Peer-to-Peer Networking and Applications (2025) 18:0
https://doi.org/10.1007/s12083-024-01830-8

Parallel Byzantine fault tolerance consensus based on trusted execution environments

Ran Wang[1,5] · Fuqiang Ma[2,3,4] · Sisui Tang[1] · Hangning Zhang[1] · Jie He[1] · Zhiyuan Su[4] · Xiaotong Zhang[1] · Cheng Xu[1,6]

Received: 20 June 2024 / Accepted: 17 October 2024

Abstract
Ensuring consistency and reliability in distributed systems is crucial for their adoption. Blockchain technology offers a robust framework for these systems, characterized by decentralization, immutability, auditability, and traceability. In public blockchains, consensus algorithms like Proof-of-Work (PoW) are foundational for securing transactions but are criticized for high energy consumption and limited transaction throughput, making them less suitable for high-frequency environments. In contrast, consortium blockchains rely on Byzantine Fault Tolerance (BFT) algorithms to address these issues. However, existing Practical Byzantine Fault Tolerance (PBFT) protocols still face challenges in performance, scalability, and fault tolerance. This paper introduces a novel Parallel Byzantine Fault Tolerance protocol, TEP-BFT, leveraging Trusted Execution Environments. The TEP-BFT protocol utilizes a Unique Sequential Identifier Generator (USIG) based on Intel Software Guard Extensions (Intel SGX) to generate unique identifiers, thus ensuring the monotonicity, uniqueness, and order of messages. This innovation reduces the requisite number of communication phases and replicas, substantially enhancing the efficiency and fault tolerance of the consensus process. Moreover, the protocol implements parallel processing strategies both inter-thread and intra-thread to augment the throughput of the blockchain system significantly. Our experimental and performance analysis indicates that TEP-BFT achieves an optimal balance among performance, scalability, and fault tolerance, surpassing other BFT protocol variants. This advancement positions TEP-BFT as a superior choice for blockchain systems in scenarios requiring rapid and frequent transaction processing, marking a significant step forward in the evolution of blockchain consensus mechanisms. Our code is made public available at: https://github.com/SICC-Group/TEP-BFT.git.

Keywords Byzantine fault tolerance · Consensus mechanisms · Trusted execution environments · Blockchain

1 Introduction

As distributed systems become increasingly prevalent in today's digital era, ensuring their consistency and reliability has emerged as a crucial challenge. The swift advancement of blockchain technology provides a reliable foundation for distributed systems, characterized by decentralization, immutability, auditability, and traceability [31]. However, securing transactions on the blockchain to reach consensus is challenging. For example, in blockchain systems, increasing the number of nodes enhances decentralization and improves resistance to attacks. However, this also raises the complexity and communication costs of reaching the consensus, reducing overall efficiency. Balancing system efficiency with security becomes a significant challenge. Additionally, addressing Byzantine Fault Tolerance, where some nodes may act maliciously or fail, and preventing double-spending, where digital assets could be reused in multiple transactions, are critical. These challenges underscore the need for a robust consensus mechanism to ensure reliable operation in complex and dynamic environments.

The Proof of Work (PoW) is a well-known consensus protocol for public blockchains, where consensus is achieved through computational work required to solve cryptographic puzzles, ensuring that all participating nodes agree on the valid state of the ledger. Bitcoin's blockchain relies on the PoW mechanism to ensure transactions are executed in the correct order [10]. Similarly, Proof of Stake (PoS), which has gained prominence after Ethereum's transition from PoW to PoS in 2022 [14], is another consensus mechanism widely used in public blockchains. While PoS addresses some of the

This article belongs to the Topical Collection:*3 - Track on Blockchain*
Guest Editor: Haojin Zhu

Extended author information available on the last page of the article energy inefficiencies of PoW, it introduces challenges such as the need for robust economic incentives and potential centralization risks due to stake accumulation.

However, the consensus requirements in consortium blockchains (or permissioned blockchains) differ significantly from those in public blockchains [32]. Consortium blockchains are designed for environments with pre-approved participants, where trust and accountability are often established among the nodes. In such scenarios, PoW and PoS protocols are not well-suited due to their inefficiencies and reliance on economic incentives, which are unnecessary in permissioned settings. Instead, Byzantine Fault Tolerance (BFT) protocols, which are specifically designed to handle fault tolerance and malicious nodes within a smaller, trusted network, have become the preferred consensus mechanism for consortium blockchains [22].

BFT protocols offer higher efficiency and lower latency compared to PoW and PoS for consortium blockchains, making them ideal for applications in sectors such as finance and industrial IoT, where frequent transactions and high throughput are essential [13, 21, 34]. While BFT protocols have long been fundamental to distributed systems [2, 3, 6, 17, 18], recent research has focused on optimizing these protocols for blockchain use cases, particularly in consortium settings. These optimizations are critical in enhancing the security and reliability of blockchain systems by preventing malicious nodes from disrupting the consensus process. For example, IBM's Hyperledger Fabric blockchain [11] and the open-source consortium FISCO BCOS blockchain [1] rely on Practical Byzantine Fault Tolerance (PBFT) [3] to achieve consensus, showcasing the practical application of these optimized BFT protocols in consortium blockchains.

Taking the PBFT protocol as an example, PBFT achieves higher throughput than Bitcoin's consensus layer [30], it cannot match the transaction volume of existing payment systems [16]. Moreover, PBFT requires 3f+1 replicas, which must be diverse (different operating systems, software) to tolerate attacks and intrusions, thus increasing the extra costs (hardware, software development, management, etc.) associated with more replicas [9, 27]. Additionally, PBFT is only scalable to a limited number of nodes as it requires the exchange of $O(n^2)$ messages among n servers to reach consensus [3]. Therefore, enhancing the scalability and performance of BFT protocols, as well as reducing the configuration costs of replicas, is crucial for their practical deployment in existing industrial blockchain solutions.

To address the issues with traditional BFT protocols, many improved BFT protocols have been proposed and studied in recent years [15, 18, 23, 29]. For example, the FastBFT protocol proposed by Liu et al. [23] balances computational and communication loads by arranging nodes in a tree topology, facilitating inter-server communication and message aggregation at the tree's edges. FastBFT adopts an optimistic BFT paradigm [23], requiring only a portion of active nodes to participate in the consensus. However, FastBFT's operation relies on a relatively stable cluster environment, as malicious nodes can intentionally trigger member replacement in the tree topology communication. Most existing research attempts to improve consensus throughput and latency by reducing communication phases or adopting asynchronous protocols but fails to organically combine consensus communication phases, intra-thread asynchrony, and inter-thread asynchrony [23, 33]. Considering the above limitations, we introduce a more efficient and flexible consensus protocol that not only reduces communication phases through trusted counters but also supports parallel operations within and between consensus threads.

This paper proposes a novel Parallel Byzantine Fault Tolerance protocol based on Trusted Execution Environments (TEP-BFT) aimed at addressing the issues present in existing research. In our protocol, to enhance security and improve consensus efficiency, the TEP-BFT protocol leverages Intel SGX's Trusted Execution Environment to secure the confidentiality and integrity of messages broadcasted by nodes, preventing the transmission of erroneous message packets and thereby reducing one consensus communication phase. Additionally, TEP-BFT has designed multi-tiered parallel processing operations, further enhancing the throughput of the BFT algorithm and reducing latency. This improvement is particularly evident in scenarios requiring high-frequency transactions, as demonstrated by our performance evaluations. For example, in an Industrial IoT setting, where devices constantly generate data that needs to be validated and recorded on a blockchain, the TEP-BFT protocol excels due to its ability to process multiple consensus operations in parallel. Specifically, the main contributions are summarized as follows:

1. We propose a new Byzantine Fault Tolerance consensus protocol, TEP-BFT, that fully utilizes Intel SGX-based Unique Sequential Identifier Generator (USIG) to generate unique identifiers, ensuring the monotonicity, uniqueness, and orderliness of messages. This reduction in communication phases during the consensus process enhances consensus efficiency. Additionally, TEP-BFT only requires the participation of $2f + 1$ nodes in the consensus process to withstand attacks from f Byzantine nodes, effectively saving the system's cost of tolerating intrusions.
2. TEP-BFT introduces multi-level parallel processing, not only supporting parallel processing between transaction packaging and consensus threads but also within consensus threads themselves. This includes parallel operations within the block batch parallel production phase and between the block pipeline execution phase. Multi-level

parallel processing further increases the blockchain system's throughput.

3. To validate the enhanced performance of the optimized consensus protocol, we conducted practical deployment tests on physical machines. Experiment results indicate that TEP-BFT surpasses other PBFT protocol variants in performance, scalability, and fault tolerance. This confirms the effectiveness of TEP-BFT in real-world scenarios, demonstrating its superior operational capabilities.

The structure of this paper is as follows: Section 2 details existing Byzantine Fault Tolerance consensus protocols and their limitations. Section 3 introduces the framework of the TEP-BFT protocol. Section 4 describes the design and working mechanism of the TEP-BFT protocol, including the USIG service based on Intel SGX, the normal operation process, and view switching. Section 5 demonstrates the security and liveliness of the TEP-BFT protocol. Section 6 presents the performance advantages of TEP-BFT through experiments and performance analysis. Finally, Section 7 summarizes the main contributions of this paper and future research directions.

2 Preliminary

2.1 Byzantine consensus

As the application demand for blockchain technology continuously increases, consensus mechanisms have become a focal point of research in both the academic and industrial sectors. This has led to a renewed exploration of effective Byzantine consensus solutions. The main idea behind Byzantine consensus protocols is that the system can continue to operate correctly even if some of its components behave maliciously [29]. Byzantine consensus protocols typically require $3f+1$ nodes to tolerate f Byzantine (or faulty) nodes. Their fundamental concept is that the benign nodes can overcome Byzantine (or faulty) ones through a series of votes [35].

One of the well-known BFT consensus protocols is the PBFT protocol [3]. The core concepts in the PBFT protocol include three parts: View, Nodes, and Roles (Client, Primary, Replica), as shown in Fig. 1. The view represents the current global state of the system. The PBFT protocol also accommodates the aforementioned capability to tolerate f malicious nodes, with a total node count of $3f+1$. The process of the protocol is primarily divided into five phases: Request, Pre-prepare, Prepare, Commit, and Reply. Following a client's request, the primary node assigns a sequence number to the request and sends the message to replica nodes during the Pre-prepare phase. If the nodes agree with the Pre-prepare message, they send their messages to other nodes during the Prepare phase. Each node checks the correctness of the message and the number of Prepare messages it has received (at least $2f$). After completing the Prepare phase, nodes send confirmation messages to other nodes. Nodes verify the data and correctness of the confirmation messages received and send a reply message to the client. If the client receives more than $f+1$ identical reply messages, the request is considered complete; otherwise, a new request is initiated.

2.2 Enhancing Byzantine fault tolerance

In traditional PBFT, consistency has been associated with low efficiency and high costs. Consequently, some research efforts have sought to reduce the number of nodes and communication phases to improve the performance of consensus. We categorize and summarize the existing work as shown in Table 1. These research works can be categorized into four types:

1. *Speculative protocols*: Kotla et al. proposed Zyzzyva [18], which utilizes speculation to enhance performance. Unlike traditional PBFT, replicas in Zyzzyva execute client requests in the order proposed by the primary node, without the need for any explicit consensus protocol. However, under normal operating conditions, Zyzzyva can reduce the overhead of state machine replication to nearly optimal levels. However, once the primary node makes an error, it must switch to the PBFT view change process, which does not offer a clear advantage against attacks from Byzantine nodes.

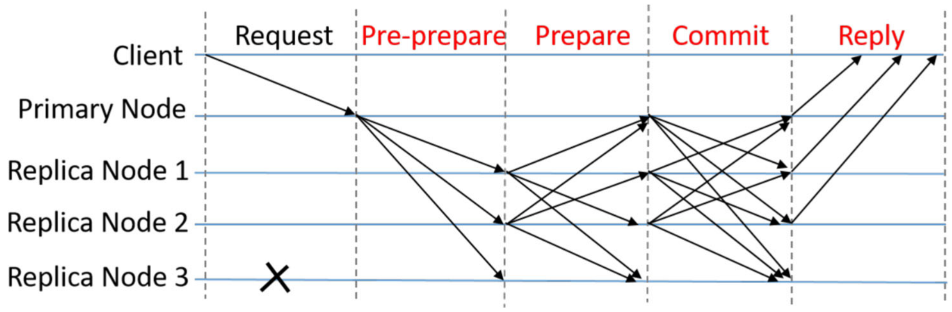


Fig. 1 The protocol process of traditional PBFT consensus. It is consist of four phases: *pre-prepare*, *prepare*, *commit*, and *reply*. Assuming there are f Byzantine nodes, participation from $3f+1$ nodes is necessary to maintain liveness. Among these, Server3 represents a Byzantine or faulty node and does not participate in the consensus process

Table 1 Comparison of BFT protocols

Protocol Type	Protocol	Key Features	Strengths	Limitations
Speculative	Zyzzyva (Kotla et al.)	Speculative execution, relies on primary node	Reduces overhead of state machine replication, nearly optimal performance under normal conditions	Reverts to PBFT view change process upon primary node error, lacks clear advantage against Byzantine attacks
Optimistic	ReBFT (Distler et al.)	Resource-efficient, passive replica updates	Subset of replicas run consensus protocol, improved resource efficiency	Efficiency significantly reduced under malicious attacks, similar trade-offs as speculative protocols
Asynchronous	HotStuff (Yin et al.)	Linear leader changes, CPU pipelining, O(n) complexity	Parallel processing, improved scalability, lower communication complexity	Three rounds of interaction before commitment, adds communication delay
Asynchronous	HoneyBadgerBFT (Miller et al.)	No specific primary node, blocks prevent duplication	Higher efficiency through block proposals, fault tolerance	High latency, especially over wide-area networks, delays can exceed one minute
Hardware Security	MinBFT (Veronese et al.)	Trusted counter for sequence numbers, tamper-resistant	Reduces communication phases from three to two, TEE guarantees monotonicity	Requires separate TEE counter, lacks trustworthy authentication mechanism between nodes
Hardware Security	MinZyzzyva (Veronese et al.)	Uses TEE to reduce replicas, maintains communication phases	Reduces replica count, maintains Zyzzyva's performance benefits	Similar limitations as MinBFT, reliance on TEE counter
Hardware Security	CheapBFT (Kapitza et al.)	Optimistic protocol with TEE, transitional protocol for faults	$f+1$ active replicas for consensus, efficient in fault-free scenarios	Complex transition between consensus protocols, increased complexity of BFT programming model

