RESEARCH ARTICLE

Parallel Byzantine Fault Tolerance Consensus for Blockchain Secured Swarm Robots

Ran Wang[1,2] | Fuqiang Ma[3,4,5] | Sisui Tang[1] | Zhiyuan Su[5] | Cheng Xu[1,2]

[1]School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China | [2]Shunde Innovation School, University of Science and Technology Beijing, Beijing, China | [3]Jinan Inspur Data Technology Co. Ltd., Jinan, China | [4]State Key Laboratory of High-end Server & Storage Technology, Beijing, China | [5]Inspur Electronic Information Industry Co. Ltd., Jinan, China

Correspondence: Cheng Xu (xucheng@ustb.edu.cn)

Received: 22 June 2024 | **Revised:** 27 December 2024 | **Accepted:** 7 June 2025

Funding: This study is supported in part by National Natural Science Foundation of China (NSFC) under Grant 62101029, in part by Shandong Provincial Natural Science Foundation under Grant ZR2022LZH013, and in part by the China Scholarship Council Award under Grants 202006465043 and 202306460078.

Keywords: blockchain | Byzantine consensus | fault tolerance | swarm robotics

ABSTRACT

Establishing common knowledge about environmental conditions, task objectives, and coordination rules is crucial for improving the collaborative efficiency of swarm robots. In complex scenarios, relying on a centralized facility to maintain this knowledge is impractical, necessitating a decentralized approach. Blockchain technology offers a promising solution for decentralization and can tolerate some degree of malicious or malfunctioning entities. However, widely used blockchain approaches, such as those employed in Ethereum and relying on proof-of-work (PoW) or proof-of-authority (PoA), demand significant computational resources, rendering them impractical for swarm robotics applications. This paper introduces PTEE-BFT, a novel parallel Byzantine fault tolerance protocol leveraging the trusted platform module (TPM). PTEE-BFT employs a Unique Sequential Identifier Generator (USIG) to ensure the monotonicity, uniqueness, and order of messages, thereby reducing the number of communication phases and replicas required. This significantly enhances the efficiency and fault tolerance of the consensus process. Additionally, PTEE-BFT implements parallel processing strategies to substantially increase blockchain system throughput. Furthermore, we develop an algorithm that enables the robot swarm to recognize attacks from a specific type of malicious robot known as Byzantine robots. Our experimental analysis and performance evaluation demonstrate that PTEE-BFT achieves an optimal balance among performance, scalability, and fault tolerance, outperforming practical Byzantine fault tolerance (PBFT). Results from physical robots show that our approach significantly reduces computing overhead and accelerates consensus formation compared to baseline solutions. This represents a significant advancement in blockchain consensus mechanisms for swarm robotics.

1 | Introduction

Swarm robotics, involving networks of cooperative autonomous robots designed to perform tasks collectively, holds significant practical and research value (Debie et al. 2023; Nguyen et al. 2020). Inspired by natural systems such as ant colonies and bee swarms, the collective behavior of these robots finds applications in agriculture, search-and-rescue operations, and complex industrial processes. As the deployment of swarm robotics expands, ensuring their reliable and safe operation in challenging or unpredictable environments becomes critically important (Abouelyazid 2023).

To achieve secure operation and consistent decision-making in swarm robots, blockchain technology has been introduced to

[Correction added on 18 October 2025, after first online publication: The author's affiliations 1 and 2 have been updated.]

ensure that all nodes in the network agree on the transaction and data state (Xu et al. 2023). Blockchain's decentralized and immutable characteristics provide a reliable framework for swarm robots to address these challenges. The immutability of blockchain ensures the integrity of all transactions and data records, with each robot node maintaining a copy of the entire network state, enabling immediate detection of any data tampering. Furthermore, the decentralized structure of blockchain enhances the fault tolerance and stability of the system, ensuring continuous operation even when some nodes fail or are attacked (Thakur et al. 2023).

However, current research primarily focuses on blockchain consensus mechanisms to achieve transaction consensus rather than decision consensus (Bao et al. 2023). This has led to a misunderstanding of blockchain's application in swarm robotics, conflating transaction consensus with decision consensus. Transaction consensus ensures that all nodes agree on transaction records, guaranteeing data integrity and tamper resistance (Singh et al. 2020). In contrast, decision consensus focuses on achieving agreement on actions or strategy decisions among multiple robots, a crucial aspect of swarm robotic systems (Yuan and Ishii 2022, 2024).

Existing studies often fail to distinguish between transaction consensus and decision consensus, leading to confusion about their roles and application scenarios (Strobel et al. 2020, 2023). This confusion not only impacts the understanding of blockchain's potential advantages in swarm robotics but also hinders the optimization and improvement of research targeting different application scenarios. Future research should focus on distinguishing these two consensus mechanisms and investigating their specific applications in swarm robotics to fully leverage blockchain technology's potential in ensuring safe and consistent decision-making in swarm robots. This paper primarily focuses on the efficiency and fault tolerance of blockchain consensus mechanisms, as they form the foundation for decision consensus. The efficiency and Byzantine fault tolerance (BFT) capabilities of blockchain consensus mechanisms determine the overall efficiency and robustness of the decision-making process in swarm robotic systems.

Traditional blockchain consensus algorithms, such as proof of work (PoW) and proof of stake (PoS), cannot meet the unique needs of swarm robotics, such as real-time performance and resource constraints (Gervais et al. 2016). Recent studies have explored lightweight consensus mechanisms tailored for the resource-constrained environments typical of swarm robotics, such as BFT (Driscoll et al. 2003; Distler 2021). These mechanisms focus on achieving rapid consensus with minimal computational overhead, ensuring that the swarm can operate efficiently in real-time applications. Therefore, BFT consensus protocols are particularly suitable for environments requiring quick consensus while allowing relatively low fault thresholds.

The classical approach to BFT consensus involves algorithms like practical BFT (PBFT) (Castro and Liskov 1999). However, PBFT requires $3f + 1$ replicas, which must be diverse (different operating systems, software) to withstand attacks and intrusions, thereby increasing the additional costs associated with more replicas (hardware, software development, management, and so on). Additionally, PBFT can only scale to a limited number of nodes because it needs to exchange $O(n^2)$ messages among n servers to reach consensus (Ahmad et al. 2021). PBFT often does not scale efficiently in the context of swarm robotics, which may involve hundreds to thousands of nodes with high mobility and frequent state changes (Krishnamohan 2022). Therefore, enhancing the scalability and performance of BFT protocols and reducing fault tolerance costs are crucial for their practical deployment in swarm robotic systems.

Based on this analysis, we introduce a novel parallel BFT protocol based on trusted execution environments (PTEE-BFT) that not only reduces communication phases through trusted counters but also supports parallel operations within and between consensus threads. Specifically, the main contributions are summarized as follows:

1. **BFT Consensus:** We propose PTEE-BFT for swarm robotics, utilizing a TPM-based unique sequential identifier generator (USIG) to generate unique identifiers, ensuring the monotonicity, uniqueness, and orderliness of messages. This reduction in communication phases enhances consensus efficiency. Additionally, PTEE-BFT requires only $2f + 1$ nodes in the consensus process to withstand attacks from f Byzantine nodes, effectively reducing the cost of tolerating intrusions.
2. **Multi-Level Parallel Processing:** PTEE-BFT introduces multi-level parallel processing, enabling parallel operations between transaction packaging and consensus threads, as well as within the consensus threads. This includes parallel processing during block batch production and block pipeline execution phases. Such multi-level parallelism significantly enhances the efficiency of swarm robotics.
3. **Experimental Validation:** We conducted practical deployment tests in a collective decision-making scenario where the robot swarm moves on a floor covered with black and white tiles and determines the relative frequency of the white tiles in a ROS2 environment. Experimental results indicate that PTEE-BFT surpasses PBFT in efficiency, scalability, and fault tolerance, confirming its effectiveness in swarm robotics scenarios and demonstrating superior operational capabilities.

The structure of this paper is as follows: Section 2 details existing BFT consensus protocols and their limitations. Section 3 introduces the framework of the PTEE-BFT protocol. Section 4 describes the design and working mechanism of the PTEE-BFT protocol, the normal operation process, and view switching. Section 5 presents the performance advantages of PTEE-BFT through experiments and performance analysis. Finally, Section 6 summarizes the main contributions of this paper and outlines future research directions.

