the deposit operation in ❶ if the minted blind box does not have a high rarity. This action contradicts the nonrefundable and nonexchangeable principles inherent in the blind box mechanism.

B. Contract PUA.

The functionality of smart contracts is significantly enhanced by their interoperability, which allows a single transaction to invoke multiple smart contracts through interoperation. Transactions are atomic, meaning that if any of the contracts involved in a transaction fails and reverts, all other executed contract calls will also be rolled back [97]. Attackers can exploit this interoperability by deploying a malicious contract

```
1  function ContractPUA(user)
2  ❹   blindBox = call NFT.MintNFT with (user, 1)
3      rarity = call NFT.GetRarity with (blindBox)
4      if rarity == 0 then
5  ❺       call revert();
6      end
7      return true
8  end function
```

Figure 3: Pseudocode representation of the Contract PUA malicious contract. NFT represents the BlindBox contract.

beforehand and invoking the victim contract within the malicious contract. The malicious contract then examines the results of the execution of the victim contract [14], [70]. Figure 3 presents the pseudocode of the malicious contract, which includes a method named "ContractPUA" that initiates the PUA. In line 2, the "MintNFT" method of the victim contract is called, and in line 3, the rarity of the minted NFT token is retrieved and checked in the code block ❺. The attacker invokes the victim contract in ❹, calling the "MintNFT" method of the BlindBox contract and minting an NFT with a random rarity. The attack contract then verifies the result against their expectations in ❺. If the minted NFT token is not considered rare, the attacker can deliberately reverse execution, undoing Critical Operation (❶ in Figure 2). Contract PUA differs from the pre-computation attack, as it does not compute random numbers.

C. Fallback PUA.

The fallback function is a default mechanism in smart contracts that is executed when receiving a transaction without a specified function call or if the requested function does not exist [14]. This feature allows contracts to handle unexpected scenarios or implement custom logic for incoming transactions [70]. Attackers may exploit the fallback function to target victim contracts in the following way. By creating a malicious contract with a strategically designed fallback function, the attacker can influence the behavior of the victim contract during interactions with the malicious contract [55]. Consequently, the attacker can utilize the fallback function to verify the execution result of the target contract and potentially alter or reverse the intended outcome of the contract. This attack is known as a fallback PUA. Figure 4 illustrates a fallback PUA contract. If the "target" address in code block ❸ of Figure 2 matches the address of the malicious contract, the malicious contract is invoked when executing❸. The malicious contract can then assess the rarity of NFT in ❻ [64]. By exploiting the fallback function mechanism, attackers can launch attacks that interfere with the standard operation of the victim contract and compromise its security [80].

D. Fee-based PUA.

Transaction fees play a crucial role in incentivizing validators or miners to maintain the blockchain network. Attackers who cannot directly verify state changes may exploit fee-based side channels to execute PUAs without interacting with the targeted contracts. This strategy is called a fee-based

```
function Fallback(from, amount, object)
    rarity = call NFT.GetRarity with (blindBox)
❻   if rarity == 0 then
        call revert();
    end
    return true
end function
```

Figure 4: Pseudocode representation of a Fallback PUA contract. NFT represents the vulnerable BlindBox contract.

```
function MintNFT(from, target, amount)
    CheckWitness(from)
    Require(amount == 1)
    Transfer(from, this, amount)
    rarity = GetRandom(1) % 2
    if (rarity == 1){
❼       // Cost 0.5 GAS
        blindBox = RareBlindBox(target, rarity)
    }else{
        // Cost 0.6 GAS
❽       // revert if fee insufficient
        blindBox = CommonBlindBox(target, rarity)
    }
    Mint(target, blindBox)
    Transfer(this, target, 1, {blindBox})
    StoreBlindBox(blindBox, rarity)
    return blindBox
end function
```

Figure 5: Pseudocode representation of a Fee PUA.The code block in ❼ consumes an additional 0.5 GAS, while ❽ consumes an additional 0.6 GAS. ❼ and ❽ are marked as ❷ in Figure 2.

PUA. Figure 5 illustrates an example of fee-based PUA. As mentioned above, the "MintNFT" function is a function of the BlindBox contract. Line 6 of the code examines the rarity, which is determined by a random number. If the rarity is 1, the code proceeds to ❼, creating a rare blind box and incurring a transaction fee of 0.5 GAS. Otherwise, it moves to ❽, generating a common blind box at the cost of 0.6 GAS. If the attacker is aware of the gas costs for both paths and intends to revert Critical Operation when ❼ is selected, they can set the maximum transaction fee to a higher value, which is sufficient for executing ❼, but lower than the cost of ❽. Identifying fee-based PUAs is substantially more challenging compared to other PUA categories.

These three prevalent PUAs mentioned above are prevalent in many smart contract platforms and fundamentally arise from the core design principles of blockchain and smart contracts [14], [80], [97]. These principles comprise interoperability, fallback mechanisms, and gas fees, which are crucial components of the smart contract ecosystem [80]. Apart from these generalized PUAs, a distinct PUA is identified, specifically unique to script-based smart contract platforms [40].

E. Script-based PUA.

In script platforms, transactions are represented as scripts and executed within a virtual machine (e.g., Neo Virtual Machine [75]). Attackers can exploit this feature by adding additional checking logic in the malicious transaction script after invoking a victim contract, to verify and manipulate

```
load "Blindbox Contract"
push amount
push target
push from
call "MintNFT" // push blindBox to the stack
call "GetRarity" //pop blindBox, push rarity
jmpz revert() // if rarity == 0, jump to revert() ❾
```

Figure 6: Transaction script pseudocode representation of a Script PUA. The code block ❾ is the attached verification logic.