2. *Optimistic protocols*: Distler et al. [7] introduced a resource-efficient BFT (ReBFT) replication architecture. Under normal operating conditions, only a subset of replicas needs to run the consensus protocol, with others passively updating their state and only actively participating if the consensus protocol fails. BFT protocols that follow this message pattern are termed optimistic BFT. However, this improved method shares the same drawbacks as speculative approaches, trading off security for efficiency. Once there is an attack by malicious nodes, the efficiency of this method will be significantly reduced.
3. *Asynchronous protocols*: The main contribution of HotStuff, proposed by Yin et al. [33], is the implementation of linear leader changes. Using the concept of CPU pipelining, it designs the three communication phases to be identical in message sending and receiving, allowing for parallel processing of different communication phases for multiple transactions. This mechanism reduces the communication complexity of the BFT protocol to O(n), improving scalability. However, proposals in this protocol require three rounds of interaction before they can be committed, adding extra communication delay.
   Based on the above challenges, Miller et al. [26] proposed HoneyBadgerBFT, an asynchronous consensus protocol devoid of a specific primary node. To enhance efficiency, nodes package a subset of transactions into proposal blocks to prevent duplication. However, due to the iterative nature of asynchronous BFT consensus mechanisms, achieving a final consensus result incurs higher latency, particularly noticeable over wide-area networks, where delays can exceed one minute in certain scenarios.
4. *protocols Based on Hardware Security Mechanisms*: Hardware security mechanisms have been widely integrated into general computing platforms [8]. Trusted Execution Environments (TEEs) are prevalent on mobile platforms, and newer TEEs, such as Intel's SGX [12, 25], offer protected memory and isolated execution, preventing standard operating systems or applications from controlling or observing the data processed within. In essence, TEEs can crash but not exhibit Byzantine faults.

Several studies have explored using hardware security to reduce the number of replicas and/or communication phases in BFT protocols [4, 5, 15, 19, 28, 29]. For instance, MinBFT [29] utilizes a trusted counter within the primary node to assign sequence numbers to client requests. This tamper-resistant component does more than just assign a number; it generates a signed certificate that explicitly associates the number with a specific message, and not any other, leveraging the TEE's guarantee of counter monotonicity to prevent assigning the same counter value to different messages. Consequently, the communication phases from three to two. Similarly, MinZyzzyva uses TEEs to decrease the number of replicas required in Zyzzyva, maintaining the same number of communication phases [29].

However, both protocols necessitate maintaining a separate counter within the TEE. Moreover, the USIG service, based on TPM, offers a simple implementation but lacks a trustworthy authentication mechanism among nodes. CheapBFT [15] employs TEEs within an optimistic BFT protocol framework. In fault-free scenarios, CheapBFT only needs $f + 1$ active replicas to achieve consensus and execute client requests. Upon detecting suspicious faults, CheapBFT triggers a transitional protocol to activate passive replicas, switching to MinBFT. Nevertheless, CheapBFT involves transitioning between three distinct consensus protocols, increasing the complexity of the BFT programming model.

For Byzantine fault tolerance protocols, a common latency metric is the number of communication steps required under optimal conditions-without faults and with sufficient system synchronization to avoid primary node changes. MinBFT and MinZyzzyva are highly efficient under this metric, operating with the minimum known communication steps for non-speculative and speculative protocols, respectively four [24] and three steps [18]. However, in cases of node failures or Byzantine behaviors, these protocols may have to roll back some executions, thereby complicating the programming model and incurring additional computational and communication costs.

3 System framework

As shown in Fig. 2, the TEP-BFT protocol incorporates two-stage parallel processing, and each Consensus thread includes a three-phase communication. The following provides an explanation of the nodes, threads, processes, and phases within the TEP-BFT framework:

Nodes The consensus framework mainly comprises three types of nodes: clients, primary nodes, and replica nodes. Clients initiate requests, each possessing a unique pair of public and private keys. The private key is used to sign requests. Primary nodes receive requests from clients, sequence them, and utilize a USIG service based on SGX to generate a unique identifier (UI) for each message request. For a detailed description of the implementation method of USIG based on Intel SGX, please see Section IV-A. Subsequently, they forward these sequenced messages to replica servers and return the results to the clients. Replica nodes receive messages from the primary node, execute the verified operations, and send back the results to the clients. Both primary and replica nodes generate a key pair within a trusted execution environment during system initialization. The private key is securely stored and used within this environment, inaccessible to any external entities, while the public keys are publicly available for verification.

Threads Consensus on a block primarily involves two processes: transaction packaging and consensus. Therefore, a consensus process is composed of the Seal and Consensus thread. The Seal thread is responsible for retrieving transactions from the transaction pool and packaging them based on the highest block on the node, thereby generating new blocks. The new blocks are then handed over to the Consensus thread. The Consensus thread is tasked with receiving new blocks, either locally or through the network, and completing the consensus process based on the received messages, even-

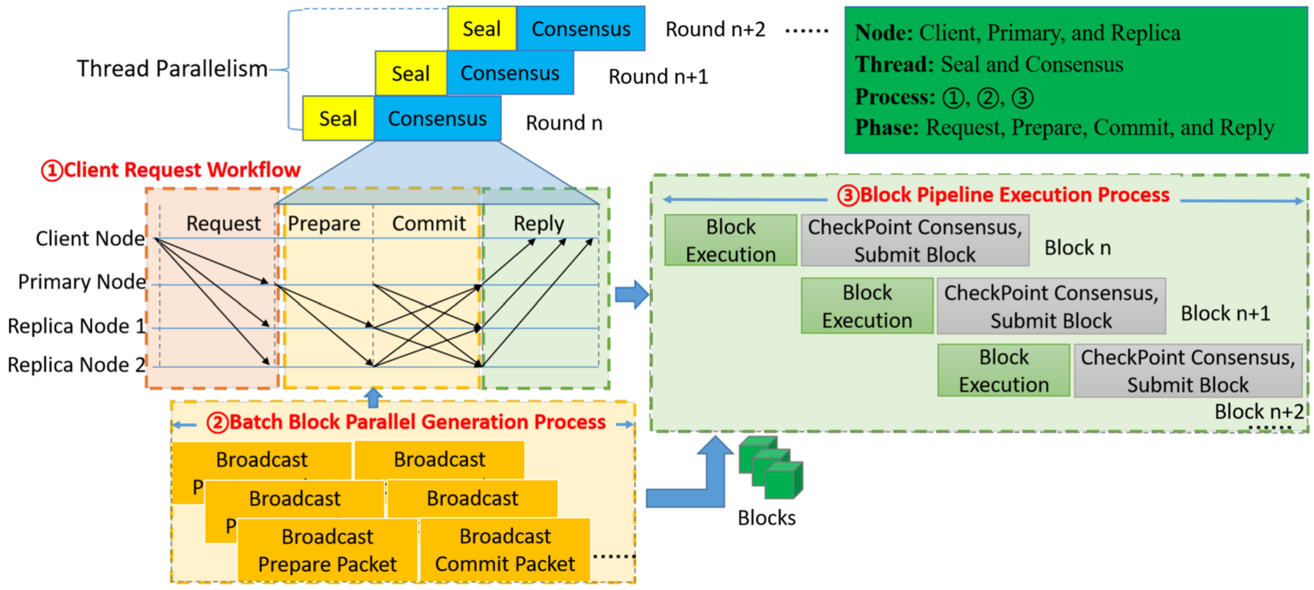


Fig. 2 TEP-BFT is structured into three distinct processes, facilitating multiple client interactions and transaction consensus simultaneously

tually writing the new blocks that reached consensus into the blockchain. Once a block is added to the blockchain, the transactions it contains are removed from the transaction pool. Since these two threads require non-conflicting resources, they operate independently and can process in parallel. Assuming the Seal thread requires time s and the Consensus thread c, the parallel processing can save time equal to $(n-1) * s$ after n rounds of consensus.

Processes Within the blockchain system, the Consensus is divided into three consensus processes: client request, batch block parallel generation, and block pipeline execution. These processes can execute in parallel, allowing multiple blocks to be consensused simultaneously. Both batch block generation and block pipeline execution support parallel consensus on multiple blocks, thereby enhancing the blockchain's throughput. In process ①, multiple clients can concurrently submit transaction consensus requests, referred to as Request packets. The primary node aggregates transactions from the transaction pool and organizes them sequentially. This operation is managed by the *Seal* thread, which is tasked with the systematic packaging of transactions. The *Consensus* thread encompasses processes ② and ③, which are executed in parallel. In process ②, block generation occurs concurrently through the simultaneous broadcasting of prepare and commit messages. This parallel processing enhances the efficiency of the consensus mechanism. Meanwhile, in process ③, the consensus engine consistently retrieves unexecuted blocks from the block queue, processes them, and performs pipeline validation of the results. This process ensures that blocks are executed in a correct and timely manner. Upon successful validation, the results are disseminated back to the clients via Reply messages. This structured approach allows the TEP-BFT protocol to efficiently handle multiple transactions and consensus requests concurrently, thereby optimizing throughput and reducing latency within the network.

Phases Each Consensus thread completes its task through four phases: request, prepare, commit, and reply. The primary node determines the order of client requests and forwards them to the replica nodes. Then, all nodes execute a two-phase (prepare/commit) protocol to reach an agreement on the order of requests. Subsequently, each node processes the requests and sends responses to the respective clients. A client accepts the result only after receiving at least $f + 1$ consistent replies. In this consensus protocol, identifiers are generated by the USIG service provided by an SGX-based trusted component, ensuring that an identifier can be assigned to only one message request, and that these identifiers are monotonic, unique, and ordered. Replica nodes need only verify the message's signature and the integrity of its content, without the need to compare the content of the same identifier's messages received by other nodes. Consequently, TEP-BFT simplifies the prepare phase of the traditional PBFT protocol. In terms of the number of replicas, faulty nodes can decide not to send messages or to send corrupted ones but cannot send two different messages with the same UI and correct certificate. Thus, TEP-BFT requires the participation of only $2f + 1$ nodes in the consensus workflow to withstand f Byzantine nodes.

4 TEP-BFT protocol

This section primarily explains the implementation principles and specific procedural steps of TEP-BFT. Firstly, we discuss the core functionalities of USIG implemented based on SGX. SGX provides a secure and isolated area for USIG operations, ensuring the confidentiality and integrity of USIG executions. The application of USIG ensures that each message sequence number corresponds to only one message, thereby maintaining the monotonicity, uniqueness, and order of messages. Additionally, the TEP-BFT mechanism based on USIG converts the tolerance of f Byzantine nodes into the tolerance of f faulty nodes, thus requiring only $2f+1$ nodes to participate in the consensus process to withstand attacks from f Byzantine nodes. Subsequently, we apply USIG based on Intel SGX to the TEP-BFT protocol, explaining both the normal operational flow of the TEP-BFT protocol and the view change process when nodes fail.

4.1 The SGX-USIG

In this paper, we propose the SGX-USIG, a *Unique Sequential Identifier Generator* (USIG) service based on Intel Software Guard Extensions (SGX). The USIG has traditionally been implemented as a service based on the Trusted Platform Module (TPM). This implementation leverages the capabilities of TPM, utilizing a private Attestation Identity Key pair (AIK) to sign data, thereby ensuring the security and integrity of the USIG service. Our SGX-USIG, service shares a similar structure with the previous TPM-based implementation but offers enhanced security and remote attestation capabilities. However, SGX provides a secure isolated area for USIG operations, ensuring the confidentiality and integrity of these operations, even in the presence of compromised servers. Here is an overview work mechanism of our proposed SGX-USIG service:

Initialization of SGX enclave The core component of the SGX-based USIG service is located within a secure enclave created by Intel SGX. This enclave is tamper-resistant and isolated from the rest of the system, ensuring the security of USIG operations. Firstly, Establish a secure enclave using

Intel SGX, which serves as the isolated execution environment for the USIG service. Then, deploy the USIG service software within the enclave. This software should include the necessary cryptographic libraries for digital signature generation.

The choice of monotonic counter The blockchain height h is used to increment the monotonic counter, recorded as $UI.h$, eliminating the need to create and maintain a separate counter. This approach saves storage and computational overhead. Using blockchain height as the monotonic counter offers benefits like immutability, decentralization, synchrony, temporality, transparency, and auditability. These properties make blockchain height a reliable and secure way to implement a monotonic counter, especially in environments requiring high verifiability and trustworthiness.

USIG service functions The interface of the service has two functions:

- $createUI(m)$: This function generates a USIG certificate containing a unique identifier (UI) and proves that the UI was created for a given message m within the SGX Enclave, signed by the attestation key private key during the remote attestation process. The unique identifier is a reading of the monotonic counter, which is incremented with each call to *createUI*.
- $verifyUI(PK, UI, m)$: The function is responsible for verifying the validity of the unique identifier UI for a given message m, utilizing Intel's remote attestation service to obtain the attestation key public key for certificate validation. It checks if the USIG certificate matches the message and other associated data in UI.

Remote authentication SGX's hardware-based remote authentication allows secure communication, ensuring requests from other servers are legitimate and unaltered. The communication's security is protected by mechanisms provided by SGX. Remote servers can establish a communication session with the USIG isolated area. During this session, SGX executes an authentication protocol to verify the requesting source server. Only servers that pass verification can communicate with the USIG isolated area.