2 | Related Work

Swarm robotics, characterized by decentralized control and the absence of centralized failure points, underscores the importance of robust consensus mechanisms to facilitate collective decision-making. Key attributes of swarm robots include scalability, flexibility, and the ability to accomplish complex tasks through simple individual behaviors and local interactions. Nevertheless, these systems face significant security challenges due to their distributed nature, rendering them vulnerable to individual robot failures or malicious attacks that could compromise the collective outcome.

Strobel et al. (2020) demonstrated how a swarm of robots can achieve consensus in the presence of Byzantine robots by leveraging blockchain technology. However, they did not distinguish between blockchain consensus and consistency decisions, conflating the two concepts and treating consistency decisions as part of blockchain consensus. Moreover, employing the PoW consensus protocol results in poor efficiency and scalability for swarm robotics systems. Addressing these challenges, Song et al. (2023) presented a distributed swarm system utilizing PBFT to ensure secure and reliable coordination among small UAVs, enabling them to perform various missions while maintaining fault tolerance and resilience against malicious attacks. Krishnamohan et al. (2022) surveyed existing blockchain consensus algorithms and their suitability for swarm robotics, including proof of resource, proof of authority, and Byzantine agreement. Their survey highlighted the limitations of current blockchain consensus algorithms and concluded that a novel consensus approach is required for swarm robot systems. However, these solutions primarily focus on blockchain consensus for resisting Byzantine behaviors, neglecting the efficiency of blockchain consensus mechanisms.

Swarm robots require consensus protocols that maintain high fault tolerance while supporting scalable and efficient operations. Key requirements for these protocols include low latency and minimal communication overhead. Traditional PBFT has been associated with low efficiency and high costs. Consequently, some research efforts aim to reduce the number of nodes and communication phases to enhance consensus performance. Kotla et al. proposed Zyzzyva (Kotla et al. 2010), which utilizes speculation to enhance performance. Under normal operating conditions, Zyzzyva reduces the overhead of state machine replication to nearly optimal levels. However, if the primary node errs, it must switch to the PBFT view change process, which does not offer a clear advantage against Byzantine node attacks. Distler et al. (2016) introduced a resource-efficient BFT (ReBFT) replication architecture, where only a subset of replicas runs the consensus protocol under normal conditions. Despite its improvements, ReBFT shares similar drawbacks with Zyzzyva, trading off security for efficiency, and its effectiveness diminishes under malicious node attacks. FastBFT protocol (Liu et al. 2018) balances computational and communication loads by arranging nodes in a tree topology. FastBFT adopts an optimistic BFT paradigm (Distler et al. 2015), requiring only a portion of the active nodes to participate in the consensus. However, FastBFT relies on a relatively stable cluster environment, where malicious nodes can intentionally trigger member replacement in tree topology communication.

To address these challenges, some researchers have moved away from optimistic paradigms to improve the efficiency of consensus protocols, instead opting for asynchronous consensus protocols. HotStuff (Yin et al. 2019) implements linear leader changes using CPU pipelining concepts. This mechanism reduces the communication complexity of the BFT protocol to $O(n)$, enhancing scalability. However, proposals in this protocol require three rounds of interaction before they can be committed, adding extra communication delay. Miller et al. (Miller et al. 2016) proposed HoneyBadgerBFT, an asynchronous consensus protocol devoid of a specific primary node. However, due to the iterative nature of asynchronous BFT consensus mechanisms, achieving a final consensus result incurs higher latency.

Several studies have explored using hardware security to reduce the number of replicas and communication phases in BFT protocols (Veronese et al. 2011; Chun et al. 2007; Correia et al. 2005; Kapitza et al. 2012; Levin et al. 2009; Veronese et al. 2010; Wang et al. 2024). For instance, MinBFT (Veronese et al. 2011) utilizes a trusted counter within the primary node to assign sequence numbers to client requests, leveraging the TEE's guarantee of counter monotonicity to prevent assigning the same counter value to different messages. Consequently, the communication phases are reduced from three to two. Similarly, MinZyzzyva uses TEEs to decrease the number of replicas required in Zyzzyva, maintaining the same number of communication phases (Veronese et al. 2011). MinBFT and MinZyzzyva are highly efficient under this metric, operating with the minimum known communication steps for nonspeculative and speculative protocols, respectively four (Martin and Alvisi 2006) and three steps (Kotla et al. 2010). CheapBFT (Kapitza et al. 2012) 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. Nevertheless, CheapBFT involves transitioning between three distinct consensus protocols, increasing the complexity of the BFT programming model.

3 | Framework and Preliminaries

As shown in Figure 1, the PTEE-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 PTEE-BFT framework:

Nodes. The consensus framework consists of three types of nodes: robot clients, primary nodes, and replica nodes. Robot clients can send transaction requests at any time, storing them in the transaction pool. Primary node elections occur in a Round-Robin manner (Castro and Liskov 1999). When the primary node is suspected to be faulty, a new primary node is elected through the Viewchange mechanism. The primary node receives client requests and broadcasts them to all replica nodes in a predefined order. After executing the request, each replica node returns the results to the primary node, which aggregates and sends the final response to the client. All non-primary nodes are replica nodes, responsible for executing client requests as instructed by the primary node. Each replica node maintains the system's status and operation logs independently, ensuring proper functionality even if the primary node fails. Replica nodes provide redundancy and fault tolerance by executing requests and returning results to the primary node.

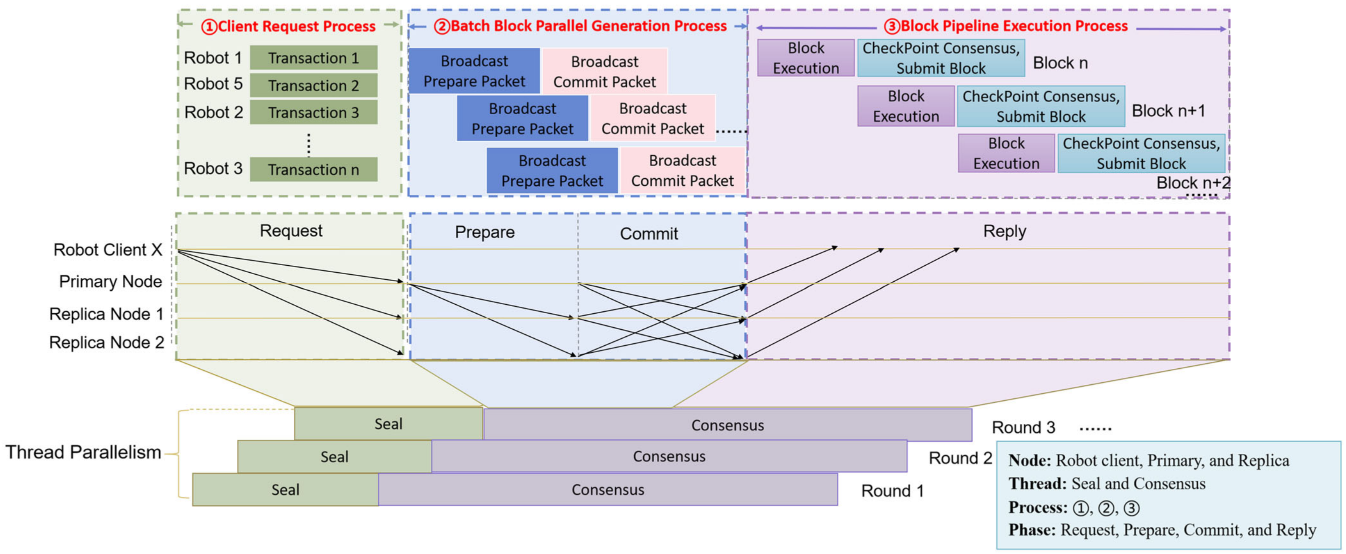

FIGURE 1 | The structure includes three distinct processes, facilitating multiple client interactions and transaction consensus simultaneously. In process ①, multiple robot clients concurrently submit transaction consensus requests, referred to as Request packets. The primary node aggregates transactions from the transaction pool and organizes them sequentially. This operation is managed by the *Seal* thread, which systematically packages 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, enhancing 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 ensures that blocks are executed correctly and timely. Upon successful validation, the results are disseminated back to the robot clients via Reply messages. This structured approach allows the PTEE-BFT to efficiently handle multiple transactions and consensus requests concurrently, optimizing throughput and reducing latency within the network.