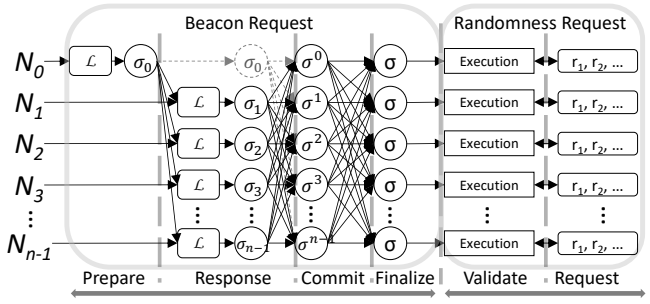


Figure 7: High-level Π_{BFTRAND} description for beacon request and for randomness request. $\mathcal{L}$ is the *proposal* from the leader for some view v of round b. $\{\sigma_i\}$ are the partial beacons, whereas $\{\sigma^i\}$ are the beacons aggregated by nodes, and σ is the final beacon. $\{r_i\}$ are random numbers. Execution entails running smart contracts with transaction inputs on a virtual machine. We omit the corresponding subscripts relating to the view v and round b in $\mathcal{L}$, σ_i, σ^i and σ.

the execution results. Figure 6 presents the pseudocode for a malicious transaction script. During the execution process, the virtual machine (VM) first loads and runs the victim contract (*line 1*), pushes the required parameters to the stack, and subsequently calls the "MintNFT" method (*line 5*). This method retrieves the parameters from the stack, processes them, and pushes the resulting output back onto the stack. After the "MintNFT" method has concluded, the script invokes "GetRarity" (*line 6*), which accesses the NFT token from the stack and pushes its rarity value. The script then examines the rarity and if it is deemed unsatisfactory, the attacker can intentionally reverse the transaction [61]. This attack exemplifies the security concerns associated with script-based smart contract platforms and emphasizes the importance of implementing robust security measures while developing and deploying smart contracts on these platforms.

V. BFTRand Protocol Π_{BFTRAND}

We introduce Π_{BFTRAND}, an instantiation of BFTRand, specifically crafted for BFT-based consensuses. It can also be adapted for various BFT-style consensus blockchain systems. As illustrated in Figure 7, it functions in two primary stages: the beacon request phase, responsible for generating the block beacon, and the randomness request phase, providing random numbers for smart contracts.

We assume the existence of an efficient pseudo-random function *PRF*. Let *PRF* be an efficient keyed function family,

PRF : $\{\mathcal{K} \times \mathcal{X} \to \mathcal{Y}\}$, where $\mathcal{K}, \mathcal{X}$ and $\mathcal{Y}$ are indexed with a security parameter λ. For a function F_k in *PRF* where its key k is uniformly chosen from $\mathcal{K}$, the following holds for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$,

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

where $f(\cdot)$ is a function uniformly chosen from the functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. Looking ahead, we denote $\mathcal{X}$ as $\mathtt{Ran}$, while $\mathcal{Y}$ as $\mathtt{Ran}'$ in our construction.

A. Beacon Request

We adopt the *DRB* scheme from [53] in our beacon request protocol. Although integrating the DRB directly into the consensus process to generate a beacon for each block appears straightforward, our analysis identifies a significant semantic gap between DRB and consensus. Without careful design, the system would be vulnerable to random number attacks, potentially compromising the security properties of both DRB and consensus and degrading consensus performance. The primary challenges identified are as follows.

First, the beacon must be produced at precisely controlled times to avoid exploitation. Second, the threshold of DRB differs from the Byzantine tolerance of the consensus and has distinct security assumptions. Improper initialization could result in validator collusion attacks or DoS attacks. Third, the DRB and consensus operate in separate rounds, and their direct integration could lead to precomputation or replay attacks.

1) Semantic Gap between DRB and Consensus: BFTRAND is designed upon *PBFT*, and it requires seamless integration with *DRB*. To achieve this, the generation of beacons from *DRB* must be completed before the Finalize phase of *PBFT* to prevent any disruption of the consensus process. Additionally, the aggregated beacon should be broadcast along with the commit message. During the Prepare phase, the *PBFT* leader must also utilize *DRB* to generate the partial beacon and broadcast it with the message "prepare". The other consensus nodes generate their partial beacons during the Response phase and transmit them with the response message. Failure to follow these phases will introduce additional consensus phases, compromising the overall performance of the consensus process.

2) Semantic Gap between k and t: The *DRB* generates a random beacon by aggregating no less than k authenticated partial beacons. To align threshold k in *DRB* with the Byzantine fault tolerance assumption of *PBFT*, we need to carefully choose the range of k. First, k must exceed the maximum *PBFT* Byzantine tolerance t, otherwise, attacker $\mathcal{A}$ could prematurely aggregate the beacon, leading to pre-computation attacks. Furthermore, k must not exceed $2t + 1$, otherwise, *DRB.Comb* would require the cooperation of Byzantine nodes and potentially cause DoS attacks.

3) Semantic Gap between DRB Round and Consensus Round: *DRB* operates in rounds rn, producing a random beacon for each. Therefore the **UpdateState** of DRB (Section II-B) is also related to rn and runs every time a new round starts to get a new state. Meanwhile, *PBFT* operates on consensus round b. RNP requires only one random beacon per consensus round. A straightforward approach might assign the consensus round b as the *DRB* round rn. However, such integration introduces potential security risks due to the multi-view nature of PBFT, and causing different views in the same consensus round having the same input parameters for **UpdateState** and get the same states.

Specifically, *PBFT* will switch to a new view in the same consensus round if it fails in the current view. Therefore, if one sets rn as b, then view v has state: $\mathsf{st}_b^{b,v} = \textbf{UpdateState}(\mathsf{st}_{b-1}, \sigma^b, \pi^b, pk)$, view $v{+}1$ has state: $\mathsf{st}_b^{b,v+1} = \textbf{UpdateState}(\mathsf{st}_{b-1}, \sigma^b, \pi^b, pk)$. Since the parameters of **UpdateState** for both view v and view $v + 1$ are exactly the same, we have the same state for different views, that is, $\mathsf{st}_b^{b,v} = \mathsf{st}_b^{b,v+1}$, resulting in beacon leakage and replay attacks. Therefore, if the prior consensus changes the view at or after the *PBFT* Commit phase, the attacker $\mathcal{A}$ can compute the random beacon before the new view begins.

