electronics

MDPI

Article

Enhancing Consensus Security and Privacy with Multichain Ring Signatures Based on HotStuff

Mingan Gao, Zhiyuan Wang and Gehao Lu *

School of Information Science, Yunnan University, Kunming 650500, China; gaomingan@stu.ynu.edu.cn (M.G.); wzyyx@mail.ynu.edu.cn (Z.W.)
* Correspondence: glu@ynu.edu.cn

Abstract: The paper introduces a novel consensus algorithm named MRPBFT, which is derived from the HotStuff consensus protocol and improved upon to address security deficiencies in traditional consensus algorithms within the domain of digital asset transactions. MRPBFT aims to enhance security and privacy protection while pursuing higher consensus efficiency. It employs a multi-primary-node approach and a ring signature mechanism to reinforce security and privacy preservation features in the consensus system. This algorithm primarily focuses on two main improvements: Firstly, it proposes the ed25519LRS signature algorithm and discusses its anonymity for transaction participants and the non-forgeability of signature information in the identity verification and message verification processes within the consensus algorithm. Secondly, the paper introduces MPBFT asynchronous view changes and a multi-primary-node mechanism to enhance consensus efficiency, allowing for view switching in the absence of global consensus. With the introduction of the multi-primary-node mechanism, nodes can be flexibly added or removed, supporting parallel processing of multiple proposals and transactions. Finally, through comparative experiments, the paper demonstrates that the improved algorithm performs significantly better in terms of throughput and network latency.

Keywords: MRPBFT consensus algorithm; multi-primary nodes; privacy protection; ring signature

Citation: Gao, M.; Wang, Z.; Lu, G. Enhancing Consensus Security and Privacy with Multichain Ring Signatures Based on HotStuff. *Electronics* **2023**, *12*, 4632. https://doi.org/10.3390/electronics12224632

Received: 13 September 2023
Revised: 7 November 2023
Accepted: 7 November 2023
Published: 13 November 2023

1. Introduction

1.1. Research Background and Motivation

Since the initial release of Bitcoin blockchain technology by Satoshi Nakamoto in 2008 [1], the application and development of this technology have been notable worldwide, with broad applications in fields, such as finance, supply chain management, healthcare, and more. Blockchains primarily exist in three types: public, consortium, and private chains. Public chains, like Bitcoin and Ethereum, are entirely open and decentralized networks. Consortium chains, on the other hand, are blockchains maintained and transaction-verified by specific entities. These chains often employ consensus algorithms different from those used in public chains. Lastly, private chains stringently restrict participant access and use to ensure high levels of privacy and authority control. However, the considerable energy consumption of the Proof of Work (POW) consensus algorithm [2] used by Bitcoin has elicited widespread societal concern. As a result, some blockchain projects, such as Ethereum, have adopted more eco-friendly consensus algorithms, like Proof of Stake (POS) [3].

1.2. Limitations of Previous Work

Consensus algorithms, such as Paxos [4], Practical Byzantine Fault Tolerance (PBFT) [5], and HotStuff [6], have been predominantly adopted in the context of consortium and private chains. The Paxos algorithm, proposed by Leslie Lamport in 1990, is a classic consensus algorithm for distributed systems aimed at ensuring their consistency and security. However, it has limitations concerning complexity, performance overhead, and single-leader

issues. The PBFT algorithm, proposed by Miguel Castro and Barbara Liskov in 1999, offers a solution for the consistency issue in distributed systems. Nevertheless, its performance loss increases significantly with the addition of nodes. HotStuff, an improved Byzantine fault-tolerant consensus algorithm, presents considerable advancements compared to PBFT. However, it still suffers from problems, such as a single primary node issue, insufficient security of the aggregated signature algorithm, and overall performance deficiency.

1.3. Contributions of This Research

The main contributions of our study are as follows. First, we propose an efficient consensus algorithm based on multiple primary nodes and ring signatures to address the shortcomings of the HotStuff consensus algorithm. Our algorithm enhances system liveness and scalability by introducing a design with multiple primary nodes that can flexibly add or remove nodes. This model supports parallel processing of multiple proposals and transactions, adapting better to dynamic system changes. The design of multiple primary nodes also improves fault tolerance and scalability, thereby further enhancing the liveness of the consensus system. Second, we enhance the security and privacy protection of the consensus algorithm by adopting a ring signature algorithm. Ring signatures allow multiple nodes to jointly sign documents or messages while maintaining anonymity without disclosing their identity information. The use of ring signatures ensures the validity and non-forgery of signatures and provides higher security for identity and message verification in the consensus algorithm. Consequently, the system becomes more resilient to attacks and tampering. Third, we optimize the view change process to improve consensus efficiency. Traditional consensus algorithms have a time-consuming view change process, and view switching can only occur once all nodes have reached consensus. To address this, we propose a consensus algorithm based on an asynchronous, optimized view change process. Our algorithm allows nodes to start a new view without requiring global consensus, reducing waiting time, and improving the system's throughput and response performance.

1.4. Organization of the Article

The remainder of this paper is organized as follows: Section 2 provides a review of the related work. Section 3 introduces the principles of the ed25519LRS signature algorithm, followed by an analysis of its security and analyzes the view changes for the MRPBFT consensus algorithm. Section 4 presents the experimental evaluations. Finally, Section 5 concludes this paper.

2. Related Work

Current improvements in the performance and security of the PBFT consensus algorithm have been extensively studied. However, these improved algorithms still present several issues: Firstly, most of these protocols adopt a single master node mechanism, resulting in poor scalability and inadequate fault tolerance. Secondly, these protocols cannot satisfy the requirements for anonymity and reliability. The signature algorithms they adopt are generally BLS12 aggregate signatures [7,8] or ecdsa [9] signatures, failing to meet security requirements adequately. Lastly, while some algorithms satisfy the above two points, they cannot meet performance requirements. Existing consensus algorithm solutions have at least one of these issues.