Threads. Consensus on a block involves two primary processes: transaction packaging and consensus. These processes are managed by the *Seal* and *Consensus* threads. The *Seal* thread retrieves transactions from the transaction pool and packages them based on the highest block on the node, creating new blocks. These new blocks are then passed to the *Consensus* thread. The *Consensus* thread receives new blocks, either locally or through the network, and completes the consensus process based on the received messages, ultimately writing the new consensus-approved blocks into the blockchain. Once a block is added to the blockchain, the transactions it contains are removed from the transaction pool. Since the *Seal* and *Consensus* threads require nonconflicting resources, they operate independently and can process in parallel. If the *Seal* thread takes time s and the *Consensus* thread takes time c, 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. Firstly, the client request process sends the transaction request messages to primary and replica nodes. Then, the batch block generation process is responsible for sorting transactions received in the transaction pool and parallel generation of sorted yet unexecuted blocks. Finally, the block pipeline execution process conducts pipeline consensus on the block execution results and commits blocks that have successfully reached consensus.

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. All nodes then 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 based on TPM, ensuring that each identifier can be assigned to only one message, 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, PTEE-BFT eliminates the **Prepare** phase of the traditional PBFT protocol. Regarding 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 identifier and correct certificate. Thus, PTEE-BFT requires the participation of only $2f + 1$ nodes in the consensus workflow to withstand f Byzantine nodes.

4 | PTEE-BFT Protocol

This section primarily explains the implementation principles and specific procedural steps of PTEE-BFT, explaining both the normal operational flow of the PTEE-BFT protocol and the view change process when nodes fail. The PTEE-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. At the same time, PTEE-BFT is reduced from three-phase communication (Pre-prepare/Prepare/Commit) to two-phase communication (Prepare/Commit). To further improve the performance of PTEE-BFT, we also introduce a parallel processing mechanism.

4.1 | Client Request Process

The workflow of client request process is shown in Figure 2. A *robot client* initiates an operation, denoted as *op*, by sending a message to all servers:

$$\langle \text{Request}, c, seq, op, sig_c \rangle. \tag{1}$$

Each client possesses a unique pair of public and private keys. The private key is used to sign requests. 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. As shown in Figure 2, after sending a request, the client waits for *Reply* messages with matching results rst_{ji} from $f+1$ different nodes:

$$\langle \text{Reply}, S_j, seq, rst_{ji} \rangle, \tag{2}$$

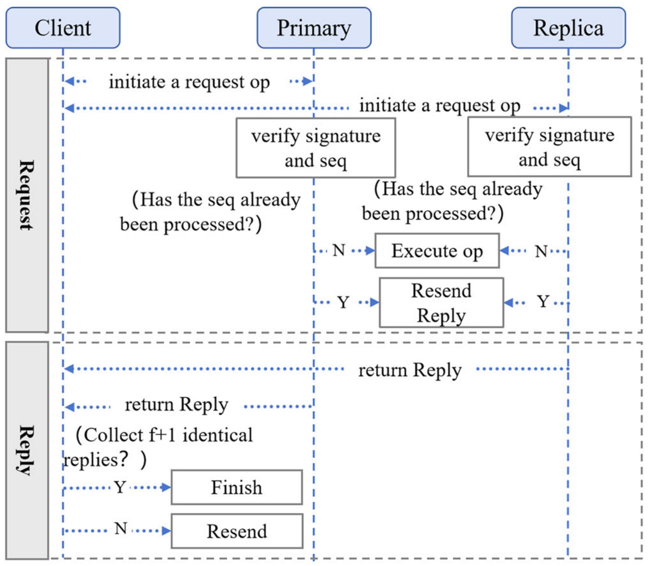

FIGURE 2 | The workflow chart of request and reply processes.

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.

4.2 | Batch Block Parallel Generation Process

During the process, every message uses the USIG based on the TPM to generate a unique identifier. The core function of USIG is to generate a unique sequence number for incoming messages and to create a signature in conjunction with the message content. This ensures the integrity and verifiability of the messages. USIG operates within a TPM, safeguarding it from external attacks and ensuring the monotonicity, uniqueness, and sequentiality of messages, with each sequence number corresponding to only one message. The value of the monotone counter maintained by USIG is given by the block height, instead of maintaining a separate counter. Let the current block height of the blockchain be h. Both primary and replica nodes generate a key pair within a trusted execution environment (TEE) during system initialization. The private key is securely stored and used to create a unique identifier within this environment, inaccessible to any external entities, while the public keys are publicly available for verification of a unique identifier.

As shown in Figure 3, PTEE-BFT for the specific operation process is as follows:

1. 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. The primary node retrieves several packaged blocks from the transaction pool, denoted as $\text{Blocks} = [\text{Block}_1, \text{Block}_2, \ldots, \text{Block}_i, \ldots, \text{Block}_{\text{BlockLimit}}]$, and places these blocks in the *Prepare* message packet, producing $\text{PreMessages} = [\text{Prepare}_1, \text{Prepare}_2, \ldots, \text{Prepare}_i, \ldots, \text{Prepare}_{\text{BlockLimit}}]$. *BlockLimit* is a parameter that limits the number of blocks that can be concurrently consensed, ensuring the stability of the blockchain system. 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:

$$\langle \text{Prepare}, v, S_p, Block_i, UI_{p_i} \rangle, \tag{3}$$

where UI_{p_i} is a unique identifier generated by the USIG.

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. If the verification is successful, they continue to check the following:

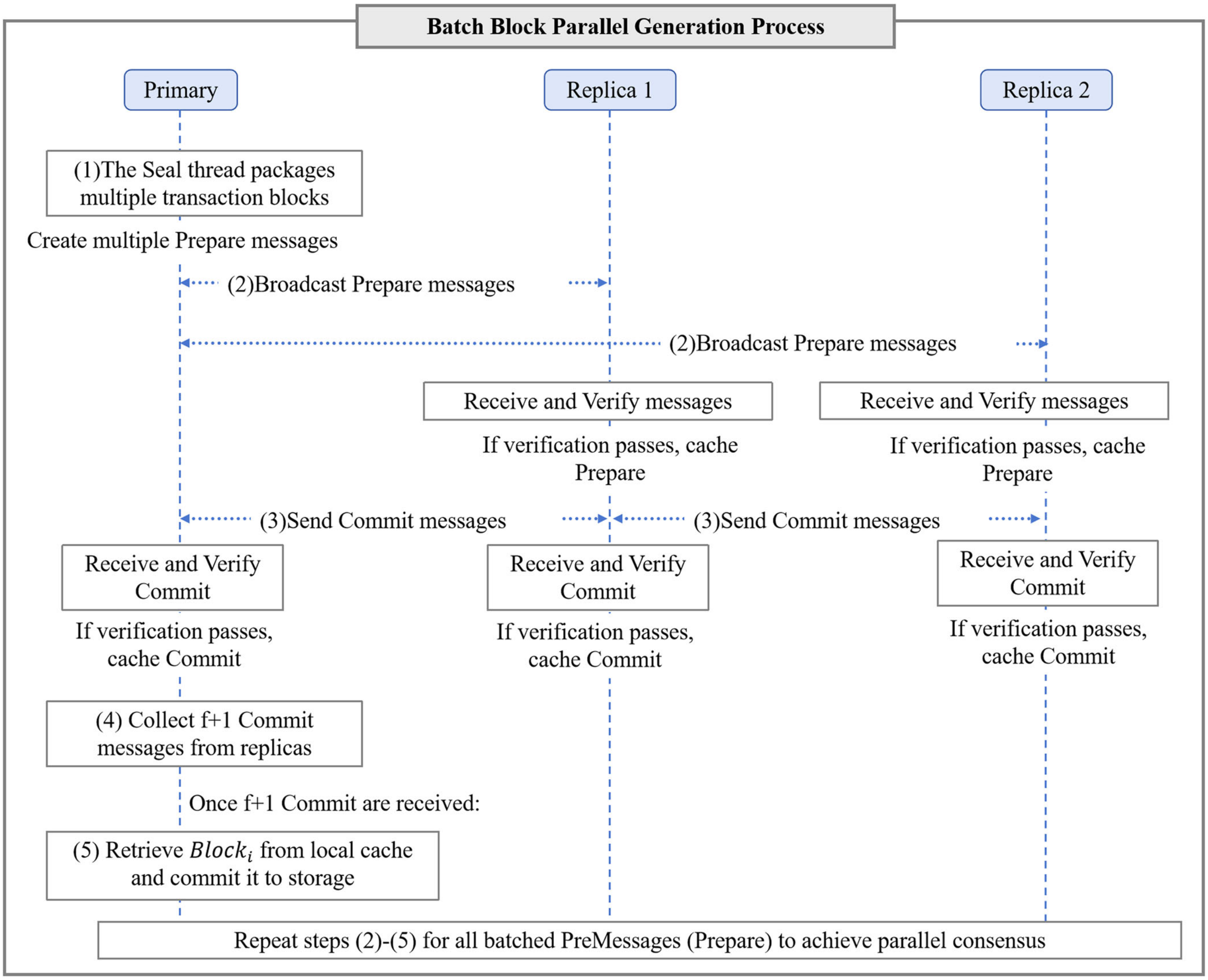
Batch Block Parallel Generation Process
Primary
Replica 1
Replica 2
(1)The Seal thread packages multiple transaction blocks
Create multiple Prepare messages
(2)Broadcast Prepare messages
(2)Broadcast Prepare messages
Receive and Verify messages
Receive and Verify messages
If verification passes, cache Prepare
If verification passes, cache Prepare
(3)Send Commit messages
(3)Send Commit messages
Receive and Verify Commit
Receive and Verify Commit
Receive and Verify Commit
If verification passes, cache Commit
If verification passes, cache Commit
If verification passes, cache Commit
(4) Collect f+1 Commit messages from replicas
Once f+1 Commit are received:
(5) Retrieve $Block_i$ from local cache and commit it to storage
Repeat steps (2)-(5) for all batched PreMessages (Prepare) to achieve parallel consensus