Adapt UpdateState for BFTRAND: Because of the potential vulnerability caused by the semantic gap between DRB round and consensus round, we modify the definition of **UpdateState** to make it related to both consensus round b and view v. Specifically, instead of directly assigning rn as b, we combine b and v to generate a unique rn. In the meanwhile, the adapted **UpdateState** should follow the same security requirement of state uniqueness as defined in Section II-B. This is achieved by concatenating b and v or using a function that maps the tuple (b, v) to a unique integer value. Consequently, the adapted **UpdateState** for the (b, v)-th round is of the form: $\textbf{UpdateState}(\mathsf{st}_{b-1}, \perp, \perp, pk) : \mathsf{st} \leftarrow \sigma_{b-1} \parallel b - 1$ when $v = 0$, and $\textbf{UpdateState}(\mathsf{st}_{b,v-1}, \perp, \perp, pk) : \mathsf{st} \leftarrow \mathsf{st}_{b,v-1} \parallel b \parallel v$, otherwise. And st_{b-1} is the state in the consensus round $b-1$ used to generate the random beacon, while $\mathsf{st}_{b,v-1}$ is the state of view $v - 1$ in the same round b. Since there is only one $\{b, v\}$ pair for each view, state uniqueness can be fulfilled.

We would like to highlight that **UpdateState** will be run at the beginning of a consensus round or after a view change. A failed view v in the PBFT consensus round b will result in a change of view and a new view of $v + 1$ will be generated in the same consensus round b. Then **UpdateState** will be run to generate a new state for this new view: $\mathsf{st}_{b,v+1} \leftarrow \mathsf{st}_{b,v} \parallel b \parallel v + 1$. Since every view has a unique (v, b) pair, **UpdateState** will generate different states for two distinct views, and therefore generate different beacons.

With the **UpdateState** defined above, a node will generate different partial beacons for different views thus ensuring that each view has a distinct *DRB* number, preventing pre-computation and replay attacks targeting beacon generation. This approach allows for the secure integration of *DRB* into the *PBFT* consensus process while maintaining the desirable security properties of both schemes.

B. Randomness Request

Using the beacon as a source of random numbers in smart contracts introduces security risks. It is possible for different transactions to receive the same random numbers, which makes the system vulnerable to duplicate transaction replay

Input Validation-based Detection (IVD) $(\mathcal{SC}, \mathcal{T}, \sigma)$

For simplicity, we use the name of the contract as the contract address and abstract invoking function of $\mathcal{SC}$ as invoking $\mathcal{SC}$:

```
1if ({SC, T} ≠⊥) {
2    // Minimum transaction fee
3    // and maximum script size from SC
4    ($G_min, S_max) ⟵ SC;
5
6    ($g_max, S) ⟵ T
7    // Fee PUA Detected
8    if ($G_min ≤ $g_max) Revert();
9    // Script PUA Detected
10   if (S_max ≥ |S|) Revert()t;
11   // Acquire the entry script
12   // from the virtual machine
13   SC_entry ⟵ EntryScriptFromVM;
14   // Contract PUA Detected
15   if (SC_entry ≠ S) Revert();
16} else {
17   // Get the script from the blockchain
18   S ⟵ GetContract(Addr)
19    // Fallback PUA Detected
20   if (S ≠⊥) Revert();
21}
```

Figure 8: Input Validation-based Detection (IVD) to mitigate PUAs.

attacks. Adversaries can exploit this vulnerability by flooding the system with numerous transactions within the same block. Furthermore, if on-chain data is utilized to process the beacon, attackers could potentially precompute random numbers and incorporate them into their attack contracts, leading to pre-computation attacks. Even if random numbers are generated securely, generating the same random numbers for multiple requests in a single transaction would expose the system to pr-computation attacks.

To mitigate these concerns, BFTRAND.*Request* employs a pseudo-random function (*PRF*) to convert the beacon into multiple unique and unpredictable random numbers. We use the transaction hash as an identifier to ensure that transactions receive distinct random numbers and enter these numbers into the *PRF*. To prevent attackers from precomputing random numbers, the "GetRandom" function is designed to return different random numbers each time it is called. This is achieved by incorporating a global counter c as input, guaranteeing that each new request for random numbers has a unique input and produces a different output.

C. Mitigating PUAs

Ensuring secure RNP in BFTRAND requires addressing PUAs. We propose Input Validation-Based Detection (IVD) as a means to counter PUAs without compromising transaction atomicity. In order to reduce alterations to the existing blockchain platform, we have developed IVD as a smart contract function that users can invoke and implement within their smart contracts. IVD operates prior to the contract retrieving a random number, thereby identifying possible PUA attacks.

The IVD method uses existing smart contract regulations to authenticate user input, identify, and mitigate potential attacks. By individually examining each potential entry point for a PUA, the IVD approach ensures thoroughness in addressing security vulnerabilities. Furthermore, IVD maintains transaction atomicity by adhering to unaltered contract rules, making it extremely difficult for adversaries to exploit or bypass the IVD detection mechanism. Adversaries would need to create an entirely new PUA to circumvent the system, which poses a formidable challenge.