Single Master Node Failure: In addressing the single master node failure issue, several significant works exist. For example, Casper [10] and Tendermint [11] provide a simple leader master node replacement mechanism. However, both protocols have a synchronous core, meaning replicas in the network must wait for the maximum network latency time to enter the next round. HotStuff and DiemBFT [12] are two designs aiming to maintain responsiveness and a simple leader replacement process. Especially in the new version of DiemBFT, it performs similarly to Fast-HotStuff [13] in the normal phase, adopting PBFT-based double view change. In our consensus algorithm, however, we propose a

multi-node mechanism with three nodes, which can effectively avoid single-node failure and reduce the PBFT master node failure time consumption from $\mathcal{O}(n^3)$ to $\mathcal{O}(1)$ (Table 1).

Anonymity Issue: For the anonymity issue, it generally involves the selection of signature schemes. For instance, HotStuff and LibraBFT (DiemBFT) use threshold signatures [14] and aggregate signatures, respectively. However, the cost of signature verification depends on the type of signature scheme used by the protocol. For example, the verification cost of aggregate signatures is linearly related to the number of aggregated signatures, while the verification cost of threshold signatures is constant. In a recent paper [15], the author proposed a double node fault-tolerant mechanism but made no improvements regarding the anonymity issue. Our MRPBFT proposes a chained ring signature algorithm [16,17] based on ed25519 [18] effectively addressing the anonymity problem.

High View Change Delay: In addressing the issue of high view change delay, several protocols have made contributions. For instance, PBFT has a two-phase message complexity during normal operation or the `happy path`. On the other hand, Fast-HotStuff uses a rotating master node, with linear view changes during normal master node rotation (view change). A parallel research work, Wendy [19], solves the high latency issue during the `happy path` in HotStuff while supporting a rotating master node, avoiding the two-phase authenticator complexity during view changes. However, if the failed master node or newly selected master node (when less than $2f + 1$ correct nodes hold locks) is Byzantine, it can force the protocol to generate additional delay costs during view change. The authors of NBFT [20] propose using a consistent hashing algorithm to group consensus nodes to improve network performance.

Table 1. Comparison between BFT algorithms.

Algorithms	Leader Failure (View-Change)	f Leader Failures	Leader Paradigm	Optimistic Responsiveness	Anonymity
PBFT	$\mathcal{O}(n^3)$	$\mathcal{O}(fn^3)$	stable	Yes	No
Tendermint	$\mathcal{O}(n^2)$	$\mathcal{O}(fn^2)$	stable	No	No
DiemBFT	$\mathcal{O}(n)$	$\mathcal{O}(fn^2)$	rotating	Yes	No
HotStuff	$\mathcal{O}(n)$	$\mathcal{O}(fn)$	rotating	Yes	No
Fast-HotStuff	$\mathcal{O}(n)$	$\mathcal{O}(fn)$	rotating	Yes	No
MRPBFT	$\boldsymbol{\mathcal{O}(1)}$	$\boldsymbol{\mathcal{O}(fn)}$	**stable**	**Yes**	**Yes**

3. Materials and Methods

3.1. Design Principle of Ring Signature Algorithm

This section introduces the improvement of the ed25519 signature algorithm in the HotStuff consensus algorithm to an ed25519-based chained ring signature algorithm, hereinafter referred to as ed25519LRS. The explanation will be given from three aspects: public key generation, signature, and verification.

3.1.1. Public Key Generation

(a) The generation of public keys is performed using the function `genPubKey(size, privateKey, index)`, which has three parameters. The parameter *size* represents the size of the generated public key *ring*, *privateKey* represents the private key to be inserted into the ring, and *index* represents the position of the public key to be inserted in the ring. *size* must be smaller than *index*, as the index cannot exceed the size range of the ring.

(b) A public key ring *ring* of size *size* is created, along with a map type variable *hashes* used to cache hash values generated by the public key.

(c) The function first uses scalar multiplication to calculate the corresponding public key

$$publicKey = privateKey * G \tag{1}$$

using the private key *privateKey* and the base point *G* on the elliptic curve. Then, it inserts this public key *publicKey* into the given ring *ring* at the *index* position. In this paper, $*$ represents scalar multiplication.

(d) For other positions in the ring, the function calculates *publicKey* using a randomly generated *privateKey*, then inserts it into the ring. Generating a public key with a randomly generated private key can prevent leakage during transmission. Algorithm 1 is a pseudocode representation of the `genPubKey` function.

Algorithm 1 genPubKey Function

```
procedure GENPUBKEY(size, privateKey, index)
    if index ≥ size then
        return error "index out of bounds"
    end if
    ring ← create an array of length size
    publicKey ← curve.ScalarBaseMul(privateKey)
    ring_index ← publicKey
    for i ← 1 to size do
        index ← (i + index)  mod size
        privateKey ← curve.NewRandomScalar()
        ring_index ← curve.ScalarBaseMul(privateKey)
    end for
    return ring
end procedure
```

3.1.2. Signature Creation

(a) The signature function `ringSign(msg,ring,privateKey, index)` carries four parameters where *msg* represents the message to be signed. The message *msg* is first processed through a hash function *H*, which results in a fixed-length (32-byte) hash value that will be used for signing. *Ring* refers to the public key ring generated by the `genPubKey` function. *privateKey* and *index* remain consistent with the previous explanation.

(b) Initially, the function validates that the length of the public key ring is at least 2, and the index falls within the range of the ring. Following this, it confirms whether the public key, corresponding to the provided index, has been truly generated by the input private key.

(c) The function initializes a new ring signature object, *RingSig*, and calculates a key image *keyImage*.

$$keyImage = privateKey * H(publicKey) \tag{2}$$

keyImage is generated by the signer's private key, but it cannot be used to reverse-engineer the private key. A key property of *keyImage* is that the same private key always generates the same *keyImage*, regardless of the ring in which the signature is created. Its primary role is to prevent double signatures.

(d) The function then traverses each public key in the ring. For each *pubicKey*, it selects a random scalar value *val* and uses this *val* and *pubicKey* together to generate a verification value $verify$. The role of the verification value $verify$ is mainly to ensure the security and non-forgeability of the signature. It is a crucial method of constructing a signature, aimed at preventing the signer or others from forging signatures without the correct private key. $verify$ is computed based on the public key in the ring and some random values through a hash function (or other functions). In the signature process, the calculation of the $verify$ value involves information from the previous *pubicKey* and some randomly chosen values. Each $verify$ value influences the calculation of the next verification value $verify$, thus forming a closed loop, which is the origin of the ring signature name. Finally, the function uses the initially chosen random scalar value *val* and the last $verify$ value to close the loop, thereby generating a complete ring signature *RingSig*. Algorithm 2 is a pseudocode representation of the `ringSign` function.