FIGURE 3 | The workflow chart of batch block parallel generation process.

- 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 $\text{UI}_i.h$ 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 $h + \text{BlockLimit}$.

3. After a replica node successfully verifies a *Prepare* message packet, it adds the packet to its local cache and broadcasts *Commit* message packet to all other nodes:

$$\langle \text{Commit}, v, Sr_j, S_p, Block_i, UI_{p_i}, UI_{r_{ji}} \rangle, \tag{4}$$

where *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.

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.
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.

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, $\mathbf{V}_{acp} = (UI_{p_1}.h, UI_{p_2}.h, UI_{p_3}.h, \ldots, UI_{p_n}.h)$, where $UI_{p_n}.h$ is the counter value of the last message sent by the primary node received by the current replica node.

4.3 | 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. As shown in Figure 4, the protocol for this process is 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 $cPHash_i$.
2. After block execution, nodes generate a *Checkpoint* message packet:

$$\langle \text{Checkpoint}, S_j, UI_{latest}, cPHash_i, UI_{cj} \rangle, \quad (5)$$

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.

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.

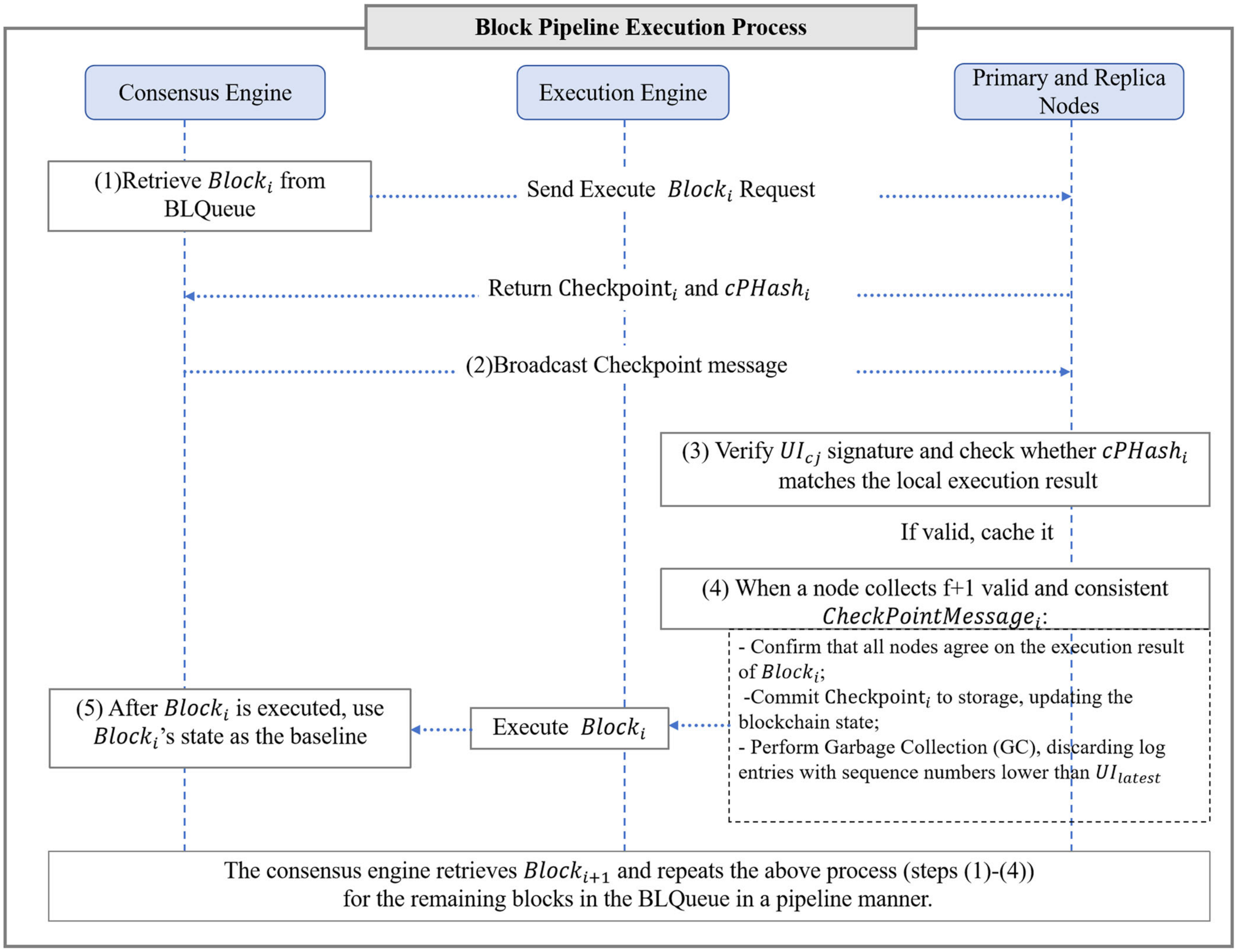
Block Pipeline Execution Process
Consensus Engine
Execution Engine
Primary and Replica Nodes
(1)Retrieve $Block_i$ from BLQueue
Send Execute $Block_i$ Request
Return Checkpoint$_i$ and $cPHash_i$
(2)Broadcast Checkpoint message
(3) Verify UI_{cj} signature and check whether $cPHash_i$ matches the local execution result
If valid, cache it
(4) When a node collects f+1 valid and consistent $CheckPointMessage_i$:
- Confirm that all nodes agree on the execution result of $Block_i$;
-Commit Checkpoint$_i$ to storage, updating the blockchain state;
- Perform Garbage Collection (GC), discarding log entries with sequence numbers lower than UI_{latest}
(5) After $Block_i$ is executed, use $Block_i$'s state as the baseline
Execute $Block_i$
The consensus engine retrieves $Block_{i+1}$ and repeats the above process (steps (1)-(4)) for the remaining blocks in the BLQueue in a pipeline manner.

FIGURE 4 | The workflow chart of block pipeline execution process.

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.
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*.

4.4 | View Change Operation

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 workflow of view change process is shown in Figure 5.