Figure 8 illustrates the IVD function against Contract, Fallback, Fee, and Script PUAs. The IVD function takes two parameters: $\{\mathcal{SC}, \mathcal{T}\}$, which represent the victim contract $\mathcal{SC}$ and the invocation of the transaction $\mathcal{T}$, and $addr$, the intended interaction address for the victim contract. If $\{\mathcal{SC}, \mathcal{T}\}$ is not empty on *line 1*, IVD proceeds to PUA script, contract, and fee detection. At *Line 4*, two contract variables, $\$G_{min}$ and S_{max}, established during development, are retrieved. $\$G_{min}$ represents the minimum transaction fee required to cover all execution paths, while S_{max} indicates the maximum allowed transaction script size, which prevents appending the verification script. At *Line 6*, the maximum transaction fee $\$g_{max}$ and the transaction script S are extracted from the parameters. At *line 8*, IVD ensures the absence of fee PUA issues by verifying that the transaction fee is sufficient. At *Line 10*, Script PUA threats are checked by comparing the transaction size to the contract's maximum script size. If either check fails, IVD is reverted. Otherwise, IVD verifies direct transaction initiation of contract execution by obtaining the execution's entry script from the virtual machine at *Line 13*. If the acquired entry script and the transaction script are not identical, IVD is reverted. At *Line 18*, IVD retrieves the contract script for $addr$ from the blockchain. If the loaded script is nonempty, indicating a contract account and potential Fallback PUA, IVD is reverted.

VI. Security Analysis of BFTRand Protocol

In this section, we provide a brief security analysis of the BFTRAND protocol, focusing on the four requirements (§III-C). The formal proofs can be found in the full version [9].

Theorem VI.1 (Pseudorandomness)**.** *If the underlying DRB is (f, k, n)-secure with $f \leq t < k \leq 2t + 1$, and PRF is a pseudorandom function, then* BFTRAND *satisfies pseudorandomness.*

Proof Sketch. This property follows directly from the pseudorandomness (i.e., the standard pseudorandomness described in [53]) of the underlying DRB protocol and *PRF*. Specifically, since the random beacons generated by DRB and the output from PRF are pseudorandom, the resulting random numbers are also pseudorandom. See the full proof in [9].

Theorem VI.2 (Uniqueness)**.** BFTRAND *satisfies uniqueness if PBFT is (f, t, n) secure, DRB is (f, k, n) secure, with $f \leq t < k \leq 2t + 1$, and PRF is a pseudorandom function.*

Proof Sketch. If *PBFT* is secure, then the adversary can control at most f nodes, as guaranteed by the underlying *PBFT* scheme, and this controlled number is less than the threshold defined in the *DRB* protocol, the adversary cannot generate valid random beacons at will, which is ensured by the uniqueness of the underlying *DRB* scheme. Furthermore, since the round number is determined by the block and view, only one predetermined beacon related to the block and view will be generated for each block. Additionally, as *PRF* is a deterministic algorithm, it guarantees that it can only produce the same result with the same inputs and that each random number generated from *PRF* has a different input. Consequently, as long as *PBFT* is secure, every random number generated for smart contracts is unique. The full proof can be found in [9].

Theorem VI.3 (Availability)**.** BFTRAND *satisfies availability if PBFT is* (f, t, n) *secure, DRB is* (f, k, n) *secure and achieves uniqueness, for any* f *with* $f \leq t < k \leq 2t + 1$.

Proof Sketch. As BFTRAND receives at least $2t+1$ beacons in the Finalize phase generated by different nodes, the beacon can be confirmed only if a value appears over t times. Following the uniqueness of *DRB*, the honest nodes generate the same beacons; hence, the honest nodes of BFTRAND can always output a valid beacon, even in the faulty nodes. Therefore, BFTRAND achieves *availability*. See the full proof in [9].

Theorem VI.4 (Irreversibility)**.** BFTRAND *achieves Irreversibility.*

Proof Sketch. BFTRAND verifies whether the contract is directly invoked by a transaction (*Contract PUA*), whether the interacting address is a contract account (*Fallback PUA*), whether there are adequate fees to cover all execution flows (*Fee PUA*), and whether there are additional transaction scripts (*Script PUA*). 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, *Irreversibility* is achieved. See the full proof in [9].

VII. Implementation and Evaluation

We have developed a prototype of BFTRAND using the Delegated Byzantine Fault Tolerant (dBFT) consensus algorithm [75], which is an optimized variant of PBFT [33] and is being adopted by many famous blockchain projects [13], [52], [69], [75]. We set its block time to 15 seconds. Since dBFT is a standard variation of PBFT, the protocols of dBFT can be adapted to other BFT platforms. In our implementation, we integrated BFTRAND into dBFT by modifying the messages in the classic phases, while preserving the original consensus function (§V-A). For the prototype DRB protocol BFTRAND, we instantiated the Boneh-Lynn-Shacham (BLS) threshold-based Dfinity-DRB [53], [57]. We use the neo virtual machine as the dApp execution environment, which can execute smart contracts that are implemented with C#.

To evaluate BFTRAND, we implemented four applications: 1) Fair NFT Distribution Application: We migrate the Loot [81] token distribution contract to nvm, where each bag is assigned a random rarity when users claim Loot tokens; 2) Multiple Random Number Request Application: We reimplemented Neoverse [78] to generate multiple random values in a single transaction, allowing users to open multiple Blind Boxes simultaneously; 3) Fair Gaming Outcome Application: We developed a rock-paper-scissor contract that enables users to participate in prize giveaways with random and fair outcomes; 4) Performance Comparison Application: We designed an application to compare the performance of commit-execute random number generation solutions with BFTRAND.

Furthermore, we also compared BFTRAND with other RNP solutions. However, to the best of our knowledge, BFTRAND is the first one-round RNP solution. Since existing schemes deploy the commit-and-execute framework, they are all two-round based. Therefore, we compare BFTRAND with two-round protocols in our experiments. To ensure a fair evaluation and comparison, we introduced BFTRAND$_{commit-execute}$, a variant of our protocol that adheres to this traditional two-round approach, serving as a bridge for comparison with current two-round schemes. Specifically, we compare our protocol with existing commit-and-execute-based ones in terms of blockchain storage overhead, GAS fees, and latency. Furthermore, our comparison also extends to Chainlink VRF [34], a widely recognized and leading standard in random number generation solutions for smart contracts, utilized by over 772 projects [46]. BFTRAND is evaluated using a quad-core 3.6 GHz Intel(R) E3-1275 v5 CPU [59] and 32 GB of memory. The operating system used was Ubuntu 20.04 LTS with the Linux kernel version 5.4.0-91-generic, and the SDK version was .NET 6.0.1. The lines of code for different parts of BFTRAND can be found in the full version [9].