Implementation of USIG-Sign The USIG-Sign certificate includes a digital signature obtained using the message and the private key within the isolated area. When *createUI(m)* is called, the message is signed using the private key with the ECDSA algorithm inside the USIG isolated area, generating a digital signature. The private key is protected by the enclave isolation provided by SGX. Compared to TPM, SGX offers a more rigorous and comprehensive security approach, particularly suitable for applications needing to protect sensitive data and private keys. It not only isolates sensitive operations but also provides mechanisms to resist various attacks, ensuring the confidentiality and integrity of data. This ensures a high level of security, even in compromised environments.

Replication and multi-instance support In scenarios where multiple applications require BFT replication on the same set of servers, multiple instances of SGX-based USIG services can be deployed on each machine. This flexibility allows for efficient utilization of hardware resources.

Transitioning from TPM-based USIG services to Intel SGX-based USIG services with remote attestation significantly improves security and isolation. The USIG isolated area in SGX ensures the confidentiality and integrity of the service, making it a reliable choice in modern server environments that require tamper-proof identifier generation services. Furthermore, multi-instance support makes it suitable for various replication scenarios.

4.2 Client request process

A *client* initiates an operation, denoted as *op*, by sending a message $\langle$Request, $c, seq, op, sig_c\rangle$ to all servers. Here, c represents the client ID, *seq* is the request identifier, and sig_c is the signature of the client on the sent message.

Then, each node stores the latest request *seq* sent by the client in a vector V_{seq}. Nodes discard requests where *seq* is less than the request identifier in the latest message, to prevent executing the same request twice and any requests received while processing the previous one. Requests are signed using the client's private key. Requests with an invalid signature sig_c are simply discarded. After sending a request, the client waits for $Reply$ messages $\text{Reply}_i = \langle \text{Reply}, S_j, seq, rst_{ji}\rangle$ with matching results rst_{ji} from $f+1$ different nodes, where i represents the block number, and S_j represents the primary and replica node ID. This ensures that at least one reply is from a benign server. If the client does not receive enough replies within the interval read by its local clock, it resends the request. If the request has already been processed, the nodes resend the cached reply.

4.3 Batch block parallel generation process

In the *Seal* thread, the primary node packages transactions from the transaction pool into multiple blocks. In the batch block parallel generation phase of the *Consensus* thread, nodes concurrently consensus the packaged blocks to generate sorted, unexecuted blocks. Let the current block height of the blockchain be h. The protocol for this stage is shown as Algorithm 1, and the specific operation process is as follows:

1. The primary node retrieves several packaged blocks from the transaction pool, denoted as $\text{Blocks} = [\text{Block}_1, \text{Block}_2, \cdots, \text{Block}_i, \cdots, \text{Block}_{\text{BlockLimit}}]$, and places these blocks

in the *Prepare* message packet, producing PreMessages = [Prepare$_1$, Prepare$_2$, $\cdots$, Prepare$_i$, $\cdots$, Prepare$_{\text{BlockLimit}}$]. Each *Prepare* message packet includes the message type, view information, primary node's id, the packaged block, and a certificate generated by the primary node for the *Prepare* message, that is Prepare$_i$ = $\langle$Prepare, v, S_p, $Block_i$, $UI_{p_i}\rangle$. UI_{p_i} is a unique identifier generated by the createUI function, so no two messages can have the same identifier. *BlockLimit* is a parameter that limits the number of blocks that can be concurrently consensed, ensuring the stability of the blockchain system. (See lines 5-9 of Algorithm 1)

2. The primary node broadcasts the multiple *Prepare* message packets simultaneously to all other replica nodes. After receiving a *Prepare* message packet Prepare$_i$, other replica nodes use *verifyUI* to check the correctness of UI_i. (See lines 10-15 of Algorithm 1) If the verification is successful, they continue to check the following:

   - Whether the *Prepare* message packet has already been received locally.
   - v is the current view number and the sender is indeed the primary node of the current view ($view\%n$).
   - The signature of the client in message m is correct;
   - The replica node has already accepted the request Block$_{i-1}$, where $UI_{p_{i-1}}.h = UI_{p_i}.h - 1$. That is, all corresponding requests smaller than UI$_i.h$ have been accepted and executed.
   - The validity of the message packet index i must be greater than the current blockchain height h and less than h + BlockLimit.

3. After a replica node successfully verifies a *Prepare* message packet, it adds the packet to its local cache and broadcasts Commit$_i$ = $\langle$Commit, v, Sr_j, S_p, $Block_i$, UI_{p_i}, $UI_{r_{ji}}\rangle$ to all other nodes. Here, *Commit* is the message type, v is the current view, Sr_j is the ID of the replica node, $Block_i$ is the packaged block, and $UI_{r_{ji}}$ is the certificate generated by the replica node for the *Commit* message. Both *Prepare* and *Commit* messages have unique identifiers UI generated by the *createUI* function, ensuring that no two messages share the same identifier. Servers use the *verifyUI* function to check the validity of identifiers received in messages. (See lines 16-23 of Algorithm 1)
4. When other nodes receive a *Commit* message packet $Commit_i$, they verify its validity. In addition to the five steps of *Prepare* message verification, this includes checking whether the node has received $f + 1$ valid *Commit* messages for Block$_i$. If a replica node does not receive a *Prepare* message from the primary node but receives a valid *Commit* message, it also broadcasts the corresponding *Commit* message. This is because the *Commit* message includes the primary node's certificate UI_{p_i}, proving that the corresponding *Prepare* message is problem-free. (See lines 26-30 of Algorithm 1)
5. Once the *Commit* message packet $Commit_i$ is verified, the node adds it to the local cache. When the node collects $f+1$ *Commit* message packets, it retrieves the block Block$_i$ from the preprocessed message packets and commits it to storage. (See lines 31-34 of Algorithm 1)

Algorithm 1 Batch parallel generation of blocks.

```
1: Input: BlockLimit $BL$, Blocks $bs$
2: Output: $BLQueue\ Q$
3: upon obtainment of $bs$ at $S_p$ do
4:   // The primary node retrieves blocks from the transaction pool
5:   while $i \leq BL$
6:     // Create a unique identifier for each block using CreateUI function
7:     $UI_{p_i}$ = CreateUI($bs[i]$)
8:     // Generate the Prepare message for each block
9:     $PrepareMessage$ = CreatePrepareMessage($v$, $S_p$, $bs[i]$, $UI_{p_i}$)
10:    // Broadcast the Prepare message to all replica nodes
11:    BroadcastPrepareMessage($PrepareMessage$)
12:  end while
13: upon reception of $< PrepareMessage >$ at $S_r$ do
14:  // Verify that the Prepare message has not been received before
15:  if NotReceivedPrepareMessage($v$, $S_p$, $UI_{p_i}$)
16:    // Validate the Prepare message's contents and signatures
17:    if VerifyPrepareMessage($v$, $S_p$, $bs[i]$, $UI_{p_i}$)
18:      // Add the valid Prepare message to the local cache
19:      AddToLocalCache($PrepareMessage$)
20:      // Create a Commit message upon successful validation
21:      $CommitMessage$ = CreateCommitMessage($v$, $S_r$, $S_p$, $bs[i]$, $UI_{p_i}$, $UI_{r_{ji}}$)
22:      // Broadcast the Commit message to all nodes
23:      BroadcastCommitMessage($CommitMessage$)
24:    end if
25:  end if
26: upon reception of $f + 1 < CommitMessage >$ at $S_p$ and $S_r$ do
27:  // Verify that the Commit message has not been received before
28:  if NotReceivedCommitMessage($v$, $S_r$, $S_p$, $UI_{p_i}$, $UI_{r_{ji}}$)
29:    // Validate the Commit message's contents and signatures
30:    if VerifyCommitMessage($v$, $S_r$, $S_p$, $bs[i]$, $UI_{p_i}$, $UI_{r_{ji}}$)
31:      // Add the valid Commit message to the local cache
32:      AddToLocalCache($CommitMessage$)
33:      // Submit the block to storage in the $Q$
34:      CommitBlockToStorage($bs[i]$)
35:    end if
36:  end if
```

For all the preprocessed message packets generated in this stage, the procedure is repeated from steps (2) to (5) to complete the parallel ordering consensus of $BlockLimit$ blocks. To ensure the correct sequence of blocks during the consensus process, each node maintains a vector $\mathbf{V}_{acp}$, where each element records the value of the last message's counter processed by each replica node (including *Prepare*, *Commit*, *Checkpoint*, *Viewchange*). For example, $\mathbf{V}_{acp} = (UI_{p1}.h, UI_{p2}.h, UI_{p3}.h, \ldots, UI_{p_n}.h)$, where $UI_{p_n}.h$ is the counter value of the last message sent by the primary node received by the current replica node.

4.4 Block pipeline execution process

During the block batch parallel generation process, consensus engine generates $BlockLimit$ deterministic blocks which are placed into the block queue, denoted as $BLQueue = [\text{Block}_i, \text{Block}_{i+1}, \ldots, \text{Block}_{i+\text{BlockLimit}}]$. In the process, the consensus engine continuously extracts unexecuted blocks from the block queue for execution, and conducts a pipeline consensus on the execution results of these blocks. The protocol for this process is presented as Algorithm 2, with specific steps as follows:

1. The consensus engine retrieves an unexecuted block, referred to as $Block_i$, from the block queue and inputs it into the execution engine. The state resulting from executing the block is noted as $Checkpoint_i$, with its corresponding hash denoted as $cPHash_i$.(See lines 5-8 of Algorithm 2)
2. After block execution, nodes generate a $Checkpoint$ message packet, denoted as $\text{CheckPointMessage}_i = \langle \text{Checkpoint}, S_j, UI_{latest}, cPHash_i, UI_{cj} \rangle$, where UI_{latest} is the signature of the most recently executed request, $cPHash_i$ is the current node state's hash value, and UI_{cj} is the signature obtained by calling *createUI* on this *Checkpoint* message. This $Checkpoint$ message packet is then broadcast to all nodes. (See lines 9-14 of Algorithm 2)
3. Other nodes receiving $CheckPointMessage_i$ verify the validity of the signature. If the signature passes validation, the message packet is placed into local cache. (See lines 17-22 of Algorithm 2)
4. When a node collects $f+1$ $Checkpoint$ message packets with execution results matching their own and from distinct consensus nodes, it is considered that all consensus nodes have reached an agreement on the block execution result $Checkpoint_i$. The execution result $CheckPoint_i$ is then committed to storage, and the blockchain state is updated to the latest. At this point, nodes employ the Garbage Collection (GC) mechanism based on the checkpoint to discard all log entries with sequence numbers less than UI_{latest}. When a view change occurs, a new checkpoint is generated, and the log list is cleared. (See lines 23-26 of Algorithm 2)
5. Once $Block_i$ has finished executing, $Block_{i+1}$ can execute based on the state of $Block_i$, using the hash of $Block_i$ as its parent block hash. This produces a new execution result $Checkpoint_{i+1}$, and the steps above are repeated to reach consensus on the execution results of the remaining $BlockLimit - 1$ blocks in the $BLQueue$. (See lines 31-33 of Algorithm 2)

Algorithm 2 Block pipeline execution.

```
1: Input: BlockLimit $BL$, $BLQueue$ $Q$, $UI_{latest}$ $UI_l$
2: Output: $checkPoint$ set $cP$
3: upon obtainment of $BL < Q >$ at $S_j$ do
4:   for $i = 1$ to $BL$
5:     // Retrieve the next unexecuted block
6:     $Checkpoint_i$ = ExecuteBlock($Q[i]$)
7:     // Calculate the hash for the execution result
8:     $cPHash_i$ = CalculateHash($Checkpoint_i$)
9:     // Create a unique identifier for the block using the latest signature
10:    $UI_{cj}$ = createUI($Q[i]$)
11:    // Generate a Checkpoint message with necessary signatures
12:    $CheckpointMessage_i$ = CreateCheckpointMessage ($S_j, UI_l, cPHash_i, UI_{cj}$)
13:    // Broadcast the Checkpoint message to all nodes
14:    BroadcastCheckpointMessage($CheckpointMessage_i$)
15:  end for
16:
17: upon reception of $f + 1 < CheckpointMessage >$ at $S_j$ do
18:   if NotReceivedCheckpointMessage($S_j, UI_l, UI_{cj}$)
19:     // Verify the signature of the received message
20:     if VerifyCheckpointMessage($S_j, UI_l, cPHash_i, UI_{cj}$)
21:       // Store the valid message in the local cache
22:       AddToLocalCache($CheckpointMessage_i$)
23:       // Commit the execution result once agreement is reached
24:       $cP$ := CommitExecutionResult($Checkpoint_i$)
25:       // Update blockchain state and trigger garbage collection
26:       PerformGarbageCollection($UI_{latest}$)
27:     end if
28:   end if
29: end upon
30:
31: // Once Block $i$ finishes execution, proceed to the next block
32:   ExecuteNextBlock($Block_{i+1}$)
33:   Repeat the above steps for the remaining $BlockLimit - 1$ blocks in $BLQueue$.
```

4.5 View change operation

In the normal operation, the primary node assigns sequence numbers to the requests it receives and broadcasts these numbers to other replica nodes using *Prepare* messages. Our protocol significantly limits the malicious actions that the primary node can perform: it cannot duplicate or arbitrarily assign higher sequence numbers. However, a malicious primary node can still prevent consensus operations by either not assigning sequence numbers to some requests or not assigning sequence numbers to any requests.

A view change must be executed, and a new primary node chosen when the primary node fails or acts maliciously. View changes are triggered by timeouts. When a replica node receives a request from a client, it starts a timer T_{sp} that times out after a fixed period. The timer stops when the request is accepted. If the timer expires, the replica node suspects the primary node of failure and initiates a view change.