Algorithm 2 Sign Function

procedure SIGN(*msg*, *privateKey*, *index*)
  size ← length of *ring.publicKey*
  publicKey ← `ring.ed25519.ScalarBaseMul`(privateKey)
  H ← `hashToCurve`(publicKey)
  $verify$ ← `ed25519.ScalarMul`(privateKey, H)
  ringSing ← new RingSign object with *ring* and $verify$
  val ← `ed25519.NewRandomScalar`()
  l ← `ed25519.ScalarBaseMul`(val)
  r ← `ed25519.ScalarMul`(val, H)
  $idx \leftarrow (index + 1) \mod size$
  $verify[idx]$ ← `verifyVale`(ring.ed25519, msg, l, r)
  for $i \leftarrow 1$ **to** *size* **do**
    $idx \leftarrow (index + i) \mod size$
    val_idx ← `ed25519.NewRandomScalar`()
    $verify_{idx+1}$ ← `verifyVale`(ed25519, msg, l, r)
  end for
  $s_index \leftarrow val - verify \times x$
  $ringSing.val \leftarrow val$
  $ringSing.verify \leftarrow verify_0$
  return *sig*
end procedure

3.1.3. Signature Verification

(a) The verification function `verify(msg)` receives a message *msg* to be verified.

(b) The function is initially set up by retrieving the public key ring from the signature and its size. It also creates an array *arr* to store verification values $verify$. These verification values are used to check the validity of the signature within the ring.

(c) The function enters a loop, where for each public key in the ring, it calculates two values, L and R. These two values are calculated as follows:

$$L = val * G + verify * publicKey. \tag{3}$$

This equation is an elliptic curve operation, where *val* is a part of the *RingSig* signature object generated by the signing function above, G is the base point of the elliptic curve, $verify$ is the verification value, and *publicKey* is the public key.

$$R = val * H(P) + verify * keyImage. \tag{4}$$

is also an elliptic curve operation, where $H(publicKey)$ is the hash value of the public key, which is directly fetched from the cache in the code to avoid calculating the hash value again. *keyImage* is the key image in the ring signature.

(d) The function calculates the new verification value

$$val[i + 1] = H(msg, L, R). \tag{5}$$

using L, R, and message *msg*. When the calculation reaches the last element of the ring, the new challenge value will be stored in val_0; otherwise, it is stored in val_{i+1}, causing an array out-of-bounds error.

(e) Finally, the function checks whether the computed challenge value c_0 equals the original verification value $RingSig.c$ in the signature. If $c_0 = RingSig.c$, the signature is considered valid, and the function returns *true*. Otherwise, it returns $false$. Algorithm 3 is a pseudocode representation of the `verify` function.

Algorithm 3 Verify Function

procedure VERIFY(m)
  $ring \leftarrow$ `ringSing.ring`
  $size \leftarrow$ length of $ring.pubkeys$
  $c_0 \leftarrow$ `ringSing.verify`
  $ed25519 \leftarrow$ `ring.ed25519`
  for $i \leftarrow 0$ **to** $size$ **do**
    $l \leftarrow val_i \times G + verify_i \times publicKey_i$
    $r \leftarrow val_i \times H(publiKey_i) + verify_i \times I$
    if $i = size - 1$ **then**
      $verify_0 \leftarrow$ `verifyVale`(ed25519, msg, l, r)
    else
      $verify_{i+1} \leftarrow$ `verifyVale`(ed25519, msg, l, r)
    end if
  end for
  return $ringSing.verify$ is equal to $verify_0$
end procedure

The improved signature algorithm has several improvements compared to the original ed25519 signature algorithm:

Anonymity: The Linkable Ring Signature (LRS) algorithm provides stronger protection for anonymity. In the regular ed25519 signature algorithm, each signature can be explicitly traced back to its generator. However, in the LRS algorithm, a signature only signifies that it was generated by one member of a fixed set, but it cannot determine which specific member generated it.

Linkability: Another advantage of the LRS algorithm is its linkability. This means that if the same user signs the same message twice, the two signatures can be linked together. This feature can be very useful in certain situations, such as preventing double-spending in blockchain systems.

Resistance to Collusion Attacks: The LRS algorithm can also resist collusion attacks. Even if a portion of users attempt to conspire to disrupt the system, they can not determine which user generated a given signature.

Trustless Setup: Linkable Ring Signatures do not require a prior trust setup. Anyone can generate a set of public keys and generate a signature associated with any public key in the group without obtaining permission from any public key owner.

Lastly, to improve performance, we used caching multiple times. For instance, we cached the `hashToCurve` function at the beginning and then directly obtained values during signing and verification, significantly improving performance. We also took advantage of the concurrent operation features provided by the Go language.

3.2. Security Analysis

Linkable Ring Signatures (LRSs) are a cryptographic signature technique with excellent security properties, especially outstanding in maintaining anonymity and immutability. In the context of the ed25519 elliptic curve encryption standard, these key properties are preserved.

Anonymity: The main feature of Linkable Ring Signatures lies in their strong capability to protect anonymity. During the signature process, the signing entity can choose any public key in a "ring" of public keys to sign without revealing the specific executor of the signature. This means that even if a third party observes this signature, it is unable to determine which public key's corresponding private key owner generated the signature. This anonymity property remains effective in environments using the ed25519 signature algorithm.

Immutability: Another core advantage of Linkable Ring Signatures is their immutability. Once the signature is generated, it cannot be altered. Any tampering with the signature or associated message will be immediately identified during the verification process since the signature is calculated from the specific message and corresponding private key. This

immutability is mainly ensured by the ed25519 signature algorithm, which was designed with tampering prevention in mind from the outset.

Next, we will demonstrate the security of the ed25519 ring signature algorithm in detail from both anonymity and immutability perspectives, thus fully elucidating the reliability and security of this algorithm.