A. Application Transaction Cost.

Table I summarizes the transaction costs of running the developed applications. In nvm, the network fee is proportional to the length of the transaction script, while the OpCodes determine the system fee that a transaction executes. GAS is the token in nvm that pays the 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 five blind boxes. RPS::Play produces a random shape and compares it with the one provided by the user. This evaluation demonstrates that smart contracts can conduct random number-related operations inexpensively with one single transaction.

B. Transaction Fee Cost.

We compared the fees for requesting and providing random numbers between BFTRAND and BFTRAND$_{\texttt{commit-execute}}$. The implementation of generating multiple random numbers within BFTRAND$_{\texttt{commit-execute}}$ is carried out in C# on

Table I: 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:: Play	0.00616260/0.013	0.06588677/0.146

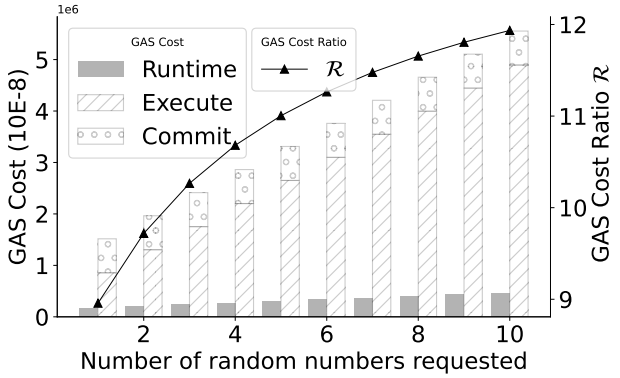


Figure 9: GAS cost when calling `GetRandom`. The left y-axis is the total GAS consumption, while the right y-axis is the GAS cost ratio $\mathcal{R} = (\texttt{Commit} + \texttt{Execute})/\texttt{Runtime}$.

the VRF-based Chainlink platform[1]. The evaluation result is shown in Figure 9.

From the results, we observed that BFTRAND costs 0.00169291 GAS ($0.037), which is only 11% of the cost of issuing a random number request for $\text{BFTRAND}_{\texttt{commit-execute}}$ (0.01516372 GAS, $0.033). If each transaction requests ten random numbers, then BFTRAND saves 91.6% GAS (0.05088786 GAS, $0.113). BFTRAND saves a considerable amount of transaction fees since $\text{BFTRAND}_{\texttt{commit-execute}}$ needs 2 transactions, `Commit` and `Execute`, to complete one request. `Execute` needs to verify the fairness of the beacon and extend the beacon to random numbers in smart contracts through the virtual machine, which is expensive and inefficient. In contrast, BFTRAND can generate random numbers in the native environment and guarantee the fairness of the beacon through consensus.

GAS Cost of Irreversibility Check. The cost to ensure irreversibility with IVD is less than 0.00001 GAS ($0.00002), which is negligible compared to the transaction cost.

C. Blockchain Overhead.

To evaluate the impact of RNP on the blockchain ledger, we define the size of the data that must be written on the blockchain to complete a request for random numbers as blockchain overhead (BO). Since Chainlink VRF [34] is one of the most influential RNP solutions, we use it as a baseline. Let the transaction size of $\mathcal{T}_{\texttt{execute}}$ be len. We evaluate the BO with the transaction size $\mathcal{T}_{\texttt{execute}}$ $len = 193$ bytes (the size of a typical nvm transfer transaction) and $len = 624$ bytes[2], while $\mathcal{T}_{\texttt{commit}}$ size is set as the nvm transaction size of 193 bytes. The BO for BFTRAND is 193 bytes, and the overhead for $\text{BFTRAND}_{\text{commit-execute}}$ is $193 + len$. If $len = 193$ bytes, then BFTRAND saves approximately 50% BO compared to $\text{BFTRAND}_{\texttt{commit-execute}}$; If $len = 624$ bytes, BFTRAND saves approximately 76.4% BO.

[1]Chainlink VRF [35] is the most influential RNP solution, which has processed approximately 8 million requests for random numbers for more than 656 projects.

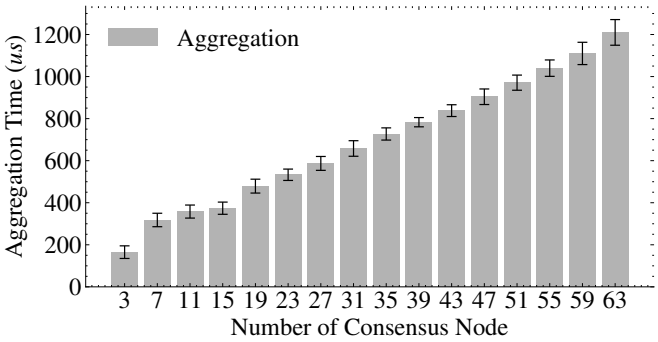


Figure 10: The BLS signature aggregation time.

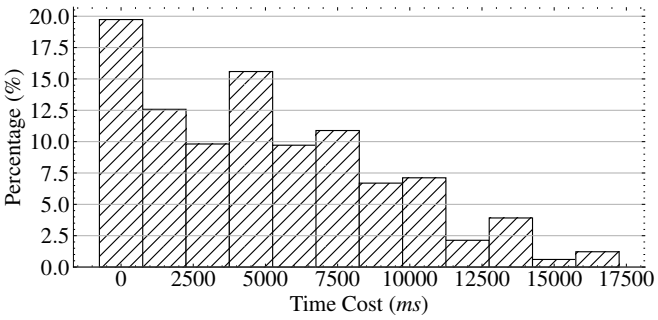


Figure 11: BLS setup time cost in a testnet with 22 consensus nodes. Note that BLS setup is separate to the consensus, it will not slow down the consensus.