The view change operation protocol is shown as Algorithm 3. When the timer of a replica node Sr_j times out, Sr_j sends a message $\langle \text{ViewchangeReq}, Sr_j, v, v' \rangle$ to all nodes, where v

Algorithm 3 View change process.

```
1: Input: currentView $v$, replica node $Sr_i$, new-view certificate $V_{nv}$, set of requests to be executed $NV_c$, Checkpoint certificate $cP_{latest}$
2: Output: $newView\ v'$
3: upon reception of $f+1$ $\langle ViewchangeReq, Sr_j, v, v'\rangle$ messages at the replica node $Sr_j$ do
4:   $currentView := v'$
5:   multicast $\langle Viewchange, Sr_j, v', cP_{latest}, M, UI_v\rangle$
6:
7: upon reception of $\langle Viewchange, Sr_j, v', cP_{latest}, M, UI_{vj}\rangle$ at other replica nodes $Sr_j$ do
8: // $Sr_j$ check if message is consistent with system state
9:   if $cP_{latest}$ contains $f+1$ Checkpoint messages set
10:     if $UI_{vj}.h = UI_{vj-1}.h + 1$
11:       if $M$ is not null
12:         The counter values of the messages in $M$ should be continuous
13:       end if
14:     end if
15:   end if
16:
17: upon reception of $f+1$ $\langle Viewchange, Sr_j, v', cP_{latest}, M, UI_{vj}\rangle$ messages at the new primary node $S'_p$ do
18:   $S'_p$ store $Viewchange$ messages in $V_{nv}$
19:   addToNVc($S'_p$, $v'$, $cP_{latest}$, $M$, $UI_{vj}$)
20:   computeNVc(prepared messages $P$, accepted messages $A$)
21:   multicast $\langle newView, S'_p, v', V_{nv}, NV_c, UI_n\rangle$
22:
23: upon reception of $\langle newView, S'_p, v', V_{nv}, NV_c, UI_n\rangle$ message at other replica nodes $Sr_j$ do
24:   $Sr_j$ check if $V_{nv}$ contains all required requests
25:   computeNVc(prepared messages $P$, accepted messages $A$)
26:   if isSComputedProperly() // Verify if $NV_c$ was computed properly
27:     executeRequestsNVc() // Begin the new view
28:   else
29:     requestMissingCommitCertificates() //Request missing commit certificates to update state
30:   end if
```

is the current view number and $v' = v + 1$ is the new view number. When Sr_j receives the other $f+1$ $ViewchangeReq$ messages, it transitions to view v' and broadcasts a message ⟨Viewchange, Sr_j, v', cP_{latest}, M, UI_{vj}⟩, where cP_{latest} is the latest checkpoint certificate (i.e., the collection of those $f+1$ valid $Checkpoint$ messages), and M is the set of all messages sent by the node since the latest checkpoint was generated, including: $Prepare$, $Commit$, $Viewchange$, and $newView$ messages. At this point, the node stops accepting messages in view v.

The $Viewchange$ messages utilize unique identifiers UI_{vj} obtained by calling $createUI$. The goal is to prevent Byzantine nodes from sending $Viewchange$ messages with different cP_{latest} and M to different nodes, leading to different decisions on the last request of the previous view. If Byzantine nodes do this, benign nodes can still detect them by validating UI_{vj}. Only the normal nodes that are consistent with the system state will consider the ⟨Viewchange, Sr_j, v', cP_{latest}, M, UI_{vj}⟩ message. Normal nodes perform the following checks on $ViewChange$:

1. cP_{latest} actually has $f+1$ valid UI identifiers.
2. In UI_{vj}, $UI_{vj}.h = UI_{vj-1}.h + 1$.
   - If M is not empty, then $UI_{vj-1}.h$ is the highest counter value in M.
   - If M is empty, then $UI_{vj-1}.h$ is the counter value in cP_{latest}.
3. The counter values in messages M are consecutive.

When the new primary node S'_p of view v' receives *Viewchange* messages from $f+1$ distinct replica nodes, it stores them in a collection called V_{nv}, which is the new view certificate. V_{nv} will include all requests made after the previous checkpoint, including those that are only prepared but not yet accepted. To define the initial state for the new view v', the new primary node uses the information from the cP_{latest} and M fields in the *Viewchange* messages to define NV_c, which is a collection of requests that have been prepared/accepted since the checkpoint. To compute NV_c, the primary node first selects the most recent and valid checkpoint certificate received in the *Viewchange* messages. Next, it chooses requests from the M collection that have counter values greater than those in the latest checkpoint certificate.

After making this calculation, the primary node broadcasts a message ⟨newView, S'_p, v', V_{nv}, NV_c, UI_n⟩. When a replica node receives a *newView* message, it verifies the validity of the new view certificate V_{nv}. All replica nodes also perform the same calculation as the primary node to verify that NV_c is correctly computed. Replica nodes then start the set of all requests in the new view v' that were accepted in view v, denoted as S_{acc}. If a replica node detects that the counter values between its latest executed request and the first request in NV_c are not consecutive, it initiates a Commit check with all other nodes to retrieve missing requests. If these message requests have been deleted by other nodes due to the garbage collection mechanism, they use the same State Transfer mechanism as PBFT to directly transition the state (without needing to execute requests).

In previous BFT protocols, even when a view change occurs, the sequence numbers for requests would continue to be allocated for execution in the order of the previous view number. However, in TEP-BFT, this is not the case, as each view's sequence numbers are provided by the different node's SGX-based USIG. Therefore, when a view change occurs, the first sequence number for the new view must be defined. The new view v' starts with the counter value of UI_n in the $newView$ message plus one. When a node sends a *viewChange* message, it starts a timer T_{nc}, which if expires before receiving a valid *newView* message, necessitates an additional view change. Each time, the timer is multiplied by two, increasing exponentially until the new primary node responds. The goal is to avoid perpetual timeouts due to prolonged communication delays.

5 Proof of correctness

Our protocol upholds the safety and liveness attributes inherent to conventional Byzantine Fault Tolerance (BFT) protocols. Within this context, safety ensures that all benign nodes process identical requests in a consistent sequence, whereas liveness guarantees that requests from benign clients are invariably executed. The demonstration of the correctness of TEP-BFT is elaborated upon in this section.

5.1 Safety

Theorem 1 *In the same view v, if a benign node executes an operation op with the identifier $UI.h$, no other benign node will execute this operation with a different identifier $UI.h'$ where $UI.h' \neq UI.h$.*

Proof $UI.h$ is an identifier assigned by the primary node using the *createUI* function. If a benign node executed an operation op with identifier $UI.h$, it must have accepted $f+1$ valid *Commit* messages for $\langle op, UI.h\rangle$. Let these $f+1$ nodes be denoted as N_s.

By proof of contradiction, assume there is another node S' that executes operation op with identifier $UI.h'$ where $UI.h' > UI.h$. According to TEP-BFT, S' would have accepted $f+1$ valid *Commit* messages for $\langle op, UI.h'\rangle$. Let these $f+1$ nodes be denoted as N_s'. Since $n = 2f+1$ and $|N_s| + |N_s'| = 2f+2 > n$, there must be at least one node S_l (the intersecting node), that sent *Commit* messages for both $\langle op, UI.h\rangle$ and $\langle op, UI.h'\rangle$. Therefore, it can be concluded that S_l is a Byzantine node. S_l could be the current primary node S_p or a replica node Sr_i. We need to consider the following cases:

1) ***When S_l is the primary node S_p***: S_p generates two UIs for the same operation op, i.e., $UI = \langle UI.h, H(op)\rangle$ and $UI' = \langle UI.h', H(op)\rangle$. At the same time, it is possible that Byzantine replica nodes also accepted UI and UI', sending *Commit* messages with $\langle UI, UI_c\rangle$ and $\langle UI', UI_c'\rangle$ to some benign node. Now, the execution of the benign node S_c can be divided into the following two cases:

- $\underline{S_c \text{ has executed } UI.h.}$ At this time, $V_{seq}[Client] = op.seq$. Then S_c will not accept the *Commit* message with $\langle UI', UI_c'\rangle$, because it contains the same request as $op.seq$. S_c will only accept those *Commit* messages where $op'.seq > V_{seq}[Client]$.
- $\underline{S_c \text{ has not executed } UI.h \text{ yet.}}$ S_c must execute all requests before $UI.h'$ (gaps between identifiers are not allowed). Hence, $UI.h$ must be executed before $UI.h'$.

2) ***When the primary node S_p is benign and S_l is a replica node***: S_p will not generate two different UIs for the same operation op. If S_l sends two different *Commit* messages, i.e., $\langle op, UI.h\rangle$ and $\langle op, UI.h'\rangle$, the other benign nodes, upon validating UI and UI' through *VerifyUI*, will only accept the *Commit* message with the identifier $UI.h$.

To conclude, it is known that a benign node will not execute the same request operation with different identifiers.

Theorem 2 *In view v, if a benign node executes an operation op with identifier $UI.h$, then in any view $v' > v$, no other benign nodes will execute this operation with a different sequence number $UI.h'$ where $UI.h' \neq UI.h$.*

Proof Although v' can be any value, any view v'' between v and v', can be considered as multiple iterations of $v' = v+1$, regardless of whether any requests were executed in v''. Hence, we only need to discuss the case where $v' = v+1$.

Lemma 2 is also proved by contradiction. If a benign node S_c in view v executes an operation with identifier $UI.h$, it must receive $f+1$ valid *Commit* messages for $\langle op, UI.h, v\rangle$, denoted as N_s. Suppose there is another benign node S_c', which in view v' executes the operation op with identifier $UI.h'$ where $UI.h' > UI.h$. According to the TEP-BFT protocol, S_c' receives $f+1$ valid *Commit* messages for $\langle op, UI.h', v'\rangle$, denoted as N_s'. Because $n = 2f+1$ and $|N_s| + |N_s'| = 2f+2 > n$, there must be at least one intersecting Byzantine node S_l that sent two different *Commit* messages, namely $\langle op, UI.h, v\rangle$ and $\langle op, UI.h', v'\rangle$.

Firstly, we must prove that the new primary node S_p of view v' must acknowledge that op was accepted or executed in view v. This is proved through the new view certificate V_{nv}, which contains $f+1$ *Viewchange* messages $\langle \text{Viewchange}, S_i, v', cP_{latest}, M, UI_i\rangle$, from $f+1$ nodes, denoted as N_s'', with at least one benign node S_c among them. The *Viewchange* message from S_c is $\langle \text{Viewchange}, Sc, v', cP_{latest}, M, UI_c\rangle$. Also, Byzantine node S_l would have sent a *Viewchange* message before sending $\langle op, UI.h', v'\rangle$. Now we need to consider the following two cases:

1) ***If op was executed after the most recent stable checkpoint***: op's *Commit* message will be included in M of Viewchange. However, Byzantine S_l may not include op's *Commit* message in M. In this case, we further consider the following scenarios:

- $\underline{\text{If } S_p \text{ is benign:}}$ If $S_c \in N_s''$ executed operation op in $\underline{\text{view } v}$, then S_c's *Viewchange* message's M contains the *Commit* message for operation op. Hence, through this

Viewchange message, S_p can determine that *op* has been executed. If S_c did not execute *op* in view v, S_l would have to perform one of two actions detectable by S_p to exclude *op* from M: 1) If S_l executed a request op' after *op*, S_l could include op''s *Commit* message in M but not *op*'s, leaving a gap in M detectable by S_p; 2) If S_l sent a *Commit* message with $\langle UI.h, op\rangle$, it might not include any *Commit* messages with $UI.h' > UI.h$ in M, but S_p will detect this because S_l must sign the *Viewchange* message with $UI.h'$. Therefore, for S_l's *Viewchange* message to be inserted into V_{nv} by S_p, S_l must include *op*'s *Commit* message in M.

- If S_p is Byzantine: S_p may attempt to modify the M it inserts into V_{nv}. If it merely removes *op* from M, it leaves a gap that is detectable. If it removes *op* and all subsequent messages, this is also detectable because S_p cannot forge a UI from S_c with a counter value higher than the later messages. If S_p inserts S_l's *Viewchange* message into V_{nv}, however, benign nodes will verify the validity of V_{nv} upon receiving the *newView* message from S_p. Therefore, Byzantine S_p cannot forcibly add S_l's *Viewchange* message to V_{nv}.

2) ***If op was executed before the most recent stable checkpoint***: The execution of *op* is implicit in the certificate of the most recent stable checkpoint. Byzantine S_l may attempt to place an older checkpoint in the *Viewchange* message. In this case, we further consider the following scenarios:

- If S_p is benign: If $S_c \in N_s''$ executed operation *op* in view v, since S_c is benign, S_c's Viewchange message's cP_{latest} already includes the fact that *op* was executed. Hence, S_p can determine that *op* has been executed. At the same time, S_p will not insert the *Viewchange* message sent by S_l into the new view certificate V_{nv}, because by comparing the hash value $cPHash$ and cP_{latest} of the current node state in the *Viewchange* message of S_c and S_l, it will be found that S_l does not include the execution of *op* in the checkpoint. If S_c did not execute *op* in view v, S_p will never insert S_l's Viewchange message into V_{nv}, because S_l must perform one of the detectable actions indicated in the condition 1)-1.
- If S_p is Byzantine: If S_p attempts to replace cP_{latest} with an older checkpoint certificate, it also cannot forge the UI. Even if it uses an older checkpoint sent by S_c, this is also detectable. This is because when a benign node receives a *newView* message, it checks the validity of V_{nv}. Therefore, Byzantine S_p cannot tamper with the contents of benign nodes' *Viewchange* messages. If S_p inserts S_l's *Viewchange* message into V_{nv}, however, benign nodes will verify the validity of V_{nv} upon receiving the *newView* message from S_p. Therefore, Byzantine S_p cannot forcibly add S_l's *Viewchange* message to V_{nv}.