Definition 1 (Anonymity)**.** *In the context of Linkable Ring Signatures, as long as the signature is valid, any third-party verifier cannot determine the specific identity of the signer, i.e., they cannot ascertain the specific public key corresponding to the signer.*

Subsequently, we will proceed to demonstrate the anonymity inherent in this algorithm by employing a zero-knowledge proof approach grounded in the Schnorr protocol [21] in the context of the ed25519 Linkable Ring Signature algorithm.

Preliminary Phase: Assume each user (such as Alice) in the Linkable Ring Signature system has a pair of related private (x) and public keys (P) that satisfy $P = x * G$, where G represents the generator of the elliptic curve. When Alice intends to sign a message m, she selects her public key from a "ring" composed of a set of public keys, subsequently chooses a random number u, and calculates two values, namely $R = u * G$ and $I = u * H(P)$, based on u and her public key, where H denotes a hash function and I is the linkable tag (key image).

Commitment Phase: Alice constructs a new hash function H', and takes all public keys, R, I, and message msg as inputs, eventually obtaining a hash value

$$h_0 = H'(P_1, P_2, \ldots, P_n, R, I, msg). \tag{6}$$

Challenge and Response Phase: In this stage, Alice iterates over each public key P_i in the ring. For each public key, Alice generates a random number r_i (when the public key is not her own) and calculates the next hash value

$$h_{i+1} = H'(P_i, R_i, h_i). \tag{7}$$

where

$$R_i = r_i * G + h_i * P_i. \tag{8}$$

As for her own public key P_j, she calculates

$$u = r_j + h_j * x. \tag{9}$$

Verification Phase: The verifier (like Bob) will first use the same hash function H' and recalculate h_0 with the same input, then go through the same iteration process, checking if the final h_n equals the original h_0. If they are equal, it indicates that the signature is valid. In addition, he will also check if the linkable tag (I) has been used before to prevent double-signing [22].

Through the above process, we can see that Alice only used her private key when dealing with her public key; all other public keys serve to obscure the identity of the real signer. Only entities who know the private key can generate valid signatures, but due to the design of the ring signature, outsiders cannot determine which private key associated with a public key generated the signature, thus achieving the anonymity of the signature.

Definition 2 (Unforgeability)**.** *Assume an entity possesses a set of public and private keys (pk, sk), and the signature is generated by the corresponding private key sk, then unforgeability can be defined, and unless the entity knows the private key sk and the information of all participants, they cannot modify the signature or create a forged signature that appears valid.*

In Linkable Ring Signatures, immutability is ensured by the characteristics of its digital signature and the properties of the hash function. We will illustrate this feature from three aspects.

Immutability of Signatures: Let Sig represent the signature generated by the `private key sk`, and Msg represents the original message. The private key sk is only known to the message sender. This means that for any attacker, unless they know the private key sk, they cannot create a valid signature Sig' such that $Sig' = Sig(Msg)$. Any attempt to modify the signature will lead to signature verification failure because $Ver(publicKey, Msg, Sig') = False$, where Ver is the function to verify the signature using the `public key pk`. The Edwards-curve Digital Signature Algorithm (EdDSA) is used, and the chosen specific elliptic curve edwards25519 has good security, which keeps its private key safe. Unless the elliptic curve encryption is cracked, the private key cannot be obtained.

Collision Resistance of Hash Functions: The hash function H transforms any length of input x into a fixed length of output, i.e., $H(x)$. One of the key features of H is collision resistance. This means that for any $x \neq y$, we have

$$H(x) \neq H(y). \tag{10}$$

In this algorithm, we choose the SHA-3 function, which has good security and collision resistance.

Concealment and Anonymity in Linkable Ring Signatures: Assume we have a set of signers $S = s_1, s_2, \ldots, s_n$, and each signer s_i will leave some information I_i on the chain, but nobody knows which information Ii is left by which signer s_i, so attackers cannot determine which part of the information to try to tamper with the signature.

3.3. View Change Analysis

In this section, we will provide a detailed introduction to the consensus algorithm of MRPBFT (Modified Ring-based Practical Byzantine Fault Tolerance), which is an improvement on HotStuff. MRPBFT maintains the efficiency of HotStuff while enhancing its security and fault tolerance. It is mainly divided into two parts: the consensus algorithm and the view change process. Due to the introduction of a multi-leader mechanism and ring signature algorithm, the consensus algorithm and signing process are more complicated than HotStuff, but it has improved the resistance to malicious leaders and the anonymity of participants. As asynchronous view change is adopted, the overall performance does not degrade compared to HotStuff.

3.3.1. Transaction On-Chain Process

This section will exhaustively explicate the details of the MRPBFT algorithm in the process of transaction on-chain, including all the steps and logic of the process.

Step 1: Creating Transaction:

First, the client builds a transaction according to its digital property transaction needs, covering key information such as the sender, recipient, transaction data, hash summary, and timestamp of the transaction.

Step 2: Signing Transaction:

As digital asset transactions involve sensitive customer privacy information, the client needs to digitally sign the transaction with its private key to ensure the safety of the information. In the HotStuff protocol, ECDSA is usually used for signing, while in MRPBFT, we use the chain ring signature technology based on BLS12 to ensure the safety of user privacy and verify the authenticity and integrity of the transaction.

Step 3: Broadcasting Transaction:

Subsequently, the client broadcasts the signed transaction to the nodes m1, m2, and m3 in the network. Then, the m1 primary node broadcasts the signed message to all secondary nodes. The transmission mode selected by MRPBFT is the same as PBFT and HotStuff, which are all broadcast, not the point-to-point transmission based on Gossip protocol [23], because the Gossip protocol may bring uncertainty and delay to information propagation.

Step 4: Transaction Verification:

The secondary nodes receiving the signed information will verify it, including verifying the transaction's signature and the timestamp of the transaction information.

Step 5: Transaction Buffer Pool: Transactions verified will be added to the transaction buffer pool by the primary nodes m1, m2, and m3. In this pool, transactions are arranged in the order of timestamps, waiting for subsequent processing.

Step 6: Block Packaging:

The primary node m1 organizes the arranged transaction information, and the selected transactions are combined into an aggregated transaction. Then, a chain ring signature is generated for this aggregated transaction, and then the aggregated transaction and chain ring signature are added to the block, forming a QC Tree (Quorum Certificate Tree) [24].

Step 7: Block Broadcasting:

Node m1 sends the packaged block to other secondary nodes and the other two primary nodes through broadcasting.