D. Evaluation on Beacon Request.

Since the adopted DRB exploits the BLS signature to generate the random beacon, we evaluated the aggregation time of σ. Figure 10 shows the aggregation time on the logarithmic scale under the 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-chain) and $k = 4$, the time cost of aggregating the BLS signatures is approximately 318 μs; when the number of consensus nodes increases to 23 (comparable to 21 nodes in EOS), the time cost is approximately 533 μs. Adding BFTRAND to dBFT only incurs a negligible increase of 0.002% more time.

To evaluate the setup, we executed it on a private network of 22 dBFT nodes deployed on Azure. Azure nodes utilized Intel (R) Xeon (R) Platinum 8272CL CPU 2.6 GHz with

[2]The size of a commit-execute transaction in Chainlink VRF [49], 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 from the blockchain.

Ubuntu18.04.6 LTS. The time to live (ttl) between the nodes was approximately 57 ms. The evaluation results are depicted in Figure 11 with $2,000$ setups. More than half of the setups were completed in 5 seconds, with some cases (e.g. 15%) taking more than 10 seconds. This longer setup time is mainly influenced by the communication cost associated with collecting messages from all participants. Because the setup operation is independent of consensus and is only performed when the consensus committee requires an update and is separate from the consensus, execution of the setup does not impose overhead on the consensus layer.

E. RNP Solutions Comparison.

In Table II, we provide a comparison of BFTRAND with various projects related to the generation of random numbers. This includes beacon protocols [19], [31], [36], [56], [93], [93], blockchain projects with embedded random number protocols [2], [21]–[24], and oracles [35], [76].

1) Platform consensus: According to the table, the majority of the protocols in the comparison are based on the BFT consensus. This is because in order to generate random values in a distributed environment, the nodes must communicate interactively to establish consensus. Chainlink VRF, being an oracle, is platform independent and provides randomness for both BFT-based blockchains [28] and Proof-of-Work (PoW) based blockchains [49]. Automata [76] is a blockchain middleware specifically designed for Ethereum. As BFTRAND is built on top of the consensus algorithm PBFT, it can be easily adapted to any blockchain project that utilizes variants PBFT as its underlying consensus algorithm.

2) Method: To prevent a faulty node from obstructing the provision of random numbers, HDrand, RandRiper, Dfinity, Elrond and BFTRAND use distributed random beacon (DRB) protocols to generate random numbers. On the other hand, Klaytn and Harmony use verifiable random functions to ensure that the system generates deterministic random values. In addition to threshold signatures and verifiable random functions, several projects such as Secret, Chainlink VRF, and Automata use Trusted Execution Environments (TEE) [7], [11], [73], which are hardware-protected isolated execution environments, to generate random values in a secure and unbiased manner. The usage of TEE requires operating on devices that support TEE.

3) Comparing the number of random numbers: We compare the ability of random number generators (RNPs) to generate random numbers. Random beacon projects such as RandHerd, RandHound, and BRandRiper focus solely on beacon generation, limiting the available random value to the number of beacons σ. On the other hand, Secret and Elrond provide built-in pseudorandom functions that can generate numerous random numbers by expanding σ. However, the random value they can supply is only upper bound by the consensus (e.g., GAS limit), represented as ∞. In the case of BFTRAND, random numbers are generated using a pseudorandom function (PRF) of the consensus-based beacon. Therefore, the number of random numbers generated by BFTRAND is also constrained by consensus, such as the transaction fee.

4) Examining latency: None of the random number generators used by random beacon or blockchain projects can generate random values within a single consensus round due to vulnerabilities to random number attacks. However, Automata [76] is the only one capable of providing run-time random numbers to smart contracts. It is a trusted third-party middleware solution for the Ethereum Virtual Machine (EVM) platform. It generates random numbers in a Trusted Execution Environment (TEE) through Verifiable Random Functions (VRF) and uses Ethereum EIP712 [86] to encapsulate a user transaction and its requested random numbers within an EIP712 transaction.

While Automata is theoretically vulnerable to PUA, it is highly improbable in practice due to its closed solution nature, operating exclusively on its own services like NFTFair. In contrast, BFTRAND is an RNP solution specifically designed for BFT-based open blockchain platforms, free of the reliance on specialized hardware and trusted third parties.

5) Limitations of BFTRAND*:* Unlike the existing commit-and-execute solutions that are consensus agnostic and compatible with various blockchain platforms, $BFTRand$ is specifically tailored for BFT-based platforms. Due to the fundamental differences between BFT consensus mechanisms and other consensus types such as PoW [74] and PoS [48], $BFTRand$ cannot be directly implemented on blockchain platforms that utilize non-BFT consensus models.

We would like to highlight that BFTRAND is the first one-round RNP solution designed for BFT platforms, which is our main contribution. Existing schemes deploy the commit-and-execute model, which is two-round based. Therefore, BFTRAND stands out as the only secure runtime random number provider for BFT-based blockchains.

VIII. Related Work

In this section, we provide a concise overview of the existing literature on the provision of random numbers in blockchain systems, highlighting their limitations and vulnerabilities.

On-Chain RNPs and Commit-execute RNPs: On-chain RNP approaches utilize blockchain data as entropy sources [29], while commit-execute schemes involve a two-phase process [2], [31], [62]. However, both methods have limitations, such as susceptibility to manipulation, prediction attacks [27], [41], [94], increased latency, and storage overhead [34], [51], [84].