When the timer of a replica node Sr_j times out, Sr_j sends a message to all nodes:

$$\langle \text{ViewchangeReq}, Sr_j, v, v' \rangle \tag{6}$$

where v 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 $\langle \text{Viewchange}, Sr_j, v', cP_{latest}, M, UI_{vj} \rangle$, 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}

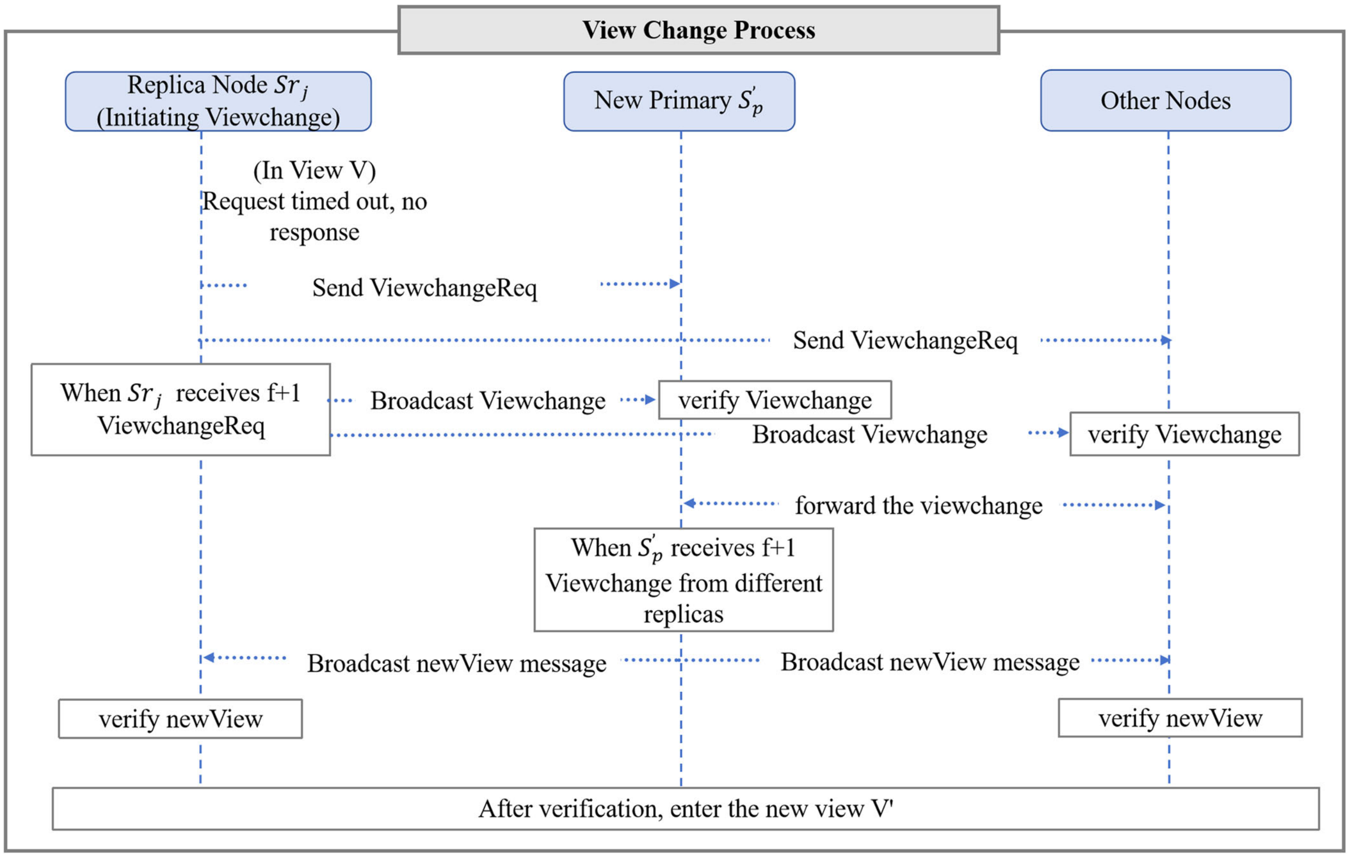
View Change Process
Replica Node Sr_j (Initiating Viewchange)
New Primary S_p'
Other Nodes
(In View V) Request timed out, no response
Send ViewchangeReq
Send ViewchangeReq
When Sr_j receives f+1 ViewchangeReq
Broadcast Viewchange
verify Viewchange
Broadcast Viewchange
verify Viewchange
forward the viewchange
When S_p' receives f+1 Viewchange from different replicas
Broadcast newView message
Broadcast newView message
verify newView
verify newView
After verification, enter the new view V'

FIGURE 5 | The workflow chart of view change process.

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 $\langle \text{Viewchange}, Sr_j, v', cP_{latest}, M, UI_{vj} \rangle$ message. Normal nodes perform the following checks on *ViewChange*:

1. cP_{latest} actually has $f + 1$ valid UI identifiers:

$$cP_{latest} = UI_1, UI_2, \ldots, UI_{f+1}. \tag{7}$$

2. In UI_{vj}, the counter value must increase by one:

$$UI_{vj}.h = UI_{vj-1}.h + 1. \tag{8}$$

   - If M is not empty, then counter value in $UI_{vj-1}.h$ is:

$$UI_{vj-1}.h = \max(M). \tag{9}$$

   - If M is empty, the counter value is taken from the latest checkpoint:

$$UI_{vj-1}.h = cP_{latest}.h. \tag{10}$$

3. The counter values in messages M are consecutive:

$$\forall\ m_i, m_{i+1} \in M, m_{i+1}.h = m_i.h + 1. \tag{11}$$

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:

$$NV_c = r|r.h > cP_{latest}.h, r \in M. \tag{12}$$

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:

$$\langle \text{newView}, S'_p, v', V_{nv}, NV_c, UI_n \rangle. \tag{13}$$

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.

5 | Experiments and Discussion

5.1 | Experiment Setup

In practical applications, blockchain consensus mechanisms significantly influence exit times and the system's ability to withstand Byzantine faults during collective decision-making processes. Therefore, the experiments presented in this paper primarily investigate the efficiency of blockchain consensus, the impact of hardware performance, and the resilience against Byzantine nodes. In our scenario, we compared the consensus efficiency of PTEE-BFT with PBFT and other enhanced BFT protocols by integrating the PTEE-BFT protocol into the FISCO BCOS blockchain (Li et al. 2023) and establishing a private network. Each participating machine hosted a single PTEE-BFT node. Field experiments were conducted using E-Puck 2 robots. At the core of each E-Puck 2 is a Raspberry Pi Zero 2 microcomputer, enhanced with a LetsTrust TPM module (pi3g 2024). This Trusted Platform Module (TPM) plays a crucial role in securing the data and operations of the robots by providing hardware-based security functions, which are essential in a collaborative environment where data integrity and security are paramount. The inclusion of the TPM enables the system to perform secure multiagent computations and authenticate inter-robot communications securely.

We evaluated the consensus efficiency of PTEE-BFT relative to PBFT and its resilience to Byzantine node attacks. The swarm's objective was to estimate the relative frequency of white tiles in a $1.5 \times 0.8\ \text{m}^2$ "checkerboard" environment, where the floor is covered with B black and W white tiles, each measuring $10 \times 10\ \text{cm}^2$, with $B + W = 400$ tiles, as depicted in Figure 6. Each robot performed obstacle avoidance and random walk routines on the floor. Depending on the scenario, the positions of the black and white tiles were either fixed by the experimenter or randomly assigned at the start of a run. At the beginning of each run, the robots' starting positions were randomly selected from a uniform distribution. To enable the swarm to aggregate information about the environment, each robot sampled its local ground sensor and exchanged information with other robots within its communication range. The experiment was conducted in discrete time steps, each corresponding to 1 s.

At each time step, robot i determines if it is above a black or a white tile via its ground sensor. Each robot operates in exploration phases. We use the subscript notation i, m for variables referring to robot i in its mth exploration phase. The duration of each exploration phase is 45 seconds. To obtain a sensor reading, robot i in its mth exploration phase