Step 8: Block Verification:

The secondary nodes receiving the new block will verify the block, including verifying the block's hash value and the legality of the transactions, etc. The primary nodes m2 and m3 will also verify and back up after receiving the block.

Step 9: Block Addition:

The block that passes verification will be added to the node's blockchain, and the block height of the node will be updated; that is, the height of the blockchain will be increased by 1, and the local state of the node will be updated.

Step 10: Transaction Confirmation:

Once a block is confirmed and added to their blockchains by the majority of nodes, the transaction is considered confirmed and irreversible. This transaction on-chain process can be expressed as a function

$$T = f(C, S). \tag{11}$$

where T is the final confirmed transaction, C is the transaction created and signed by the client, and S is the system composed of nodes in the network. This function represents the final transaction confirmation process given the client's transaction and the system.

3.3.2. View Change Process

In this section, we will detail the view change process in the MRPBFT algorithm. Compared to the view change process in the HotStuff consensus algorithm, our main difference lies in the introduction of an asynchronous confirmation mechanism. Here is a detailed description of the view change process in this algorithm.

a. Preparation Phase

1. The primary node m1 collects new view change requests and broadcasts them to each follower node.
2. Upon receiving a new view change request, the follower nodes verify it to confirm its validity and legality. If the received request satisfies the condition

$$Req\_Valid = R | H(R) \in V. \tag{12}$$

where R is the new view change request, H is the hash function, and V is the set of valid hash requests.

b. Proposal Phase

1. The primary node m1 chooses a new view number

$$VN\_new = VN\_old + 1. \tag{13}$$

and generates a view change proposal P based on this.

2. The primary node m1 broadcasts the proposal P to other nodes and then enters a waiting state, waiting for feedback from follower nodes.

c. Asynchronous Confirmation Phase

Upon receiving the view change proposal P, the follower nodes perform the following steps:

1. Verify the proposal's validity and legality, i.e.,

$$Proposal\_Valid = (P|H(P) \in P\_set). \tag{14}$$

where `P_set` is the set of valid proposals.

2. If the follower node accepts the proposal and agrees to switch to the new view, it sends an asynchronous confirmation message `Msg_Async_Confirm` to the primary node m1.
3. If the node refuses the proposal or has not yet made a decision, it does not send an asynchronous confirmation message.

d. View Switching Phase

1. The primary node m1 collects asynchronous confirmation messages and determines whether a sufficient number of asynchronous confirmations have been reached according to the rule

$$Confirm\_Async >= 2f + 1. \tag{15}$$

where f represents the possible number of faulty nodes.

2. If the primary node collects $2f + 1$ asynchronous confirmation messages, i.e., if (15) is satisfied, it begins the view switch.
3. The view switch includes updating local view information, i.e.,

$$viewNumber = viewNumber + 1$$

and resetting the consensus state.

e. Asynchronous Confirmation Reply Phase

The primary node m1 sends an asynchronous confirmation reply message to the nodes that have confirmed the switch to the new view, notifying them that the view switch has been successful, and syncs this information with nodes m2 and m3. By introducing asynchronous mechanisms, this improved view process allows nodes to start a new view without reaching a global consensus. This can enhance consensus efficiency, reduce the waiting time for view changes, and allow the system to continue processing transactions and consensus operations even when some nodes have not yet completed the view change. The Algorithm 4 shows the pseudo-code implemented by the above algorithm.

Algorithm 4 Asynchronous View Switching

```
Preparation Phase:
function CollectViewChangeRequests
    R ← new view change requests
    broadcast (R)
end function
function ValidateRequests
    Req_Valid ← {r|H(r) ∈ V}
end function

Proposal Phase:
function ProposeViewChange
    VN_new ← VN_old + 1
    P ← generate proposal(VN_new)
    broadcast (P)
end function
```

Algorithm 4 *Cont.*

Asynchronous Confirmation Phase:
function HANDLEVIEWCHANGEPROPOSAL
    for each slave node **do**
        $P\_Valid \leftarrow \{p|H(p) \in P\_set\}$
        if P_Valid **then**
            send $Msg\_Async\_Confirm$ to $m1$
        end if
    end for
end function

View Switching Phase:
function HANDLEASYNCCONFIRMATIONS
    $Confirm\_Async \leftarrow$ count$(Msg_Async_Confirm)$
    if $Confirm\_Async \geq 2f + 1$ **then**
        Start view switching
        $viewNumber \leftarrow viewNumber + 1$
        Reset consensus state
    end if
end function

Asynchronous Confirmation Reply Phase:
function SENDASYNCCONFIRMATIONREPLY
    broadcast $Msg\_Async\_Confirm\_Reply$ to $m2$ and $m3$
end function

4. Experimental Design and Results

In this study, we conducted a comprehensive performance evaluation of various consensus algorithms, including our proposed MRPBFT consensus algorithm, the traditional HotStuff protocol, and an optimized version known as FastHotStuff. Our MRPBFT consensus algorithm comprises two main components: the Ed25519LRS signature algorithm and the MPBFT consensus mechanism. We implemented these algorithms using the Go programming language (version 1.8) and performed all experiments on a Mac M1 computer with 8 GB of memory. The GoLand tool was used for experimental purposes, and SHA256 was employed for hash calculations. During the experimental phase, we systematically increased the number of nodes from 4 to 128. We utilized various signature algorithms, including BLS112 aggregate signatures, ECDSA signatures, Ed25519 signatures, and our custom Ed25519LRS signature algorithm. In this series of experiments, we assessed the throughput and network latency of the consensus algorithms individually.

In the throughput tests, as presented in Table 2, we combined the ECDSA signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. The objective was to assess the performance of our enhanced consensus mechanism when paired with the ECDSA signature algorithm and compare it against the three original consensus mechanisms. Table 3, on the other hand, showcases the results of our evaluation with the BLS12 signature algorithm. In this set of experiments, we integrated the BLS12 signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. This evaluation aimed to examine the throughput performance of our improved consensus mechanism when utilized alongside the BLS12 signature algorithm and compare it to the original three consensus mechanisms. In Table 4, we utilized the Ed25519 signature algorithm and paired it with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. We conducted these experiments to assess the throughput performance of our enhanced consensus mechanism when coupled with the Ed25519 signature algorithm and to draw comparisons against the three original consensus mechanisms. Finally, Table 5 presents the outcomes of our evaluation with the custom

Ed25519LRS signature algorithm. We combined the Ed25519LRS signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT to evaluate the throughput performance of our enhanced consensus mechanism under the Ed25519LRS signature algorithm, and we compared these results to the three original consensus mechanisms.