Distributed Random Beacons and Smart Contract RNPs: The DRB protocols are designed to generate random numbers in a secure and decentralized manner [2], [21], [45]. Various studies on random beacons utilize different cryptographic primitives and threat models [18], [31], [54], [56], [62], [88], [89]. Some projects rely on homomorphic encryption [36], [79], while others operate in a permission-less setting [1], [4], [21], [24], [35], [44], [45], [50], [65]. While DRBs have made significant advancements in efficiently and securely generating random numbers, they are not explicitly designed as RNP solutions for smart contracts. Thus, they must be combined

Table II: Comparison of RNPs for blockchain.

Protocol	Platform Consensus	Method(s)	Resistance (t)	# random values (r)	Latency (Consensus round)
Drand [31]	PABFT	Threshold SecretBLS	$t < n/2$	$\mathcal{O}(\sigma)$	≥ 2
HERB [36]	∅	Threshold ElGamal	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
RandChain [56]	Sequential PoW	PoW	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
RandHerd [93]	BFT	Threshold Schnorr	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
RandHound [93]	BFT	Client based, PVSS	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
BRandRiper [19]	BFT	VSS, q-SDH	$t < n/2$	$\mathcal{O}(\sigma)$	≥ 2
Dfinity [2]	BFT	Threshold BLS	$t < n/2$	∞	≥ 2
Secret [24]	DPoS	Scrt-RNG,TEE	$t < n/2$	∞	≥ 2
Elrond [21]	Secure PoS	BLS,onchain data	$t < n/3$	∞	≥ 2
Klaytn [23]	Istanbul BFT	VRF	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
Harmoney [22]	Fast BFT	VRF,VDF	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
⋆Chainlink VRF [35]	∅	VRF, TEE	$t < n/2$	$\mathcal{O}(\sigma)$	≥ 2
⋆Automata [76]	∅	VRF, TEE	$t < n/2$	∞	1
BFTRAND$_{commit-execute}$	BFT	∅	$t < n/3$	$\mathcal{O}(\sigma)$	≥ 2
BFTRAND	BFT	Threshold BLS	$t < n/3$	∞	1[§]

In the table, n denotes the number of consensus nodes, t is the maximum number of Byzantine nodes allowed in the system, and σ denotes the beacon. **Resistance** refers to the tolerance of the system for Byzantine faults. ⋆ is the off-chain third-party Oracle RNP. ∞ means the number of random numbers is upper-bounded by consensus. [§]BFTRAND is the first smart contract solution in runtime RNP on a BFT-based blockchain.

with the commit-execute scheme to provide randomness. However, the integration of DRB protocols with smart contracts at runtime presents unique security challenges.

Hardware RNPs and BFT Blockchain Projects: Hardware-based methods, such as ASIC-based VDF [25] and TEE-based oracles in Chainlink VRF [35] and Secret [24], rely on specific hardware. CoRNG [20] uses the shared memory to generate random numbers. Several BFT blockchain projects have also been developed [5], [22], [23], [47], [58], [77], [82], [83], [91]. BFTRAND has the potential to be integrated into these projects and is hardware-independent.

IX. Conclusion and Discussion

Conclusion. Ensuring reliable randomness for smart contracts is a crucial aspect of various blockchain operations. Many decentralized applications heavily rely on randomness to achieve utility and security. However, existing commit-execute random number provider protocols face challenges such as high costs on-chain and delayed processing times. To address these issues, this study introduces BFTRAND, a novel low-latency Random Number Protocol specifically designed for smart contracts on Byzantine Fault-Tolerant (BFT)-based blockchains. The proposed protocol, BFTRAND, is referred to as low latency due to its ability to securely provide random numbers using a single round of consensus, as opposed to the two rounds required by commit-execute RNPs. To assess its performance, we implemented a prototype of BFTRAND using a distributed random beacon protocol and seamlessly integrated it into the consensus layer without compromising efficiency. Furthermore, BFTRAND has been rigorously tested and proven secure against random number attacks. Furthermore, we identify various types of *post-reveal Undo Attacks* and propose effective mitigation strategies to improve the security of BFTRAND. Consequently, BFTRAND is the first of its kind for blockchains, providing a secure, reliable, and efficient random number provider function.

Discussion. BFTRAND is specifically designed for smart contracts on blockchain platforms, integrating Beacon Requests, Randomness Requests, and detection mechanisms tailored to this environment. However, some aspects of BFTRAND, such as the Beacon and Randomness Requests designed for BFT-based platforms, can be extended to other systems using (variants of) BFT consensus. The detection mechanisms are only applicable to smart contracts. Specifically, the core of BFTRAND is integrated within the BFT-based consensus process, enabling its application in systems that utilize BFT consensus. Nonetheless, the unique requirements for randomness in blockchain smart contracts, coupled with the specific challenges of pre-computing and Miner Extractable Value (MEV) attacks [92], necessitate specialized detection mechanisms. These attacks, inherent to the blockchain's distributed and transparent nature, endanger the integrity of randomness, making our detection mechanisms particularly suited for on-chain smart contracts or related applications. This specificity is due to the blockchain's visibility in the random number generation process, which does not generally apply outside of this context.

Acknowledgement

We wish to thank Neo Global Development and Automata Network in particular, for ongoing research discussions and generous support of a number of aspects of this work. We also wish to thank Erik Zhang, Chua Zheng Leong, Steven Liu, Yongqiang Wang, Shuai Li, and Zhitong Chen for their helpful feedback and discussion. We also thank Qiao Jin, Mengyu Liu, and Owen Zhang, Xinyi Lu for their contributions to application development and evaluation. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62372218, and Shenzhen Science and Technology Program under Grant No. SGDX20201103095408029. This work is also supported by the Hong Kong Research Grants Council under Project Nos. C1029-22G and R1012-21.

REFERENCES

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

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. *Cryptology ePrint Archive*, 2018.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[4] A Byte Ahead. 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] VDF ALLIANCE. Open vdf: Asic introduction. https://www.vdfalliance.org/news/open-vdf-asic-introduction, April 2020.