Through the above deductions, it is shown that the new primary node S_p of view *v'* must acknowledge that *op* was accepted or executed before *v'*. The following will continue to prove that benign nodes will not execute *op* in view *v'*, where the sequence number $UI.h'$ in view *v'* is different from $UI.h$. Consider the following two cases:

- If S_p is benign: As proved above, S_p can confirm that *op* has been executed, hence a benign S_p will not generate another UI for the same *op* in view *v'* and send a *Commit* message. Therefore, other benign nodes will also not execute *op* in view *v'* that has already been done in *v*.
- If S_p is Byzantine: S_p can create a new *Prepare* message containing $UI' = \langle UI.h', H(op)\rangle$ and send it to other nodes, where benign replica nodes will validate $op.seq$ and find $op.seq \le V_{seq}[Client]$, meaning the request has already been executed, thus benign nodes will not execute it again.

5.2 Liveness

Theorem 3 *During a stable view, operations requested by benign clients will be completed.*

Proof We define a stable view as one where the primary node is benign and there are no timeouts on benign replica nodes. If the $client$ is benign, it will send operation *op* with a sequence number seq greater than any previously used to all nodes. Since the primary node S_p is benign in a stable view, it will generate a $UI = \langle UI.h, H(op)\rangle$ and send a *Prepare* message with UI to all other replica nodes. Benign replica nodes, after receiving this message, will call *verifyUI* to validate the UI and send a *Commit* message for $\langle UI.h, op\rangle$. Since there are at most f Byzantine nodes in the system, at least $f + 1$ benign nodes (including S_p and other f replica nodes) will generate these *Commit* messages and send them to all others. When a benign node receives $f + 1$ *Commit* messages, it will execute *op* and send a *Reply* message to the client Client. Once Client receives $f + 1$ matching *Reply* messages, the operation will be considered complete. Because there are $f + 1$ benign nodes, the above will inevitably happen, and they will work the same as the $UI.h$-th operation when executing *op*.

Theorem 4 *If at least $f + 1$ benign nodes request a view change, then view v will eventually be changed to a new view $v' > v$.*

Proof To request a view change, a benign node S_c sends a $\langle$ViewchangeReq, S_c, v, $v'\rangle$ message to all nodes, where v is the current view number and $v' = v + 1$ is the new. Consider a view change from v to $v + 1$ requested by a set of $f + 1$ benign nodes N_s. By definition, the primary node S_p in view v faces two conditions:

1. ***The view is stable***: This means that all nodes in N_s have received *ViewchangeReq* messages from each other. When one node S receives the $f+1$-th *ViewchangeReq* message, it sends a $\langle$Viewchange, $S, v', cP_{latest}, M, UI_{vs}\rangle$ message to all others. All *Viewchange* messages sent by nodes in N_s are received by all other nodes. The primary node S'_p of view v' is benign, so it sends a $\langle$newView, S'_p, v', V_{nv}, NV_c, $UI_n\rangle$ message to all others. Since the view is stable, all nodes receive the *newView* message, and the view changes to v'.
2. ***The view is unstable***: The following two cases should be taken into consideration:

- *S_p is Byzantine, but it does not send a newView message or sends an invalid newView message discarded by all benign nodes; or S_p is benign, but communication delays cause all benign nodes' timeouts to expire*: When nodes send *Viewchange* messages, they start a timer that expires after a fixed time unit T_{nc}. In this case, all benign nodes' timers will expire, and they will initiate another view change.
- *S_p is Byzantine, but sends the newView message to at least $f+1$ nodes (denoted as N'_s) among which less than $f+1$ are benign nodes; or S_p is benign, but communication delays lead to the same effect*: In this case, the Byzantine nodes in N'_s can act according to the protocol, making the benign nodes in N'_s believe it is running correctly. The nodes in N'_s can send *Prepare* and *Commit* messages following the normal operation process. For the benign nodes not in N'_s, their timers will expire after a fixed time unit T_{nc}, and these benign nodes will send *ViewchangeReq* messages, but there will not be $f+1$ such messages, so no view change will occur. When Byzantine nodes begin deviating from the normal operation process, the requests will stop being accepted, and the benign nodes in N'_s will send *ViewchangeReq* messages, initiating the view change. In both cases, when another view change begins, the system may fall back into either of the conditions 1) or 2). However, eventually, the view will become stable, and the system will fall into condition 1), and the view will change to the new view v'.

Theorem 5 *Operations requested by benign clients will eventually be completed.*

Proof This proof is derived from the previous lemmas. In a stable view, operations requested by benign clients will eventually be completed (Lemma 3). If view v is unstable, at the expiration of the timer, there exist two conditions:

1. ***At least $f+1$ benign nodes request a view change***: In this case, the view will change to a new v' (Lemma 4).
2. ***Less than $f+1$ benign nodes request a view change***: This scenario is similar to the situation in Lemma 4-2)-2. If there is at least a subset of $f+1$ nodes N'_s that do not request a view change and continue to operate in view v, the system will remain in view v, and requests from benign clients will be executed. If there is no such N'_s or requests are not executed within a fixed time, all benign nodes will request a view change, leading to condition 1).

Condition 1) will lead to a view change, but the new view v' may not be stable. The system model assumes that processing and communication delays will not grow indefinitely, and in the protocol, the fixed time T_{nc} doubles each time a new view change is needed. Therefore, even if view changes occur consecutively, eventually there will be a view v'', and one of the following two scenarios will happen:

- $\underline{S_p \text{ is benign}}$: There are no timeouts expiring on benign replica nodes because T_{nc} is greater than the observed maximum delay. In this case, the view is stable, and operations are executed through Lemma 3.
- $\underline{S_p \text{ is Byzantine}}$: In this case, S_p can deviate from the normal operation process leading to timeouts and new view changes, or follow the normal operation process to avoid view changes. In any case, the view is not stable, so we enter the above condition 1) or 2). Eventually, there will be a view where S_p is benign, because only a minority of nodes are Byzantine, and the view will eventually become stable.

6 Performance evaluation

6.1 Experiments setup and metrics

In this section, we developed an SGX-protected framework using Gramine and incorporated the proposed TEP-BFT protocol into the FISCO BCOS blockchain [20]. We used a web framework built in Golang as the interface to initiate the FISCO processes, aiding in the setup of the experimental environment. For our experiments, each participating machine hosted a single TEP-BFT node. Specifically, we assume that the processor, which initiates the TEE platform, is trustworthy and cannot be tampered with as long as the CPU microcode is up-to-date. We also assume that the blockchain provides secure and tamper-proof storage for all transaction data. Therefore, we do not consider security vulnerabilities and attacks specific to the TEE or the blockchain.

We established the experiments on physical industrial servers to emulate real-world conditions, with each TEP-BFT replica operating on an independent machine. The TEP-BFT framework was implemented on a private network. Our experiments were conducted on CPUs with 4, 8, 16, and 32 cores. The configuration for the 4-core CPU is an Intel® Celeron® Processor J4125 with 8GB of RAM (4GB allocated as trusted RAM), and Intel SGX enabled. For the 8, 16, and 32-core CPUs, the configuration includes an 8/16/32-core Intel Xeon (Ice Lake) Platinum 8369B processor with 16/32GB RAM (8/16GB as trusted RAM) and Intel SGX. The client, operating on an 8-core Intel Xeon (Ice Lake) Platinum 8369B cloud server with 16GB RAM, simulated multiple threads sending transaction requests to the TEP-BFT nodes.

To rigorously evaluate the system's performance, we designed multiple experimental scenarios with varying key performance indicators to assess the following metrics:

- *Latency* (Average Response Time, ART): This metric measures the duration from when a client issues a request to when it receives a response.
- *Throughput*: This represents the number of requests processed per second by the protocol. We determine the peak throughput under varying numbers of malicious nodes f, and examine the associated latency for each Byzantine Fault Tolerance (BFT) protocol, highlighting the relationship between throughput and latency.
- *Load*: This metric captures the count of active and queued processes in the system. Typically, the load should not surpass the number of CPU cores available; exceeding this threshold suggests that performance degradation is imminent, with processes beginning to queue.
- *Memory usage*. In a trusted execution environment, memory is divided into secure and non-secure memory areas. Due to the challenges in directly measuring secure memory, we infer variations in secure memory by observing changes in the non-secure, facilitated by Gramine's architecture.

These indicators guide our experimental design, enabling a detailed exploration of the interactions between transaction request rate (i.e., queries per second, QPS), throughput, and latency, as detailed in Table 2. **Test Case 1** and **Test Case 2** primarily investigate the impact of different QPS on blockchain performance and system metrics. Hence, with a fixed transaction count of 200,000, we control the number of consensus nodes in the blockchain system to be 5. Tests are conducted with various QPS, specifically 500, 1000, 1500, 2000, 2500, 5000, 7500, and 10,000. **Test Case 3** and **Test Case 4** mainly examine the influence of the number of normal nodes and Byzantine nodes on the performance of the entire blockchain system. On one hand, they verify the scalability of the consensus mechanism, and on the other, they confirm the security and liveness of the consensus mechanism. Meanwhile, the selected QPS for the experiments is the optimal request rate that the system can withstand. **Test Case 5** is designed to assess the performance impact of introducing a trusted execution environment while also comparing the performance advantages of our optimized protocol under the same conditions. To examine the effect of CPU performance when using a trusted execution environment, as well as the usability and scalability of the system, control experiments for all cases are conducted on physical industrial computers. **Test Case 6** compares the performance of our protocol with other existing improved BFT protocols under the same experimental conditions. This comparison aims to evaluate the performance advantages of our protocol relative to others. It should be noted that Zyzzyva [18] was not included in the experiment due to its reliance on the PBFT view change process when the primary node makes an error, as discussed in Section 2.2. This limitation diminishes its effectiveness in scenarios involving Byzantine node attacks, making it less suitable for our experimental evaluation.

6.2 How QPS impacts on latency and throughput

This use case investigates the impact of transaction request rate (i.e., queries per second, QPS) on throughput (i.e., transactions per second, TPS) and latency (average response time, ART) across industrial servers equipped with 4/8/16/32-cores CPU. We firstly tested the latency and throughput of proposed TEP-BFT method on a 4-core CPU under various QPS. Furthermore, the consensus protocol's latency and throughput on CPUs with 4/8/16/32-cores are also evaluated

Table 2 Parameter configuration for each use case

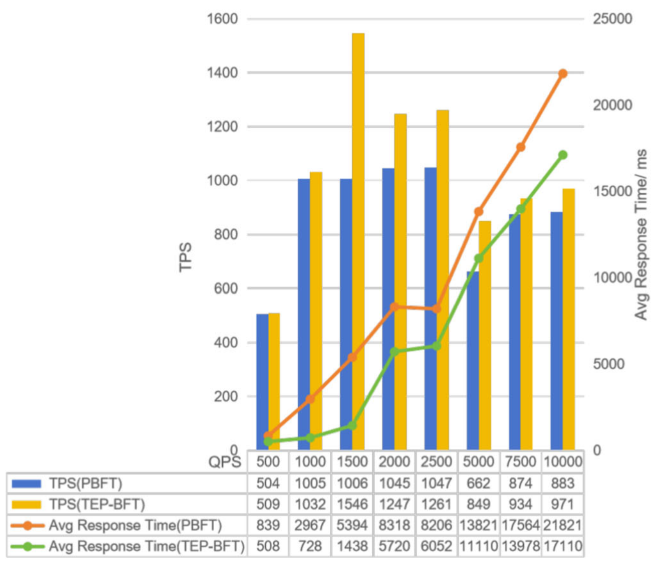
Use Case	QPS	Number of Transactions	Number of Nodes
How QPS impacts on Latency and Throughput	500, 1000, 1500...	20w	5
How the number of nodes impacts on the Latency and Throughput	optimal QPS	20w	3,5,7,9
How the number of Byzantine nodes impacts on the Latency and Throughput	optimal QPS	20w	f=0,1,2,3,4
Comparison of consensus performance w/wo Trusted Execution Environments	500, 1000, 1500...	20w	5
Performance comparison with state-of-the-art BFTs	500, 1000, 1500...	20w	5

and campared, employing 10,000 QPS. This assessment was conducted to ascertain the influence of CPU performance on protocol efficacy.

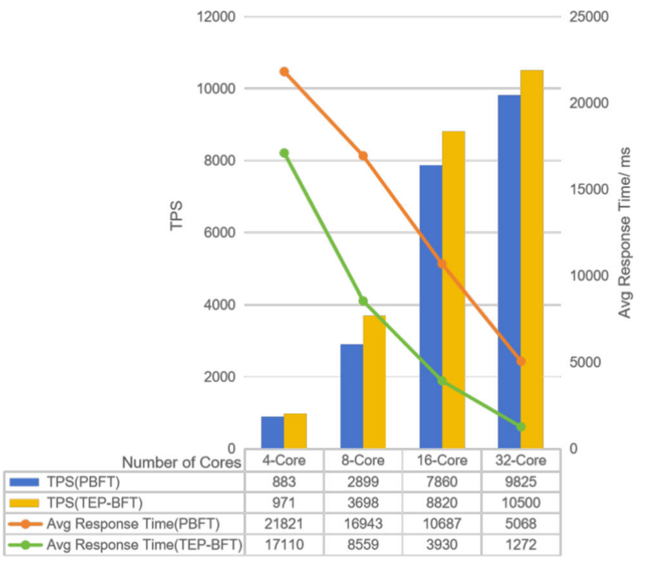
Performance evaluation under various QPS As delineated in Fig. 3-(a), the initial phase of our investigation was conducted within an industrial computing setting, equipped with a 4-core CPU. All experimental nodes were configured within the Software Guard Extensions (SGX) secure environment, thus providing a controlled and secure testing framework. The study revealed a direct correlation between the transaction request rate (QPS) and throughput (TPS), with the latter peaking at a QPS of 1500. Prior to the optimization of the Practical Byzantine Fault Tolerance (PBFT) protocol, the system registered a TPS of approximately 1000 within the SGX container. Subsequent to the protocol optimization, however, the throughput significantly increased to about 1500 TPS, representing a noteworthy 30% enhancement in performance at the maximum QPS. Moreover, the latency measurements indicated a general increase as the QPS escalated. Crucially, the optimized protocol exhibited superior performance by demonstrating lower latency compared to its pre-optimized counterpart under equivalent QPS conditions. This difference was particularly marked at higher QPS levels, underscoring the benefits of protocol optimization in high-demand scenarios.