Table 2. The ecdsa sign with different consensus algorithm throughput.

Node Num	SimpleHotStuff–tps/s	FastHotStuff–tps/s	ChainedHotStuff–tps/s	MPBFT–tps/s	Sign
4	1792	17,838	18,246	19,752	ecdsa
8	7126	7109	6994	8432	ecdsa
16	2688	2715	2642	3104	ecdsa
32	870	748	686	1053	ecdsa
64	486	431	401	653	ecdsa
128	301	267	231	367	ecdsa

Table 3. The bls12 sign with different consensus algorithm throughput.

Node Num	SimpleHotStuff–tps/s	FastHotStuff–tps/s	ChainedHotStuff–tps/s	MPBFT–tps/s	Sign
4	1279	1274	1189	1345	bls12
8	892	873	812	932	bls12
16	516	532	478	579	bls12
32	338	325	267	351	bls12
64	NA	NA	NA	NA	bls12
128	NA	NA	NA	NA	bls12

"NA" indicates that when using the BLS signature algorithm, timeouts occur when the number of nodes exceeds 32.

Table 4. The ted25519 sign with different consensus algorithm throughput.

Node Num	SimpleHotStuff–tps/s	FastHotStuff–tps/s	ChainedHotStuff–tps/s	MPBFT–tps/s	Sign
4	19,129	18,815	18,636	19,908	ed25519
8	7374	7315	7320	7902	ed25519
16	2828	2723	2850	3320	ed25519
32	921	902	898	1309	ed25519
64	517	501	476	703	ed25519
128	311	301	247	374	ed25519

Table 5. The ed25519LRS sign with different algorithm throughput.

Node Num	SimpleHotStuff–tps/s	FastHotStuff–tps/s	ChainedHotStuff–tps/s	MPBFT–tps/s	Sign
4	13,782	13,581	13,317	15,488	ed25519LRS
8	7193	6971	6731	7752	ed25519LRS
16	2709	2658	2764	3864	ed25519LRS
32	905	897	879	1928	ed25519LRS
64	502	489	431	960	ed25519LRS
128	287	268	208	489	ed25519LRS

In the latency tests, as presented in Table 6, we combined the BLS12 signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. The objective was to assess the latency of our improved consensus mechanism when paired with the BLS12 signature algorithm and compare it against the latency of the three original consensus mechanisms. Moving on to Table 7, we conducted experiments using the ECDSA signature algorithm. We integrated the ECDSA signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. This evaluation aimed to examine the latency performance of our enhanced consensus mechanism when coupled with the ECDSA signature algorithm and draw comparisons against the latency of the three original consensus mechanisms. In Table 8, we employed the Ed25519 signature algorithm and combined it with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT. The purpose was to evaluate the latency of our improved consensus mechanism when paired with the Ed25519 signature algorithm and compare it to the latency of the three original consensus mechanisms. Finally, Table 9 presents the outcomes of our evaluation with the custom Ed25519LRS signature algorithm. We combined the Ed25519LRS signature algorithm with the consensus mechanisms of SimpleHotStuff, FastHotStuff, ChainedHotStuff, and our improved MPBFT to evaluate the latency performance of our enhanced consensus mechanism under the Ed25519LRS signature algorithm and compare it against the latency of the three original consensus mechanisms.

Table 6. The bls12 sign with different consensus latencies.

Node Num	FastHotStuff–lat/ms	ChainedHotStuff–lat/ms	SimpleHotStuff–lat/ms	MPBFT–lat/ms	Sign
4	46.8	51.2	78.1	39.6	bls12
8	65.1	71.2	111.2	59.7	bls12
16	110.7	120.4	187.9	99.8	bls12
32	230.1	265	310.5	209	bls12
64	NA	NA	NA	NA	bls12
128	NA	NA	NA	NA	bls12

Table 7. The ecdsa sign with different consensus latencies.

Node Num	FastHotStuff–lat/ms	ChainedHotStuff–lat/ms	SimpleHotStuff–lat/ms	MPBFT–lat/ms	Sign
4	3.2	5.4	5.8	2.4	ecdsa
8	14	14.1	14	8.9	ecdsa
16	37	35.8	37.7	27.5	ecdsa
32	113.2	103.5	103.7	87.7	ecdsa
64	231	201	197.2	165.8	ecdsa
128	501.3	432.1	399.5	318.2	ecdsa

Table 8. The ed25519 sign with different consensus latencies.

Node Num	FastHotStuff–lat/ms	ChainedHotStuff–lat/ms	SimpleHotStuff–lat/ms	MPBFT–lat/ms	Sign
4	5.2	5.6	5.17	4.67	ed25519
8	13.56	13.9	13.2	12.5	ed25519
16	35.6	34.7	34.9	31.8	ed25519
32	109.4	104.9	175.1	97.6	ed25519
64	221.5	207.8	289.8	187.2	ed25519
128	438.6	401.1	501.3	356.3	ed25519

Table 9. The ed25519LRS sign with different consensus latencies.

Node Num	FastHotStuff–lat/ms	ChainedHotStuff–lat/ms	SimpleHotStuff–lat/ms	MPBFT–lat/ms	Sign
4	6.8	7.2	6.3	5.6	ed25519LRS
8	16.1	17.1	14.8	13.5	ed25519LRS
16	35.2	37.6	35.3	32.8	ed25519LRS
32	108.3	110.3	107.1	82.6	ed25519LRS
64	236.6	228.8	291.5	145.9	ed25519LRS
128	441.2	415.3	517.4	303.4	ed25519LRS

5. Discussion

In this section, we will delve deeper into the experimental results presented above and compare them with relevant work in the field.