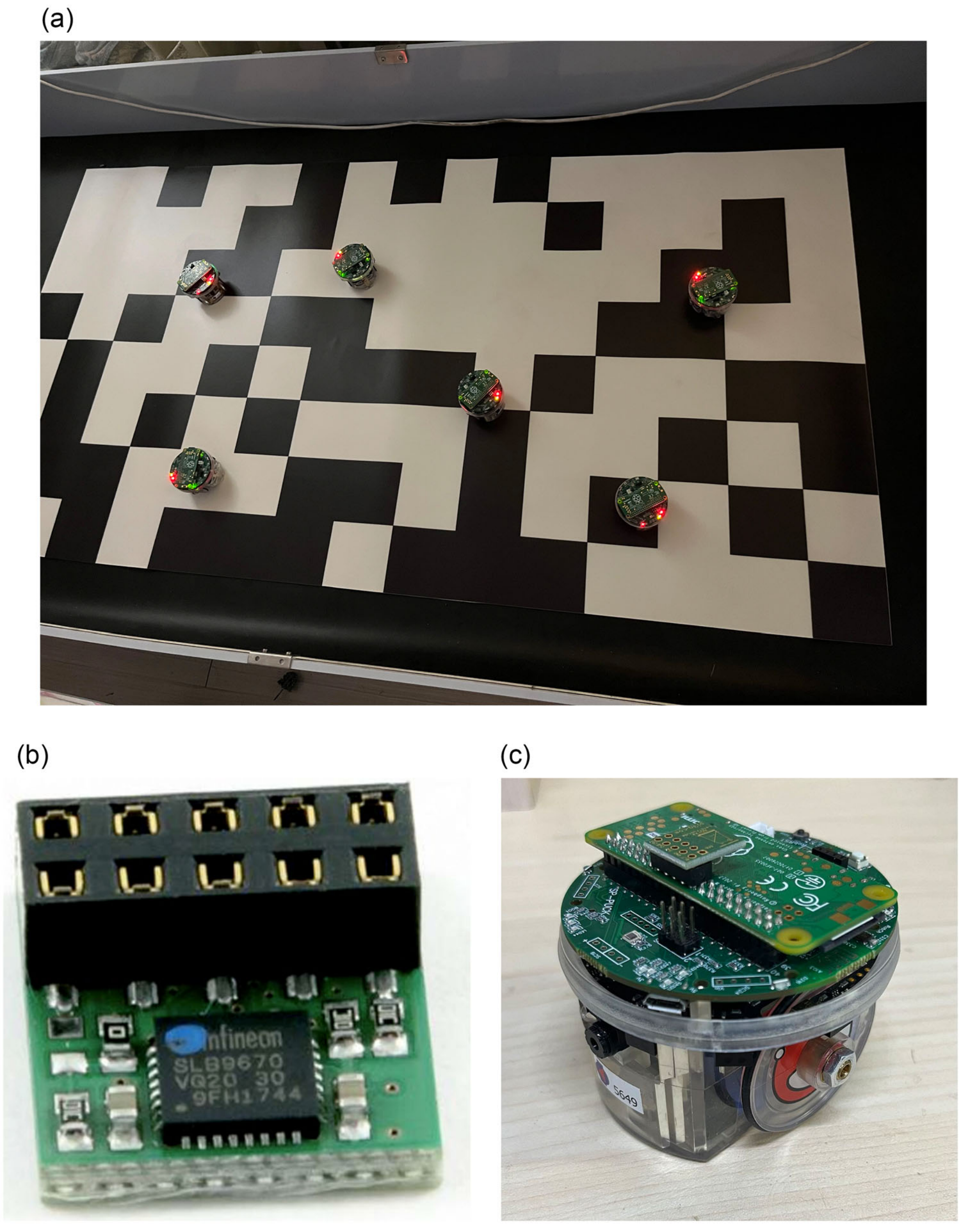

FIGURE 6 | The testbed physical environment: (a) The black-white tiles arena. (b) The LetsTrust TPM module. (c) The E-puck2 robot equipped with TPM.

calculates the ratio $\rho_{i,m}$ between the number of white tiles $W_{i,m}$ and the total number of tiles $W_{i,m} + B_{i,m}$ it sensed in this exploration phase: $\rho_{i,m} = W_{i,m}/(W_{i,m} + B_{i,m}) \in [0, 1]$. If the distance between two robots was less than 50 cm, they were within communication range and could exchange information, reflecting real swarm robotics systems that possess only local communication capabilities. This communication range resulted in an average degree of connectivity of 2.4, meaning one robot was, on average, connected to 2.4 other robots, leading to the formation of multiple non-connected clusters almost constantly. The ratio of black tiles was set at 0.30.

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 BFT protocol, highlighting the relationship between throughput and latency.
- *Exit time*: The average time to complete the common knowledge formation process, i.e. the average end time of each experiment.
- *Estimate Error*: The average error of the final common estimate value with the ground truth value.

5.2 | How Queries Per Second (QPS) Impacts on Transactions Per Second (TPS)

In the context of the white-black ratio consensus experiment involving swarm E-puck robots, QPS and TPS are crucial metrics for evaluating the performance of the consensus protocol. The following provides a detailed explanation of each term in this specific scenario:

QPS: QPS refers to the rate at which robots within the swarm submit requests for consensus on the white-black tile ratio. In this experiment, each robot periodically collects data from its ground sensors to determine whether it is above a black or a white tile. This data is then shared with other robots within its communication range, and a consensus mechanism is used to aggregate this information and reach a collective decision on the ratio of white to black tiles. For instance, if the QPS is set to 1000, it means that the entire swarm is collectively making 1000 requests per second to update and agree upon the ratio of white to black tiles.

TPS: TPS measures the number of consensus transactions that the system can process in 1 s. A transaction, in this context, refers to a successful update and agreement on the white-black tile ratio after processing the data shared by the robots. For example, if the system has a TPS of 500, it means that it can handle and process 500 consensus TPS. This involves collecting sensor data from the robots, running the consensus algorithm (such as PTEE-BFT or PBFT), and updating the collective decision on the tile ratio.

In the white-black ratio consensus experiment, E-puck robots are tasked with determining the proportion of white tiles in a given area by sampling the tiles they encounter. The procedure can be summarized as follows:

1. *Data Collection:* Each robot moves around the environment and uses its ground sensor to identify the color of the tile it is currently on (black or white).
2. *Communication and Query:* Robots within communication range share their sensor data with each other. This sharing is part of the consensus request, which is measured by QPS. Higher QPS means more frequent data sharing and consensus requests.
3. *Consensus Process:* The shared data is processed through the consensus algorithm to reach an agreement on the white-black tile ratio. Each successful agreement constitutes a transaction.
4. *Performance Metrics:* The efficiency of this process is evaluated using TPS, which indicates how many of these consensus agreements can be reached per second. Higher TPS means the system can handle more consensus transactions in a given time frame, reflecting better performance.

As illustrated in Figure 7, the experimental results clearly demonstrate that both PBFT and PTEE-BFT protocols show a steady increase in TPS as QPS increases, but their behaviors differ significantly at higher QPS levels. Specifically, the TPS for

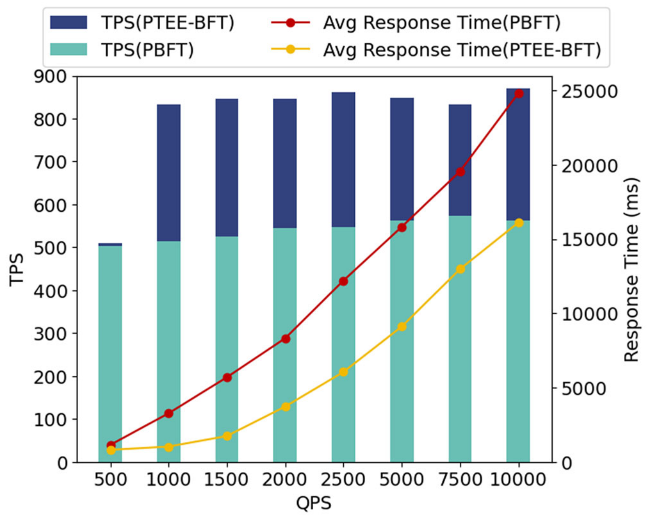

FIGURE 7 | The latency and throughput performance comparison of the consensus protocol with and without optimization.

PBFT stabilizes at around 500 TPS, whereas the PTEE-BFT protocol achieves significantly higher throughput, peaking at around 870 TPS. This indicates that PTEE-BFT can handle a larger volume of consensus transactions under high-demand scenarios, making it more suitable for real-time applications with frequent updates.

Furthermore, the ART for both protocols increases as QPS rises. However, PTEE-BFT consistently maintains a lower ART compared to PBFT across all QPS levels. For example, at QPS = 5000, the response time for PBFT exceeds 15,000 ms, while PTEE-BFT remains below 10,000 ms. This significant reduction in latency highlights the advantages of PTEE-BFT in scenarios where timely decision-making is critical, such as real-time swarm robotics or dynamic sensor networks.

This section explores the impact of transaction request rates (i.e., QPS) on throughput (i.e., TPS) and latency (ART) using Raspberry Pi Zero hardware. Initially, we evaluated the latency and throughput of the proposed PTEE-BFT method under varying QPS conditions. As illustrated in Figure 7, the preliminary phase of our study was conducted within the Raspberry Pi Zero, where all experimental nodes were configured within the TPM, ensuring a controlled and secure testing framework. The results demonstrated a direct correlation between transaction request rates (QPS) and throughput (TPS), with the throughput peaking at a QPS of 1500. Before optimizing the PBFT protocol, the system achieved a TPS of approximately 1000 within the TPM. However, post-optimization, the throughput significantly increased to around 1500 TPS, marking a notable 30% improvement in performance at the maximum QPS. Furthermore, latency measurements generally increased with rising QPS. Crucially, the optimized protocol consistently exhibited lower latency compared to its pre-optimized counterpart under equivalent QPS conditions, particularly at higher QPS levels. These findings underscore the practical applicability of the PTEE-BFT protocol in large-scale systems requiring high throughput and low latency, such as crowdsourced robotic coordination, where frequent data exchanges and rapid consensus updates are essential for achieving task synchronization and operational efficiency.