Performance evaluation under various CPU core counts As depicted in Fig. 3-(b), there is a discernible correlation between the augmentation of CPU core count and the enhanced performance of the consensus protocol. More specifically, the throughput of the protocol, both pre- and post-optimization, exhibits a positive trend with the incremental addition of CPU cores, which is concurrently accompanied by a reduction in latency. Furthermore, the performance of the optimized version of the protocol significantly surpasses that of the PBFT protocol in terms of latency minimization. Particularly, in a system configuration utilizing a 32-core CPU, the proposed protocol manifests a considerable reduction in response time, amounting to 15,838 milliseconds when contrasted with a configuration employing a 4-core CPU. This empirical evidence substantiates the hypothesis that an increase in core count serves to enhance both the efficiency and the responsiveness of the protocol.

The experimental outcomes substantiate that an increase in CPU core count exerts a positive influence on both the maximum throughput and the reduction in latency within a Software Guard Extensions (SGX) environment. Specifically, for the enhanced protocol, the maximum throughput observed on an 8-core CPU exhibited an increase of 280.84% relative to that on a 4-core CPU, while the throughput on a 16-core CPU showed an increment of 138.5% compared to the 8-core CPU. Concurrently, the data indicate that the performance of the proposed consensus protocol significantly improves as the available hardware resources are augmented. Hence, this protocol demonstrates robust scalability and distinct performance advantages. Moreover, even in environments constrained by hardware resources, our consensus protocol maintains a substantial performance superiority over traditional Practical Byzantine Fault Tolerance (PBFT) protocols, highlighting its effectiveness and efficiency in various computational settings.



QPS	500	1000	1500	2000	2500	5000	7500	10000
■ TPS(PBFT)	504	1005	1006	1045	1047	662	874	883
■ TPS(TEP-BFT)	509	1032	1546	1247	1261	849	934	971
● Avg Response Time(PBFT)	839	2967	5394	8318	8206	13821	17564	21821
● Avg Response Time(TEP-BFT)	508	728	1438	5720	6052	11110	13978	17110



Number of Cores	4-Core	8-Core	16-Core	32-Core
■ TPS(PBFT)	883	2899	7860	9825
■ TPS(TEP-BFT)	971	3698	8820	10500
● Avg Response Time(PBFT)	21821	16943	10687	5068
● Avg Response Time(TEP-BFT)	17110	8559	3930	1272

Fig. 3 The latency and throughput performance comparison of the consensus protocol w/wo optimization. (a) Performance on a 4-core CPUs under various QPS. (b) Performance on various CPU core counts under 10,000 QPS

6.3 How the number of nodes impacts on the latency and throughput under the optimal request load

This experiment, employing a 16-core CPU, sought to investigate the relationship between node count and system performance metrics, namely latency and throughput, under a load of 10,000 requests. The experimental design included configurations of 3, 5, 7, and 9 nodes to evaluate the impact of varying node counts on system performance. As illustrated in Fig. 4-(a), the results demonstrate a progressive decrease in system throughput and a corresponding increase in latency as the number of nodes escalates. The principal factors contributing to these trends include network communication overhead, protocol role switching, and message confirmation processes.

However, compared to the pre-optimized PBFT, TEP-BFT exhibits substantial enhancements in both throughput and latency, especially at higher node counts. Notably, under such conditions, TEP-BFT achieves a reduction in response time by 12,258 milliseconds compared to PBFT. This improvement is primarily attributed to the reduction in the number of communication phases from three to two and a decrease in the required number of nodes to tolerate f Byzantine nodes from $3f+1$ to $2f+1$. As a result, while PBFT necessitates $12f^2+10f+2$ instances of communication, TEP-BFT requires merely $2f^2+5f+2$, reflecting a substantial decrease of $10f^2+5f$. Thus, as node count increases, the response time advantage of TEP-BFT becomes increasingly pronounced.

In summary, although an augmentation in node count introduces additional network communication and protocol processing overhead, which in turn leads to reduced throughput and heightened latency, the application of a comprehensive suite of performance optimization strategies can still markedly enhance system performance. These optimizations bolster the processing capabilities and response speed of the system. Furthermore, our protocol demonstrates a significant performance superiority compared to the pre-optimized PBFT, underscoring its efficacy in complex computational environments.

6.4 How the number of Byzantine nodes impacts on the latency and throughput under the optimal request load

In this experiment, utilizing a server equipped with a 16-core CPU, we explore the impact of the number of Byzantine nodes on system latency and throughput under optimal request load conditions. Considering the system's fault tolerance limitations, for a system configuration set to 9 nodes, a maximum of 4 Byzantine nodes are allowed. By simulating Byzantine behavior-namely simulating faults by halting the operations of normal nodes-the experiment aims to assess the impact of different numbers of Byzantine nodes on system performance. The experimental results are shown in Fig. 4-(b). The data indicate that as the number of Byzantine nodes increases, system throughput decreases while latency increases. This outcome is expected and reflects the direct impact of Byzantine nodes' negative influence on system performance, primarily including the following two aspects:

Influence of Byzantine nodes The presence of Byzantine nodes introduces increased uncertainty and complexity within the system, particularly when these nodes occupy pivotal roles such as the primary node. The system is compelled to engage in additional processes to detect and mitigate malicious behaviors, ensuring the uninterrupted operation of the

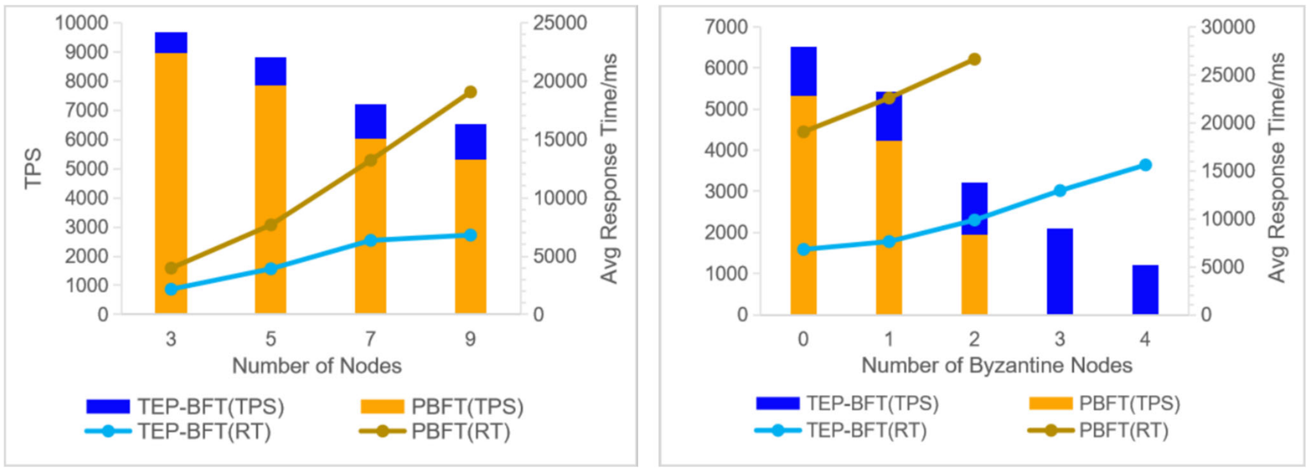


Fig. 4 The impact of the number of nodes on throughput and latency under a 16-core CPU condition. (a) The number of total nodes when there is no byzantine nodes, (b) The number of Byzantine nodes

network. These supplementary activities consume system resources and diminish processing efficiency.

Performance detriments due to view changes In Byzantine fault-tolerant systems, the view change mechanism is critical for sustaining system operability when the primary node is compromised by Byzantine failures. This mechanism necessitates comprehensive coordination among all nodes, suspending ongoing transactions until a new primary node is elected and recognized by the network. Although essential for maintaining continuity, this process significantly impedes system performance during the transition.

Despite these challenges, compared to the Practical Byzantine Fault Tolerance (PBFT) protocol, our protocol exhibits superior robustness in maintaining system liveness and security with up to 4 Byzantine nodes. As illustrated in Fig. 4-(b), while PBFT can only accommodate a maximum of two Byzantine nodes in a nine-node setup, increasing beyond this threshold disrupts consensus. Moreover, the communication overhead during view changes in PBFT is greater than in our TEP-BFT protocol, resulting in prolonged response times and diminished throughput. Consequently, our approach not only reduces the node deployment costs but also enhances scalability and fault tolerance, offering substantial improvements over traditional PBFT systems.

6.5 Consensus performance comparison w/wo trusted execution environments

This study, conducted using a server with a 16-core CPU, evaluates the system throughput in both confidential and non-confidential computing environments. As shown in Fig. 5, it is evident that in non-confidential environments, the optimal TPS exceeds 20,000. However, in confidential computing environments, the optimal TPS is around 8,800. This indicates that using the SGX framework inevitably leads to a performance decline. Nevertheless, as CPU performance improves, the throughput in confidential environments gradually increases, and the TPS can surpass 10,000. This suggests that the performance drop is not solely due to the CPU but also influenced by the Gramine framework. The following factors primarily contribute to the observed performance decrease in the confidential environment:

Performance Overhead of the Gramine Framework: The Gramine framework introduces additional overhead in the

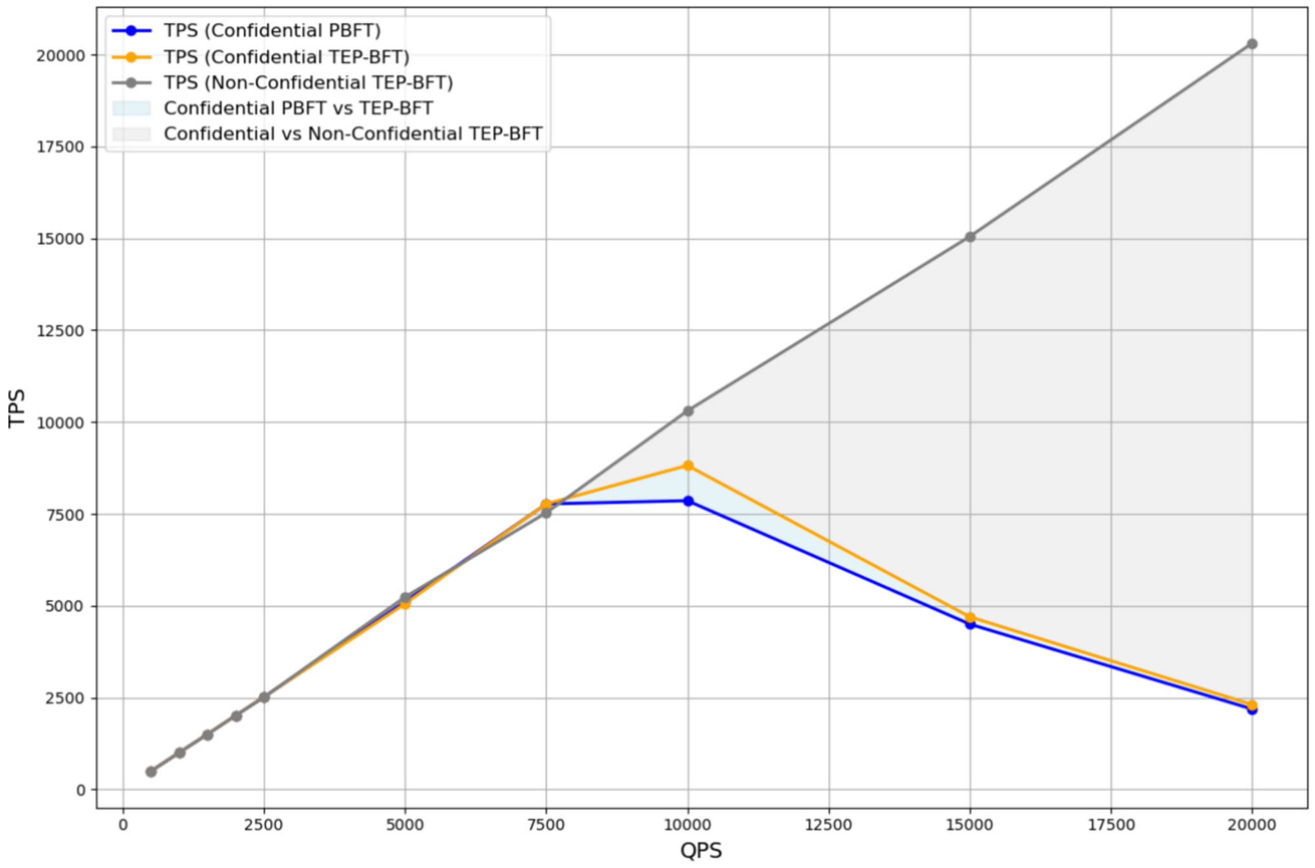


Fig. 5 The performance comparison in confidential and non-confidential environments under 16-core CPU conditions

execution of multi-process applications, particularly affecting system call handling and inter-process communication (IPC), which significantly degrades performance.

System Call Overhead: In the SGX environment, the lack of copy-on-write and memory sharing mechanisms necessitates that the Gramine framework emulate traditional Linux process forking via a more intricate checkpoint and recovery mechanism. This process involves serializing, encrypting, and transmitting the entirety of a parent process's memory and resources to a child process, which subsequently receives, decrypts, and integrates it into its own enclave memory. This method incurs considerably more overhead than traditional copy-on-write operations.

Inter-Process Communication Overhead: In the Gramine framework, all IPC operations require transparent encryption or decryption, typically employing TLS-PSK and AES-GCM encryption methods. The associated encryption overhead further contributes to the performance reduction.