Throughput Comparison: Our experimental results (as shown in Tables 2–5) reveal that when tested with the ECDSA signature algorithm and four consensus algorithms, the MPBFT consensus mechanism outperforms the others. As depicted in Table 2, MPBFT achieves the highest throughput, surpassing SimpleHotStuff by 20% at 128 nodes and demonstrating nearly a 50% improvement over ChainedHotStuff. In the case of the BLS12 signature algorithm (Table 3), although its performance is suboptimal across all four consensus mechanisms and results in latency timeouts beyond 64 nodes, MPBFT remains the top performer for configurations with fewer than 32 nodes. In the context of the Ed25519 signature algorithm (Table 4), MPBFT achieves impressive throughput, nearly reaching 20,000 transactions per second with only four nodes, making it the leading choice among the four consensus mechanisms. When employing our custom Ed25519LRS signature algorithm (Table 5), MPBFT still maintains its superiority. However, it is notable that consensus algorithms based on the Ed25519LRS signature algorithm exhibit lower throughput than ECDSA and Ed25519 for configurations with fewer than 16 nodes. This is an expected outcome as our Ed25519LRS signature algorithm introduces anonymity and tamper resistance, increasing the overall complexity of the signature algorithm. Nevertheless, beyond 16 nodes, our algorithm surpasses ECDSA and Ed25519 signature algorithms in throughput and consistently performs best throughout the throughput tests (as illustrated in Figure 1). This can be attributed to the fact that, as the number of nodes increases, the communication overhead between nodes becomes more significant than the impact of the signature algorithm on consensus algorithm throughput.

Latency Comparison: Our experimental results (as presented in Tables 6–9) consistently demonstrate that, among the four signature algorithms, MPBFT consensus exhibits the best performance in terms of latency. As evident in Table 6, when using the BLS12 aggregate signature, it registers the highest latency among all signature algorithms, and timeouts occur when the node count exceeds 32. This aligns with our earlier throughput test results, where higher throughput generally leads to lower latency. In contrast, when utilizing the ECDSA signature algorithm (as indicated in Table 7), our MPBFT consensus mechanism achieves the lowest latency, reaching as low as 2.4 milliseconds, making it the top-performing option in the entire set of experiments. In latency testing with the Ed25519 signature algorithm and various consensus mechanisms (as shown in Table 8), MPBFT continues to outperform, exceeding SimpleHotStuff by nearly 10% with four nodes. When employing our custom Ed25519LRS signature algorithm alongside the four consensus mechanisms (as demonstrated in Table 9), our signature algorithm delivers the best performance. It achieves latency as low as 5.6 ms, and notably, beyond 16 nodes, our Ed25519LRS signature outperforms other signature algorithms and consensus mechanisms within our MPBFT. This advantage is further emphasized in Figure 2, where it is evident that our performance excels, particularly as the number of nodes increases.

Shortcomings: Despite the numerous improvements we have made, there are still some limitations that need to be addressed. These include enhancing system scalability and fault tolerance, as well as finding more effective solutions for network latency and packet loss issues. Additionally, it is worth noting that our experiments were conducted in a local environment and did not involve testing in a real network setting. While our research represents significant progress compared to prior achievements, there are areas for improvement and challenges that warrant exploration and resolution. These include outlining strategies for further enhancing system scalability and fault tolerance and devising more efficient methods to address network latency and packet loss problems. These aspects will form the focus of our future research efforts.

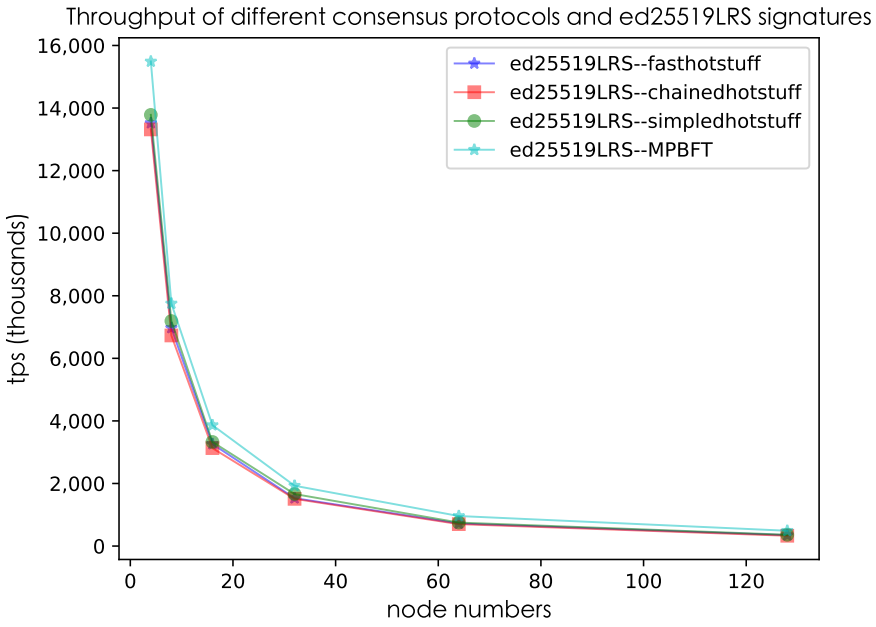

Figure 1. Throughput of different consensus protocols and ed25519LRS signatures.

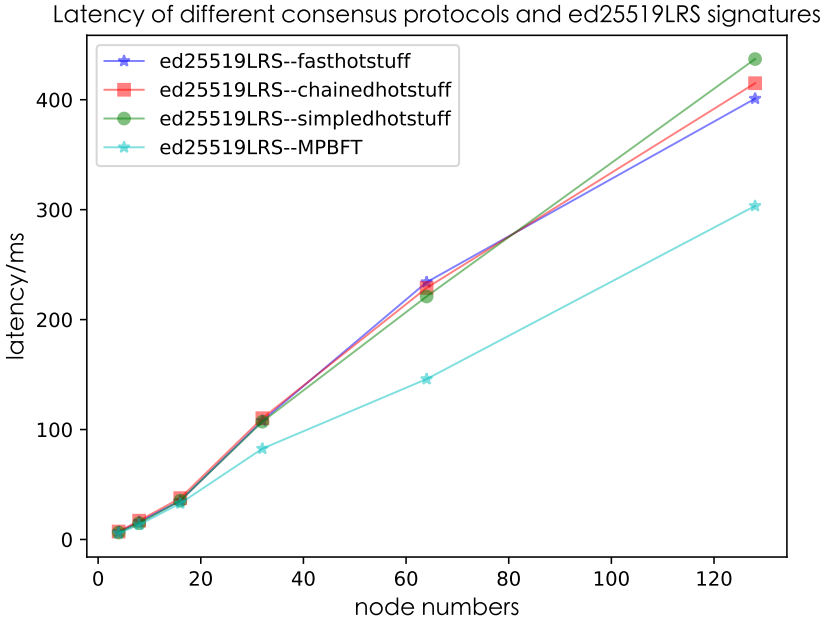

Figure 2. Latency of different consensus protocols and ed25519LRS signatures.