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

[8] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing*, 8(4):564–577, 2010.

[9] Anonymous. Bftrand: Low-latency random number provider for bft smart contracts. https://shorturl.at/clsBH, 2023.

[10] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

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

[12] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.

[13] Diem Association. Diem. https://www.diem.com/en-us/, 2019.

[14] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). *IACR Cryptology ePrint Archive*, 2017:1–35, 2017.

[15] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013.

[16] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[17] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[18] Adithya Bhat, Aniket Kate, Kartik Nayak, and Nibesh Shrestha. Optrand: Optimistically responsive distributed random beacons. *Cryptology ePrint Archive*, 2022.

[19] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. Randpiper–reconfiguration-friendly random beacons with quadratic communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3502–3524, 2021.

[20] Peva Blanchard, Rachid Guerraoui, and Julien Stainer. Concurrency as a random number generator - companion technical report. 01 2016.

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

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

[23] Klaytn Blockchain. Consensus randomness. https://docs.klaytn.com/klaytn/design/consensus-mechanism, 2022.

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

[25] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[26] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. On the necessity of a prescribed block validity consensus: Analyzing bitcoin unlimited mining protocol. *Queue*, 13(7):20, 2015.

[27] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015.

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

[29] Vitalik Buterin and ethereum.org. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[30] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[31] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[32] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[33] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[34] chain.link. https://chain.link/. https://chain.link/. accessed: 2020-08-18.

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

[36] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. Homomorphic encryption random beacon. *Cryptology ePrint Archive*, 2019.

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

[38] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[39] coinmarketcap.com. Today's cryptocurrency prices by market cap. https://coinmarketcap.com/, March 2022.

[40] Mauro Conti, Divya Diwaker, Sankardas Roy, and Radha Poovendran. A survey on blockchain: Techniques, applications, and performance evaluation. *IEEE Communications Surveys & Tutorials*, 21(4):2678–2716, 2019.

[41] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[42] DAMIEN. Hackers targeted eosbet dapp, $200k worth of eos stolen, 2018.

[43] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2502–2517. IEEE, 2022.

[44] TRON Core Devs. Safe practice of tron solidity smart contracts: Implement random numbers in the contracts, Mar 2020.

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

[46] Chainlink Ecosystem. Chainlink ecosystem. https://www.chainlinkecosystem.com/ecosystem, 2022.

[47] EOSIO. Eosio website. https://eos.io/, November 2022.

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

[49] Etherscan. Chainlink vrf. https://etherscan.io/address/0x271682DEB8C4E0901D1a1550aD2e64D568E69909/, 2022.

[50] 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.

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

[52] The Linux Foundation. Hyperledger fabric. https://www.hyperledger.org/use/fabric, 2016.

[53] David Galindo, Jia Liu, Mihair Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 88–102. IEEE, 2021.

[54] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[55] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018.

[56] Runchao Han, Jiangshan Yu, and Haoyu Lin. Randchain: Decentralised randomness beacon from sequential proof-of-work. *IACR Cryptol. ePrint Arch.*, 2020:1033, 2020.

[57] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

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

[59] 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.

[60] Aljosha Judmayer, Nicholas Stifter, Philipp Schindler, and Edgar Weippl. Estimating (miner) extractable value is hard, let's go shopping! *Cryptology ePrint Archive*, 2021.

[61] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[62] John Kelsey, Luís TAN Brandão, Rene Peralta, and Harold Booth. A reference for randomness beacons: Format and protocol version 2. Technical report, National Institute of Standards and Technology, 2019.

[63] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[64] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th USENIX Security Symposium*, pages 1317–1334. USENIX Association, 2019.

[65] Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. Distributed random number generator on hedera hashgraph. In *2020 the 3rd International Conference on Blockchain Technology and Applications*, pages 7–11, 2020.

[66] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[67] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[68] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. *arXiv preprint arXiv:1906.03819*, 2019.

[69] Ontology Foundation Ltd. Ontology. https://ont.io/, 2018.

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016.

[71] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[72] Sky Mavis. Paid whitepaper. https://docsend.com/view/jdbdpza9d9nehnf2, January 2021.

[73] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10, 2013.

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

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

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

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

[78] NGD. Neoverse, 2022.

[79] Thanh Nguyen-Van, Tuan Nguyen-Anh, Tien-Dat Le, Minh-Phuoc Nguyen-Ho, Tuong Nguyen-Van, Nhat-Quang Le, and Khuong Nguyen-An. Scalable distributed random number generation based on homomorphic encryption. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 572–579. IEEE, 2019.

[80] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018.

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

[82] The Celo Project. Celo randomness: Celo docs. https://docs.celo.org/celo-codebase/protocol/identity/randomness.

[83] The Kadena Project. Kadena whitepaper. https://docs.kadena.io/basics/whitepapers/overview.

[84] Youcai Qian. Randao: Verifiable random number generation, 2017.

[85] Mayank Raikwar and Danilo Gligoroski. Sok: Decentralized randomness beacon protocols. *arXiv preprint arXiv:2205.13333*, 2022.

[86] Leonid Logvinov Remco Bloemen. Eip-712: Ethereum typed structured data hashing and signing. https://eips.ethereum.org/EIPS/eip-712.

[87] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar Weippl. Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness. 2021.

[88] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Efficient continuous distributed randomness. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 73–89. IEEE, 2020.

[89] Srđan Daniel Simić, Robert Šajina, Nikola Tanković, and Darko Etinger. A review on generating random numbers in decentralised environments. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1668–1673. IEEE, 2020.

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

[91] solana.com. Solana documentation. https://docs.solana.com/introduction, 2022.

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

[93] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[94] RANDAO Team. Randao: A dao working as rng of ethereum. https://github.com/randao/randao, 2016.

[95] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144. IEEE, 2009.

[96] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[97] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. In *Ethereum project yellow paper*, volume 151, pages 1–32, 2014.

[98] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.