Despite these setbacks, the throughput achieved in a confidential computing environment remains adequate to meet industry standards, enhanced by improved CPU performance. By integrating the PBFT protocol within a confidential computing framework, we can ensure both the integrity of node code and the security of the runtime environment. Additionally, the Gramine framework's support facilitates the porting of applications without necessitating code reengineering, substantially easing development challenges. This balance of security and performance offers a practical approach for deploying confidential computing in critical applications, such as blockchain technologies.

6.6 Performance comparison with state-of-the-art BFTs

In this study, we implemented three consensus protocols based on TEE-, MinBFT [29], CheapBFT [15], and FastBFT [23]-within our system, according to their respective principles, and compared their throughput and latency against our TEP-BFT protocol. As depicted in Fig. 6-(a), with an increase in QPS, the throughput for each protocol improves; however, However, compared to other existing protocols, TEP-BFT shows significant advantages in throughput. Furthermore, as the QPS increases, its throughput advantage becomes even more pronounced. Notably, at higher QPS levels, TEP-BFT demonstrates superior scalability. For example, at a QPS of 7500, TEP-BFT has not yet reached its maximum throughput, while the throughput of other protocols not only plateaus but begins to decline at a threshold of 2000 QPS. Regarding latency, TEP-BFT also maintains a significant advantage. Its latency curve ascends gradually with an increase in QPS, whereas the latency of other protocols escalates more sharply, particularly at higher QPS levels. When the QPS exceeds 2000, the other protocols show a clear upward trend in latency. Conversely, when QPS does not exceed 5000, the latency of the TEP-BFT protocol remains stable and low, underscoring its enhanced scalability and performance under high-load conditions.

As shown in the Fig. 6-(b), with the increase in the number of Byzantine nodes, the throughput and response times of all BFT protocols are affected to varying degrees. However, TEP-BFT demonstrates a significant advantage in both throughput and response time compared to other BFT

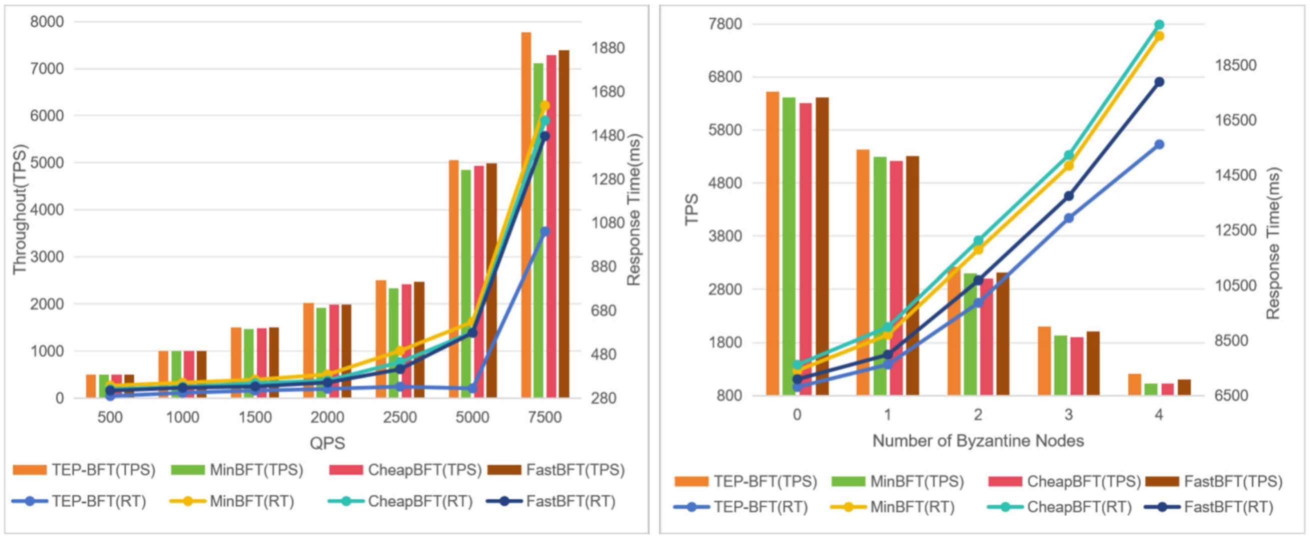


Fig. 6 The performance comparison of various BFT protocols in a confidential environment on a 16-core CPU. (a) QPS; (b) Number of Byzantine Nodes

protocols. Notably, the response time advantage of TEP-BFT becomes more pronounced as the number of Byzantine nodes increases. This suggests that the TEP-BFT protocol exhibits stronger robustness when faced with Byzantine faults and significantly improves the efficiency of handling Byzantine nodes.

The primary reason for these experimental results is related to the design optimizations of TEP-BFT. TEP-BFT employs a more efficient consensus mechanism, which allows it to reach consensus faster when there are fewer Byzantine nodes, resulting in better TPS performance. However, as the number of Byzantine nodes increases, the overhead of handling more complex Byzantine scenarios also increases, leading to higher response times. In contrast, other protocols may suffer from a lack of parallel optimizations or inherent complexity, causing their performance to degrade more significantly in the presence of Byzantine nodes.

The distinct performance advantage of TEP-BFT is primarily attributed to its parallel processing architecture. By facilitating parallel operations both among and within consensus threads, TEP-BFT effectively utilizes resources to increase throughput and decrease latency. This mechanism enables the protocol to sustain high throughput and low latency even under significant transaction loads, which is crucial for systems demanding high performance and real-time responsiveness.

6.7 Discussion

Scalability challenges in large networks TEP-BFT demonstrates significant scalability improvements compared to traditional BFT protocols, especially when dealing with mid-sized networks. However, in extremely large networks, potential bottlenecks arise, particularly due to the increasing complexity of inter-node communication. As the network size grows, maintaining low latency and high throughput becomes challenging due to the quadratic growth in communication overhead inherent in BFT protocols. This bottleneck is partially mitigated by TEP-BFT's multi-level parallel processing capabilities, which enable the protocol to handle multiple transactions concurrently. However, beyond a certain network size, even this parallelism may not fully compensate for the communication overhead. One possible solution to this issue is to implement hierarchical or shard-based architectures, which can reduce the communication burden on any single node by dividing the network into smaller, more manageable sub-networks.

Practical implementation challenges While TEP-BFT excels in scenarios requiring high throughput and low latency, its reliance on specific hardware, such as Intel SGX, poses practical challenges for deployment in real-world blockchain systems. The dependency on SGX limits the protocol's applicability to environments where this hardware is available and compatible. Ensuring that the system remains secure and efficient while handling parallel processes can be challenging, particularly in systems with less predictable workloads. To address the practical challenges associated with TEP-BFT's dependency on specific hardware like Intel SGX, we emphasize that the protocol is not limited to Intel SGX but can be implemented on any device supporting Trusted Execution Environments (TEEs). This includes, but is not limited to, platforms such as Arm TrustZone, Trusted Platform Modules (TPM), and RISC-V Keystone. By utilizing a range of TEE-supporting hardware, TEP-BFT can be more flexibly deployed across different environments.

7 Conclusion

In this paper, we introduced TEP-BFT, a novel Byzantine Fault Tolerance (BFT) protocol leveraging a Trusted Execution Environment (TEE) based on Intel SGX. By utilizing a trusted monotonic counter, TEP-BFT simplifies the consensus process to two communication phases (Prepare and Commit) and reduces the number of required replicas to $2f + 1$ while maintaining the security and liveness of traditional BFT protocols. Our extensive analysis and evaluation demonstrated that TEP-BFT significantly reduces latency and enhances throughput compared to other BFT variants. Additionally, the protocol exhibits robust scalability, with slower throughput degradation as network size increases, making it a strong candidate for blockchain applications requiring frequent transactions.

For future work, we plan to explore the implementation of hierarchical or shard-based architectures to further enhance scalability by reducing the communication burden on individual nodes in large networks. Additionally, we will focus on strengthening the security of TEP-BFT by investigating its resilience against potential vulnerabilities, such as side-channel attacks, and integrating support for alternative TEE platforms like Arm TrustZone and TPM. These efforts will ensure TEP-BFT's applicability across diverse hardware environments and enhance its robustness. Finally, we aim to develop consensus protocols for blockchain networks with varying topologies, such as graph structures, to mitigate performance bottlenecks in diverse application scenarios.

Appendix A: Performance evaluation

A.1 How QPS impacts on CPU and memory utilization

This case study evaluates the impact of varying Query Per Second (QPS) rates on CPU load. A comparative analysis of CPU core counts was conducted using industrial servers equipped with 4/8/16/32-core CPU configurations under a load of 10,000 QPS. The results are as follows:

The impacts on CPU workload under different QPS Figure 7-(a) illustrates that, before protocol optimization, the average load on the primary node consistently exceeds 4 under all QPS levels within one minute, indicating prolonged resource contention and resulting in extended response times. At lower QPS levels, replica nodes maintain a load below 4, but surpass this threshold as QPS increases. The CPU load utilization exceeds 100%, operating beyond capacity and risking system failure at any moment. Following protocol optimization, the load across all QPS levels diminishes, particularly at higher QPS where the optimization significantly alleviates system pressure. Despite nearing 100% load rate, there is a marked improvement in performance compared to pre-optimization conditions.

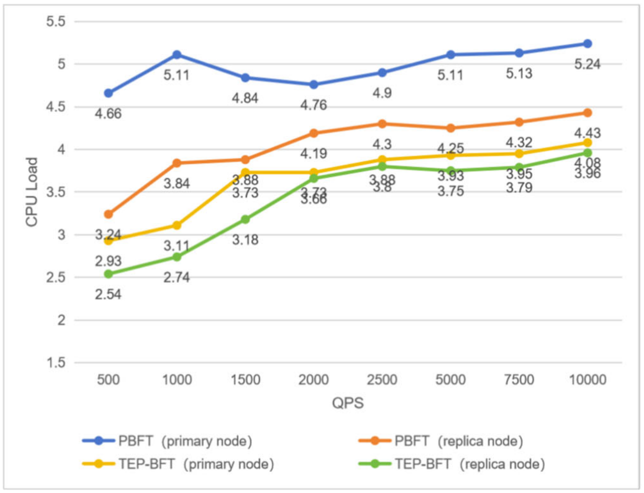
These observations underscore the positive effect of protocol optimization on system performance, particularly under high QPS conditions. This enhancement reduces the system's load rate without additional hardware resources, thereby augmenting the system's capability to manage high volumes of concurrent requests. The achievements in improving system throughput and ensuring stability and responsiveness underscore the success of our optimization strategy in enhancing resource utilization efficiency and system robustness.

The impacts on CPU workload under different CPU core counts As depicted in Fig. 7-(b), data indicate that with an increasing number of CPU cores, average CPU utilization rates for the unoptimized PBFT protocol are consistently high across smaller core counts but decrease with larger configurations: 100%, 100%, 49.7%, and 33.7% respectively. In contrast, the TEP-BFT protocol demonstrates lower average CPU utilizations of 99.6%, 93.1%, 44%, and 28.9% respectively. These figures reveal that an increase in CPU core count effectively reduces CPU utilization and enhances protocol performance, with TEP-BFT consistently showing lower CPU utilization across various core configurations.

Based on these experimental results, it is evident that in an SGX environment, the consensus efficiency of nodes is profoundly influenced by the CPU performance. Experiments on 4-core CPU demonstrate significant system strain under all measured metrics-TPS, system latency, and system load. With enhancements in CPU performance moving from 4-core to 8-core and then to 16-core CPU, as illustrated in Fig. 7-(a) and (b), the system's TPS continuously improves while the system load rate declines, indicating an enhanced capacity to manage trusted and untrusted area messages under high pressure. Notably, under similar conditions, TEP-BFT exhibits significant performance improvements compared to traditional PBFT, achieving efficient consensus even on less capable hardware.

The impacts on memory under different CPU core counts The experimental setup involved configuring Gramine to allocate 8GB of memory to each trusted container, analyzing system memory usage and its potential impact on performance under different CPU core counts. The results are as follows:

Memory usage Figure 8-(a) shows that as QPS increases, operating system memory usage rises while user memory remains stable, a phenomenon observed in 8-core and 16-core CPU configurations (Figs. 8-(b) and (c)). This indicates that higher QPS affects the utilization of trusted memory under the same Gramine settings. However, compared to



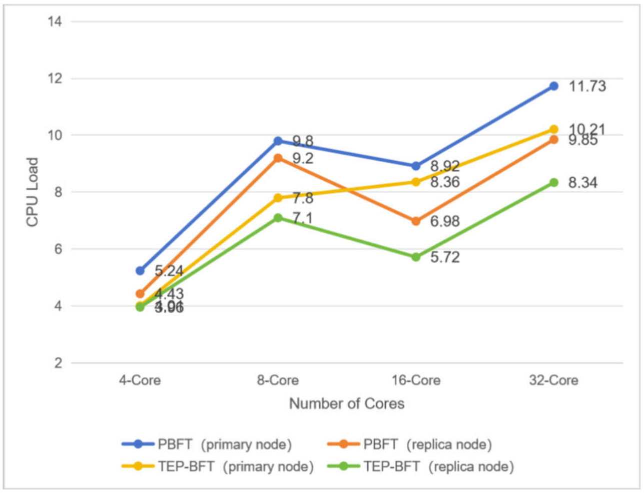


Fig. 7 The CPU workload performance comparison of the consensus protocol w/wo optimization. (a) Performance on a 4-core CPUs under various QPS. (b) Performance on various CPU core counts under 10,000 QPS