6. Evaluation

In this paper, we have introduced an improved HotStuff consensus algorithm called MRPBFT. MRPBFT enhances the efficiency of the consensus algorithm while ensuring anonymity and tamper resistance. We began by analyzing the principles of the Ed25519RLS signature algorithm and then discussed the security aspects of the signature algorithm, focusing on both anonymity and tamper resistance. Furthermore, we improved the consensus mechanism within HotStuff by introducing a multi-principal node model, which enhances system dynamics and scalability, ultimately improving consensus efficiency. The asynchronous view change mechanism further accelerates consensus reaching speed, enhancing system throughput and responsiveness.

In the future, our research will continue to focus on the MRPBFT consensus algorithm. We aim to further enhance system scalability and fault tolerance, conduct testing in real network environments, and apply the algorithm to digital asset transactions.

Author Contributions: Conceptualization, M.G.; Methodology, M.G.; Investigation, M.G.; Data curation, Z.W.; Writing—original draft, M.G.; Writing—review & editing, G.L.; Supervision, G.L.; Project administration, G.L.; Funding acquisition, G.L. All authors have read and agreed to the published version of the manuscript.

Funding: This paper is financially supported by the project of Research and Application demonstration of key Technologies of Yunnan Autonomous controllable Blockchain basic Service platform. (Grant no, 202102AD080006).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2008**, *4*, 1–14.
2. Gervais, A.; Karame, G.O.; Wüst, K.; Glykantzis, V.; Ritzdorf, H.; Capkun, S. On the security and performance of proof of work blockchains. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 3–16.
3. Yang, J.; Paudel, A.; Gooi, H.B. Compensation for power loss by a proof-of-stake consortium blockchain microgrid. *IEEE Trans. Ind. Inform.* **2020**, *17*, 3253–3262. [CrossRef]
4. Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
5. Castro, M.; Liskov, B. Practical byzantine fault tolerance. In Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, USA, 22–25 February 1999; pp. 173–186.
6. Yin, M.; Malkhi, D.; Reiter, M.K.; Gueta, G.G.; Abraham, I. HotStuff: BFT consensus in the lens of blockchain. *arXiv* **2018**, arXiv:1803.05069.
7. Bacho, R.; Loss, J. On the adaptive security of the threshold BLS signature scheme. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 193–207.
8. Wahby, R.S.; Boneh, D. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *Cryptol. ePrint Arch.* **2019**, *4*, 154–179. [CrossRef]
9. Johnson, D.; Menezes, A.; Vanstone, S. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Secur.* **2001**, *1*, 36–63. [CrossRef]
10. Buterin, V.; Griffith, V. Casper the friendly finality gadget. *arXiv* **2017**, arXiv:1710.09437.
11. Buchman, E. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. 2016. Available online: https://api.semanticscholar.org/CorpusID:59082906 (accessed on 6 November 2023).
12. Baudet, M.; Ching, A.; Chursin, A.; Danezis, G.; Garillot, F.; Li, Z.; Malkhi, D.; Naor, O.; Perelman, D.; Sonnino, A. State Machine Replication in the Libra Blockchain. *Libra Assoc. Tech. Rep.* **2019**, *1*, 1–19.
13. Jalalzai, M.; Niu, J.; Feng, C.; Gai, F. Fast-HotStuff: A fast and robust BFT protocol for blockchains. *IEEE Trans. Dependable Secur. Comput.* **2023**. [CrossRef]
14. Shoup, V. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, 14–18 May 2000 Proceedings 19*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 207–220.
15. Na, Y.; Wen, Z.; Fang, J.; Tang, Y.; Li, Y. A derivative PBFT blockchain consensus algorithm with dual primary nodes based on separation of powers-DPNPBFT. *IEEE Access* **2022**, *10*, 76114–76124. [CrossRef]
16. Bisheh-Niasar, M.; Azarderakhsh, R.; Mozaffari-Kermani, M. Cryptographic accelerators for digital signature based on Ed25519. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2021**, *29*, 1297–1305. [CrossRef]

17. Boneh, D.; Gentry, C.; Lynn, B.; Shacham, H. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology—EUROCRYPT 2003: International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, 4–8 May 2003 Proceedings 22*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 416–432.
18. Noether, S.; Mackenzie, A. Ring confidential transactions. *Ledger* **2016**, *1*, 1–18. [CrossRef]
19. Giridharan, N.; Howard, H.; Abraham, I.; Crooks, N.; Tomescu, A. No-Commit Proofs: Defeating Livelock in BFT. Cryptology ePrint Archive, Paper 2021/1308. 2021. Available online: https://eprint.iacr.org/2021/1308 (accessed on 23 July 2023).
20. Yang, J.; Jia, Z.; Su, R.; Wu, X.; Qin, J. Improved Fault-Tolerant Consensus Based on the PBFT Algorithm. *IEEE Access* **2022**, *10*, 30274–30283. [CrossRef]
21. Maxwell, G.; Poelstra, A.; Seurin, Y.; Wuille, P. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptogr.* **2019**, *87*, 2139–2164. [CrossRef]
22. Poettering, B.; Stebila, D. Double-authentication-preventing signatures. *Int. J. Inf. Secur.* **2017**, *16*, 1–22. [CrossRef]
23. Dunbar, R.I.M. Gossip in evolutionary perspective. *Rev. Gen. Psychol.* **2004**, *8*, 100–110. [CrossRef]
24. Lakshmanan, L.V.S.; Pei, J.; Zhao, Y. QC-Trees: An efficient summary structure for semantic OLAP. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; pp. 64–75.