5.3 | Real-World Black-White Ratio Estimation Error

Figure 8 presents the estimate error of the consensus protocol over 1000 rounds, comparing the performance of the optimized PTEE-BFT protocol with the traditional PBFT protocol. The extended results further validate the superior performance of PTEE-BFT, particularly in achieving faster convergence and lower steady-state errors compared to PBFT.

Initially, both protocols start with a high estimate error, with PBFT exhibiting an error rate above 0.3 and PTEE-BFT slightly below 0.3. As the rounds progress, the estimated error for both protocols declines. However, the rate of decrease clearly differentiates the two protocols:

By around 200 rounds, the estimation error for PTEE-BFT decreases to approximately 0.05, while PBFT still remains around 0.08, indicating a faster convergence rate for PTEE-BFT. Between 200 and 400 rounds, the gap between the two protocols widens further. PTEE-BFT consistently reduces the error more rapidly, reaching approximately 0.02 by 400 rounds, whereas PBFT maintains a higher error of about 0.03. By around 600 rounds, the estimation error for both protocols stabilizes. However, PTEE-BFT achieves a lower steady-state error close to 0.01, while PBFT stabilizes at a slightly higher value near 0.02. The results demonstrate that the PTEE-BFT protocol not only accelerates the convergence process but also ensures a lower final estimation error, which is critical for applications requiring precise and reliable consensus.

Compared to PBFT, PTEE-BFT enhances consensus efficiency, allowing the system to achieve decision consistency among nodes more rapidly. This improvement can be attributed to the integration of TEEs, which enhance the reliability of data processing, and the parallel execution capabilities of PTEE-BFT, which significantly reduce communication overhead.

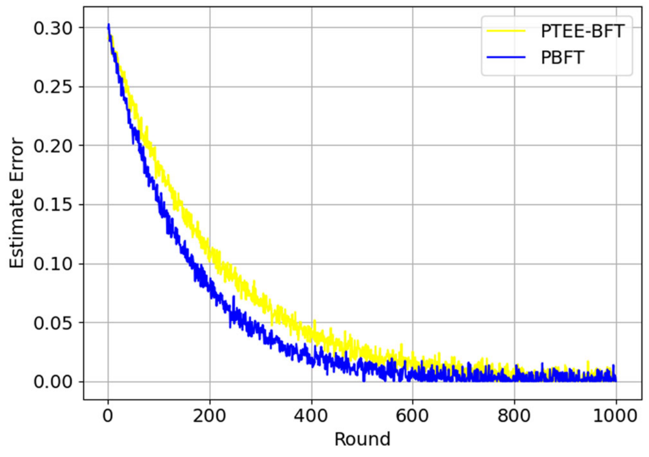

FIGURE 8 | Estimate error of consensus protocol with and without optimization.

In real-world scenarios, such as crowdsourced robotic systems or distributed sensor networks, the ability to achieve faster convergence and lower error rates ensures higher operational efficiency and accuracy. The experimental results confirm that PTEE-BFT is highly suitable, efficient, and scalable for systems with stringent real-time and accuracy requirements.

5.4 | How the Number of Byzantine Robots Impacts Exit Time

In our experiments, we employed the Linear Consensus Protocol (LCP) (Beal 2016) for consensus decision-making. As depicted in Figure 9a, our actual experimental outcomes indicate that the proposed PTEE-BFT protocol significantly reduces exit time compared to the PBFT protocol. The results demonstrate that PTEE-BFT outperforms PBFT in both throughput and response time. Consequently, the system can process more transaction data within a unit of time, substantially shortening overall response time and accelerating the centralized decision-making process. However, as the number of nodes continues to increase, the exit time will eventually reach a minimum and then start to rise again. This rebound effect is due to the increased communication costs associated with having a larger number of nodes.

Specifically, as shown in Figure 9a, for a system size of three nodes, the exit time for PBFT is approximately 528 ms, while PTEE-BFT reduces it to around 354 ms, representing a substantial improvement. As the number of nodes increases to 15, PTEE-BFT maintains an exit time below 240 ms, whereas PBFT exit time remains significantly higher at around 336 ms. This result highlights the superior scalability of PTEE-BFT in larger systems, where communication complexity tends to increase.

In this experiment, we examine the impact of the number of Byzantine nodes on exit time and estimate error. Considering the system's fault tolerance limitations, for a system configuration set to 15 nodes, a maximum of seven Byzantine nodes are allowed. By simulating Byzantine behavior—namely, by halting the operations of normal nodes—the experiment aims to assess the impact of varying numbers of Byzantine nodes on system resilience. The experimental results are shown in Figure 9b. When the system is resilient to Byzantine nodes, the final estimate error remains virtually unaffected. For instance, as shown in Figure 9b, when the number of Byzantine nodes increases from 0 to 7, the estimate error of PTEE-BFT remains stable at approximately 0.025, while PBFT exhibits a slight increase, particularly when exceeding four Byzantine nodes. This demonstrates that PTEE-BFT can sustain accurate consensus results even under adverse conditions.

However, the exit time is directly affected by the increasing number of Byzantine nodes. For PTEE-BFT, the exit time gradually rises from around 240 ms (0 Byzantine nodes) to 521 ms (7 Byzantine nodes), whereas PBFT experiences a more significant increase, rising from approximately 336 ms to over 600 ms. This result illustrates that PTEE-BFT effectively mitigates the performance degradation caused by Byzantine nodes,

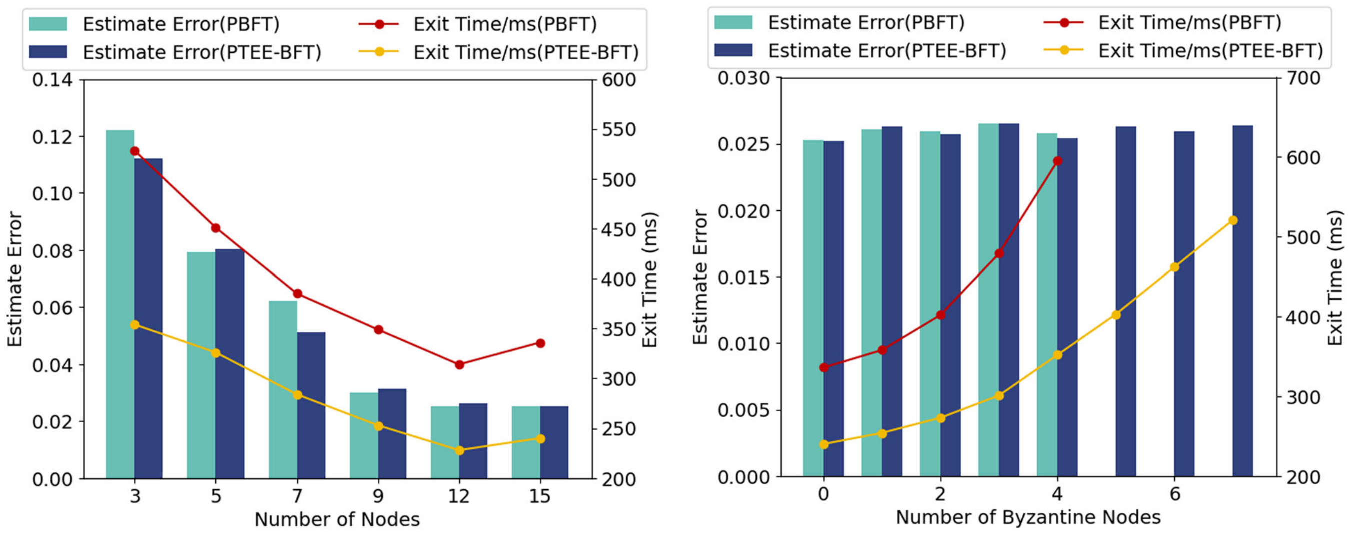

FIGURE 9 | System performance comparison before and after optimization of consensus protocols with varying node counts: (a) number of robot nodes; (b) number of Byzantine nodes.

ensuring timely consensus decisions even in fault-prone environments.