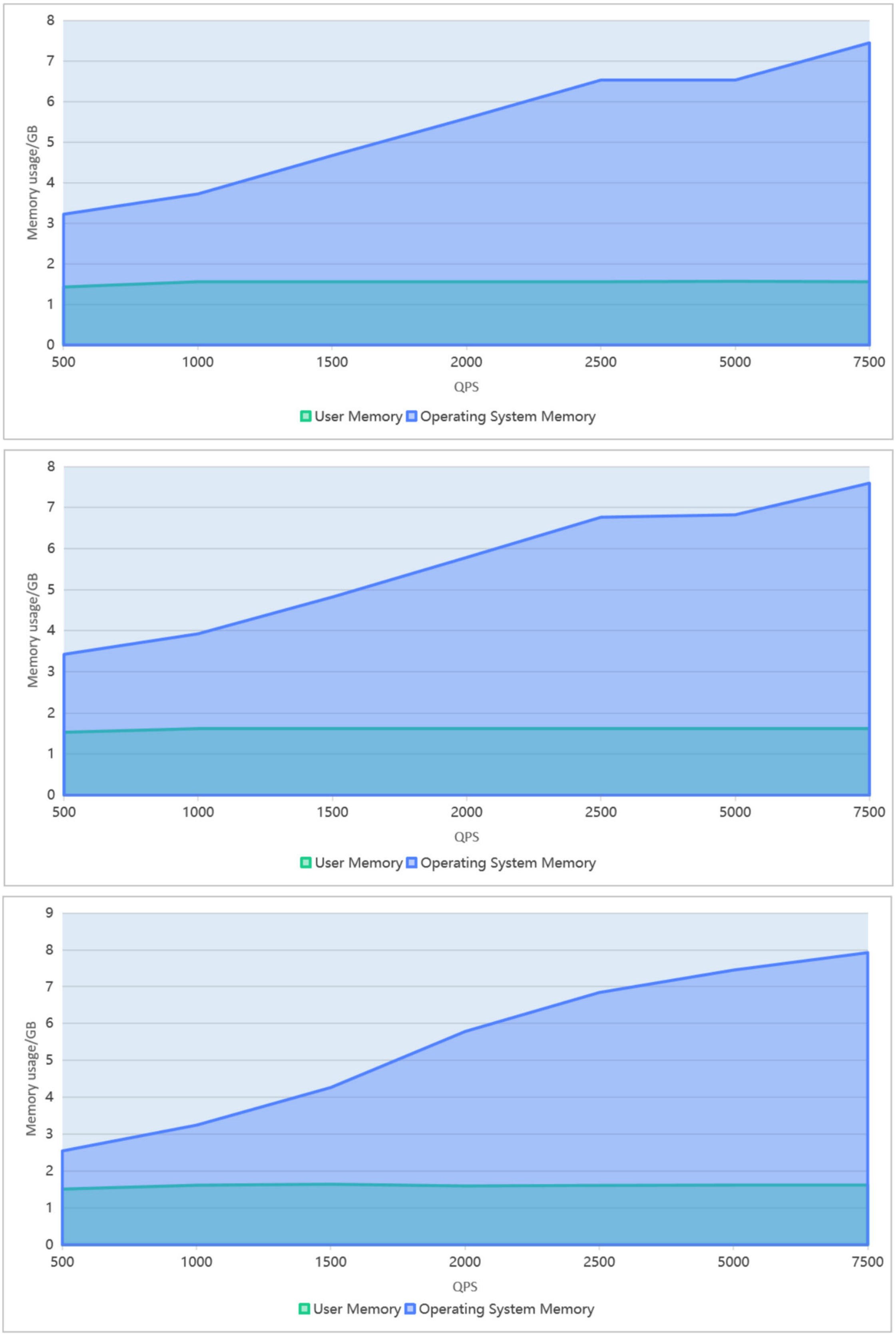


Fig. 8 The comparison of operating system memory usage under (a) 4-Core CPU, (b) 8-Core CPU, and (c) 16-Core CPU

CPUs, the direct impact of trusted memory on performance appears more limited.

Enclave memory overhead In the SGX protected execution environment (Enclave), memory overhead is relatively small, primarily encompassing code, data, and heap. While direct observations of trusted memory changes are not feasible, analysis of user space memory and operating system memory variations can infer the impact of trusted memory. SGX has minimal effect on user space memory, but kernel space memory increases with QPS.

In conclusion, although Enclave memory usage is relatively small in an SGX environment, an increase in QPS leads to higher operating system memory usage. This indirectly indicates that the utilization rate of trusted memory is influenced by QPS, although its direct impact on performance may not be substantial. Additionally, the rise in Enclave memory overhead is mainly due to the complexity of code, the volume of data structures and buffers required, and the storage of security-related information. Thus, optimizing code and data management within the Enclave, along with appropriate Gramine configuration settings, is essential for controlling memory usage and enhancing system performance.

Funding This work is supported in part by Shandong Provincial Natural Science Foundation under Grant ZR2022LZH013, in part by the National Key Research and Development Program of China under Grant 2021YFB3702403, in part by the National Natural Science Foundation of China under Grant 62101029, and in part by the China Scholarship Council Award under Grant 202006465043 and 202306460078. Cheng Xu and Fuqiang Ma are both corresponding authors.

Data Availability No datasets were generated or analysed during the current study.

Code Availability Our code is made public available at: https://github.com/SICC-Group/TEP-BFT.git.

Declarations

Consent to publish All authors approved the final manuscript and the submission to this journal.

Ethics Approval Not applicable.

Competing Interests The authors declare no competing interests.

References

1. (2024) FISCO BCOS. http://fisco-bcos.org/zh/, accessed: April 2024
2. Castro M, Liskov B (2002) Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) 20(4):398–461
3. Castro M, Liskov B, et al (1999) Practical byzantine fault tolerance. In: OsDI, pp 173–186
4. Chun BG, Maniatis P, Shenker S et al (2007) Attested append-only memory: Making adversaries stick to their word. ACM SIGOPS Operating Systems Review 41(6):189–204
5. Correia M, Neves NF, Lung LC et al (2005) Low complexity byzantine-resilient consensus. Distrib Comput 17(3):237–249
6. Distler T, Kapitza R (2011) Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. In: Proceedings of the sixth conference on Computer systems, pp 91–106
7. Distler T, Cachin C, Kapitza R (2016) Resource-efficient byzantine fault tolerance. IEEE Trans Comput 65(9):2807–2819
8. Ekberg JE, Kostiainen K, Asokan N (2014) The untapped potential of trusted execution environments on mobile devices. IEEE Security & Privacy
9. Garcia M, Bessani A, Gashi I, et al (2011) Os diversity for intrusion tolerance: Myth or reality? In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), IEEE, pp 383–394
10. Gervais A, Karame GO, Wüst K, et al (2016) On the security and performance of proof of work blockchains. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 3–16
11. Hyperledger (2024) Hyperledger Fabric. https://www.hyperledger.org/projects/fabric, accessed: April 2024
12. Intel (2023) Intel® Software Guard Extensions (SGX) Programming Reference. Tech. rep., Intel Corporation, accessed: [insert date here]
13. Javaid M, Haleem A, Singh RP, et al (2021) Blockchain technology applications for industry 4.0: A literature-based review. Blockchain: Research and Applications 2(4):100027
14. Kapengut E, Mizrach B (2023) An event study of the ethereum transition to proof-of-stake. Commodities 2(2):96–110
15. Kapitza R, Behl J, Cachin C, et al (2012) Cheapbft: Resource-efficient byzantine fault tolerance. In: Proceedings of the 7th ACM european conference on Computer Systems, pp 295–308
16. Kim Thomas (2017) On the transaction cost of Bitcoin. Finance Res Lett 23:300–305
17. Kotla R, Dahlin M (2004) High throughput byzantine fault tolerance. In: International Conference on Dependable Systems and Networks, 2004, IEEE, pp 575–584
18. Kotla R, Alvisi L, Dahlin M et al (2010) Zyzzyva: Speculative byzantine fault tolerance. ACM Transactions on Computer Systems (TOCS) 27(4):1–39
19. Levin D, Douceur JR, Lorch JR, et al (2009) Trinc: Small trusted hardware for large distributed systems. In: NSDI, pp 1–14
20. Li H, Chen Y, Shi X, et al (2023) Fisco-bcos: An enterprise-grade permissioned blockchain system with high-performance. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–17
21. Li T, Wang H, He D et al (2022) Blockchain-based privacy-preserving and rewarding private data sharing for iot. IEEE Internet Things J 9(16):15138–15149
22. Li Y, Qiao L, Lv Z (2021) An optimized byzantine fault tolerance algorithm for consortium blockchain. Peer-to-Peer Networking and Applications 14:2826–2839
23. Liu J, Li W, Karame GO et al (2018) Scalable byzantine consensus via hardware-assisted secret sharing. IEEE Trans Comput 68(1):139–151
24. Martin JP, Alvisi L (2006) Fast byzantine consensus. IEEE Trans Dependable Secure Comput 3(3):202–215
25. McKeen F, Alexandrovich I, Berenzon A, et al (2013) Innovative instructions and software model for isolated execution. In: HASP
26. Miller A, Xia Y, Croman K, et al (2016) The honey badger of bft protocols. In: Proceedings of the 2016 ACM SIGSAC Conference

on Computer and Communications Security, ACM, Vienna, Austria, pp 31–42
27. Obelheiro RR, Bessani AN, Lung LC, et al (2006) How practical are intrusion-tolerant distributed systems? Technical Report
28. Veronese GS, Correia M, Bessani AN et al (2010) Ebawa: Efficient byzantine agreement for wide-area networks. 2010 IEEE 12th International Symposium on High Assurance Systems Engineering. IEEE, San Jose, CA, USA, pp 10–19
29. Veronese GS, Correia M, Bessani AN et al (2011) Efficient byzantine fault-tolerance. IEEE Trans Comput 62(1):16–30
30. Vukolić M (2016) The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: Open Problems in Network Security: IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers, Springer, pp 112–125
31. Xu J, Wang C, Jia X (2023) A survey of blockchain consensus protocols. ACM Comput Surv 55(13s):1–35
32. Yao W, Ye J, Murimi R, et al (2021) A survey on consortium blockchain consensus mechanisms.arXiv:210212058
33. Yin M, Malkhi D, Reiter MK, et al (2019) Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pp 347–356
34. Yu K, Tan L, Aloqaily M et al (2021) Blockchain-enhanced data sharing with traceable and direct revocation in iiot. IEEE Trans Industr Inf 17(11):7669–7678
35. Zhang C, Wu C, Wang X (2020) Overview of blockchain consensus mechanism. In: Proceedings of the 2020 2nd International Conference on Big Data Engineering, ACM, Shanghai, China, pp 7–12

Ran Wang received the B.E. degree from the Beijing Information Science and Technology University, China in 2013, and the M.S. degree from the University of Science and Technology Beijing (USTB), China in 2016. She is currently working toward the Doctoral degree at University of Science and Technology Beijing. Her research interests include quantum optimization, distributed security and internet

Fuqiang Ma received the B.E. and Ph.D. degree from the University of Science and Technology Beijing (USTB), China in 2015 and 2010 respectively. He is currently working as a security researcher at Inspur Electronic Information Industry Co. Ltd and Jinan Inspur Data Technology Co., Ltd. His research interests now include information security, confidential computing and operating system. He is a member of the IEEE.

Sisui Tang is currently pursuing the master's degree with the University of Science and Technology Beijing, Beijing, China. Her research interests include distributed security, patter recognition, and Internet of Things.

Hangning Zhang is currently pursuing the master's degree with the University of Science and Technology Beijing, Beijing, China. Her research interests include distributed security, patter recognition, and Internet of Things.

Jie He received B.E. and Ph.D degree in computer science from University of Science and Technology Beijing (USTB), China in 2005 and 2012, respectively. Since July 2015, he has been an associate professor with the School of Computer and Communication Engineering, USTB since 2015. From April 2011 to April 2012, he was a visiting Ph.D student in Center for Wireless Information Network Studies, Worcester Polytechnic Institute. His research interests include wireless indoor positioning, human gesture recognition and motion capture.

Zhiyuan Su received the B.E. and Ph.D. degree from the Dalian University of Technology (DUT), China in 2008, and 2014 respectively. He is currently working as an operating system architect at Inspur Electronic Information Industry Co. Ltd. His research interests now include information security and operating system.

Xiaotong Zhang received the M.S., and Ph.D. degrees from University of Science and Technology Beijing, in 1997, and 2000, respectively. He was a Professor in the Department of Computer Science and Technology, University of Science and Technology Beijing. His research includes work in quality of wireless channels and networks, wireless sensor networks, networks management, cross-layer design and resource allocation of broadband and wireless network, signal processing of communication and computer architecture.

Cheng Xu received the B.E., M.S. and Ph.D. degree from the University of Science and Technology Beijing (USTB), China in 2012, 2015 and 2019 respectively. He is currently working as an associate professor in the Data and Cyber-Physical System Lab (DCPS) at University of Science and Technology Beijing. He is supported by the Post-doctoral Innovative Talent Support Program from Chinese government in 2019. He is an associate editor of International Journal of Wireless Information Networks. His research interests now include swarm intelligence, multi-robots network, wireless localization and internet of things. He is a member of the IEEE.

Authors and Affiliations

Ran Wang[1,5] · Fuqiang Ma[2,3,4] · Sisui Tang[1] · Hangning Zhang[1] · Jie He[1] · Zhiyuan Su[4] · Xiaotong Zhang[1] · Cheng Xu[1,6]

✉ Fuqiang Ma
mafuqiang@ieisystem.com

✉ Cheng Xu
xucheng@ustb.edu.cn

Ran Wang
wangran423@foxmail.com

Sisui Tang
tangsisui@163.com

Hangning Zhang
hangning0117@163.com

Jie He
hejie@ustb.edu.cn

Zhiyuan Su
suzhiyuan@ieisystem.com

Xiaotong Zhang
zxt@ies.ustb.edu.cn

1 School of Computer and Communication Engineering, University of Science and Technology Beijing, 100083 Beijing, China

2 Jinan Inspur Data Technology Co., Ltd., 250101 Jinan Shandong, China

3 State Key Laboratory of High-end Server & Storage Technology, 100085 Beijing, China

4 Inspur Electronic Information Industry Co. Ltd., 250101 Jinan Shandong, China

5 College of Computing and Data Science, Nanyang Technological University, 639798 Singapore, Singapore

6 School of Electrical and Electronic Engineering, Nanyang Technological University, 639798 Singapore, Singapore