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 uninterrupted network operation. 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 PBFT protocol, our protocol exhibits superior robustness in maintaining system liveness and security with up to seven Byzantine nodes. This improvement is particularly critical in real-world applications such as autonomous robotic swarms and distributed IoT networks, where the presence of malicious or faulty nodes is inevitable. The ability of PTEE-BFT to maintain low estimate error and reasonable exit time under such conditions ensures higher operational reliability and fault tolerance. As illustrated in Figure 9b, while PBFT can only accommodate a maximum of four Byzantine nodes in a 15-node setup, increasing beyond this threshold disrupts consensus. Moreover, the communication overhead during view changes in PBFT is greater than in our PTEE-BFT protocol, resulting in prolonged response times and diminished throughput. Consequently, our approach not only reduces node deployment costs but also enhances scalability and fault tolerance, offering substantial improvements over traditional PBFT systems.

6 | Conclusion

In this paper, we proposed a novel PTEE-BFT to enhance the efficiency and scalability of consensus mechanisms in swarm robotics. Our approach reduces communication phases and leverages multi-level parallel processing, significantly improving throughput and reducing latency. Experimental results demonstrate that PTEE-BFT outperforms traditional PBFT variants in performance, scalability, and fault tolerance, making it well-suited for real-time applications in swarm robotics. Building on this study, future work will focus on optimizing consensus decision-making methods, further enhancing the protocol's efficiency and adaptability to diverse and complex operational scenarios in swarm robotics.

Acknowledgments

This study is supported in part by National Natural Science Foundation of China (NSFC) under Grant 62101029, in part by Shandong Provincial Natural Science Foundation under Grant ZR2022LZH013, and in part by the China Scholarship Council Award under Grants 202006465043 and 202306460078.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

Abouelyazid, M. 2023. “Adversarial Deep Reinforcement Learning to Mitigate Sensor and Communication Attacks for Secure Swarm Robotics.” *Journal of Intelligent Connectivity and Emerging Technologies* 8, no. 3: 94–112.

Ahmad, A., A. Alabduljabbar, M. Saad, D. Nyang, J. Kim, and D. Mohaisen. 2021. “Empirically Comparing the Performance of Blockchain's Consensus Algorithms.” *IET Blockchain* 1, no. 1: 56–64.

Bao, Q., B. Li, T. Hu, and X. Sun. 2023. "A Survey of Blockchain Consensus Safety and Security: State-of-the-Art, Challenges, and Future Work." *Journal of Systems and Software* 196: 111555.

Beal, J. 2016. "Trading Accuracy for Speed in Approximate Consensus." *Knowledge Engineering Review* 31, no. 4: 325–342.

Castro, M., and B. Liskov. 1999. "Practical Byzantine Fault Tolerance." In *OSDI '99: Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Vol. 99, 173–186.

Chun, B.-G., P. Maniatis, S. Shenker, and J. Kubiatowicz. 2007. "Attested Append-Only Memory: Making Adversaries Stick to Their Word." *ACM SIGOPS Operating Systems Review* 41, no. 6: 189–204.

Correia, M., N. F. Neves, L. C. Lung, and P. Verssimo. 2005. "Low Complexity Byzantine-Resilient Consensus." *Distributed Computing* 17, no. 3: 237–249.

Debie, E., K. Kasmarik, and M. Garratt. 2023. "Swarm Robotics: A Survey From a Multi-Tasking Perspective." *ACM Computing Surveys* 56, no. 2: 1–38.

Distler, T. 2021. "Byzantine Fault-Tolerant State-Machine Replication From a Systems Perspective." *ACM Computing Surveys (CSUR)* 54, no. 1: 1–38.

Distler, T., C. Cachin, and R. Kapitza. 2015. "Resource-Efficient Byzantine Fault Tolerance." *IEEE Transactions on Computers* 65, no. 9: 2807–2819.

Distler, T., C. Cachin, and R. Kapitza. 2016. "Resource-Efficient Byzantine Fault Tolerance." *IEEE Transactions on Computers* 65, no. 9: 2807–2819.

Driscoll, K., B. Hall, H. Sivencrona, and P. Zumsteg. 2003. "Byzantine Fault Tolerance, From Theory to Reality." In *International Conference on Computer Safety, Reliability, and Security*, 235–248. Springer Berlin Heidelberg.

Gervais, A., G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. 2016. "On the Security and Performance of Proof of Work Blockchains." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Ccommunications Security*, 3–16.

Kapitza, R., J. Behl, C. Cachin, et al. 2012. "Cheapbft: Resource-Efficient Byzantine Fault Tolerance." In *Proceedings of the 7th ACM European Conference on Computer Systems*, 295–308.

Kotla, R., L. Alvisi, M. Dahlin, A. Clement, and E. Wong. 2010. "Zyzzyva: Speculative Byzantine Fault Tolerance." *ACM Transactions on Computer Systems (TOCS)* 27, no. 4: 1–39.

Krishnamohan, T. 2022. "Analysing the Suitability of Blockchain Consensus Algorithms in Swarm Robotics." *International Journal of Blockchains and Cryptocurrencies* 3, no. 4: 319–328.

Levin, D., J. R. Douceur, J. R. Lorch, and T. Moscibroda. 2009. " Trinc: Small Trusted Hardware for Large Distributed Systems." In *NSDI*, Vol. 9, 1–14.

Li, H., Y. Chen, X. Shi, 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*, 1–17.

Liu, J., W. Li, G. O. Karame, and N. Asokan. 2018. "Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing." *IEEE Transactions on Computers* 68, no. 1: 139–151.

Martin, J. P., and L. Alvisi. 2006. "Fast Byzantine Consensus." *IEEE Transactions on Dependable and Secure Computing* 3, no. 3: 202–215.

Miller, A., Y. Xia, K. Croman, E. Shi, and D. Song. 2016. "The Honey Badger of BFT Protocols." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 31–42. ACM.

Nguyen, T. T., A. Hatua, and A. H. Sung. 2020. "Blockchain Approach to Solve Collective Decision Making Problems for Swarm Robotics." In *Blockchain and Applications: International Congress*, 118–125. Springer.

pi3g. 2024. "Let's Trust TPM." https://pi3g.com/products/industrial/letstrust-tpm/.

Singh, P. K., R. Singh, S. K. Nandi, K. Z. Ghafoor, D. B. Rawat, and S. Nandi. 2020. "An Efficient Blockchain-Based Approach for Cooperative Decision Making in Swarm Robotics." *Internet Technology Letters* 3, no. 1: e140.

Song, Y., S. Lim, H. Myung, et al. 2023. "Distributed Swarm System With Hybrid-Flocking Control for Small Fixed-Wing UAVs: Algorithms and Flight Experiments." *Expert Systems With Applications* 229: 120457.

Strobel, V., E. CastellóFerrer, and M. Dorigo. 2020. "Blockchain Technology Secures Robot Swarms: A Comparison of Consensus Protocols and Their Resilience to Byzantine Robots." *Frontiers in Robotics and AI* 7: 54.

Strobel, V., A. Pacheco, and M. Dorigo. 2023. "Robot Swarms Neutralize Harmful Byzantine Robots Using a Blockchain-Based Token Economy." *Science Robotics* 8, no. 79: eabm4636.

Thakur, A., S. Sahoo, A. Mukherjee, and R. Halder. 2023. "Making Robotic Swarms Trustful: A Blockchain-Based Perspective." *Journal of Computing and Information Science in Engineering* 23, no. 6: 060803.

Veronese, G. S., M. Correia, A. N. Bessani, and L. C. Lung. 2010. "Ebawa: Efficient byzantine Agreement for Wide-Area Networks." In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, 10–19. IEEE.

Veronese, G. S., M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. 2011. "Efficient Byzantine Fault-Tolerance." *IEEE Transactions on Computers* 62, no. 1: 16–30.

Wang, R., C. Xu, S. Zhang, et al. 2024. "Matswarm: Trusted Swarm Transfer Learning Driven Materials Computation for Secure Big Data Sharing." *Nature Communications* 15, no. 1: 9290.

Xu, J., C. Wang, and X. Jia. 2023. "A Survey of Blockchain Consensus Protocols." *ACM Computing Surveys* 55, no. 13s: 1–35.

Yin, M., D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. 2019. "Hotstuff: BFT Consensus With Linearity and Responsiveness." In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 347–356.

Yuan, L., and H. Ishii. 2022. "Asynchronous Approximate Byzantine Consensus via Multi-Hop Communication." In *2022 American Control Conference (ACC)*, 755–760. IEEE.

Yuan, L., and H. Ishii. 2024. " Asynchronous Approximate Byzantine Consensus: A Multi-Hop Relay Method and Tight Graph Conditions." arXiv preprint arXiv:2403.07640.