Falcon: Advancing Asynchronous BFT Consensus for Lower Latency and Enhanced Throughput

Xiaohai Dai
HUST†
xhdai@hust.edu.cn

Chaozheng Ding
HUST†
chaozhengding@hust.edu.cn

Wei Li§
USYD‡
weiwilson.li@sydney.edu.au

Jiang Xiao
HUST†
jiangxiao@hust.edu.cn

Bolin Zhang
CMU*
bolinz@andrew.cmu.edu

Chen Yu
HUST†
yuchen@hust.edu.cn

Albert Y. Zomaya
USYD‡
albert.zomaya@sydney.edu.au

Hai Jin
HUST†
hjin@hust.edu.cn

ABSTRACT

Asynchronous *Byzantine Fault Tolerant* (BFT) consensus protocols have garnered significant attention with the rise of blockchain technology. A typical asynchronous protocol is designed by executing sequential instances of the *Asynchronous Common Sub-seQuence* (ACSQ). The ACSQ protocol consists of two primary components: the *Asynchronous Common Subset* (ACS) protocol and a block sorting mechanism, with the ACS protocol comprising two stages: broadcast and agreement. However, current protocols encounter three critical issues: high latency arising from the execution of the agreement stage, latency instability due to the integral-sorting mechanism, and reduced throughput caused by block discarding.

To address these issues, we propose Falcon, an asynchronous BFT protocol that achieves low latency and enhanced throughput. Falcon introduces a novel broadcast protocol, *Graded Broadcast* (GBC), which enables a block to be included in the ACS set directly, bypassing the agreement stage and thereby reducing latency. To ensure safety, Falcon incorporates a new binary agreement protocol called *Asymmetrical Asynchronous Binary Agreement* (AABA), designed to complement GBC. Additionally, Falcon employs a partial-sorting mechanism, allowing continuous rather than simultaneous block committing, enhancing latency stability. Finally, we incorporate an agreement trigger that, before its activation, enables nodes to wait for more blocks to be delivered and committed, thereby boosting throughput. We conduct a series of experiments to evaluate Falcon, demonstrating its superior performance.

PVLDB Reference Format:
Xiaohai Dai, Chaozheng Ding, Wei Li, Jiang Xiao, Bolin Zhang, Chen Yu, Albert Y. Zomaya, Hai Jin. Falcon: Advancing Asynchronous BFT Consensus for Lower Latency and Enhanced Throughput. PVLDB, 18(7): 2136 - 2148, 2025.
doi:10.14778/3734839.3734850

§Corresponding author.
†National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology.
‡School of Computer Science, The University of Sydney.
*Language Technologies Institute, Carnegie Mellon University.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734850

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CGCL-codes/falcon.

1 INTRODUCTION

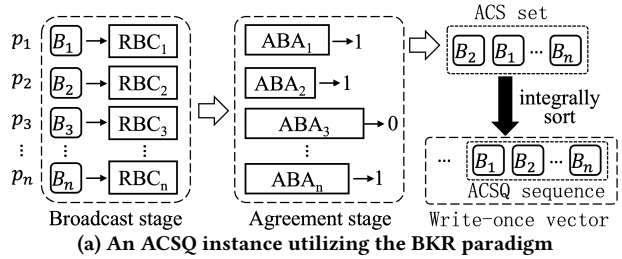
With the rising prominence of blockchain technology [5, 37, 42, 46], *Byzantine Fault Tolerant* (BFT) consensus has garnered significant attention from both academia and industry [28, 41, 43]. The BFT consensus protocols ensure agreement on data and facilitate implementing the *State Machine Replication* (SMR) among distributed nodes, some of which may be malicious, known as Byzantine nodes. These protocols are categorized by timing assumptions into synchronous protocols [3, 14], partially synchronous [13, 44], and asynchronous [4, 11] types. Given that synchronous and partially-synchronous protocols are susceptible to network attacks [27, 35], recent research has increasingly focused on developing asynchronous protocols [21, 22, 45].

1.1 Asynchronous BFT

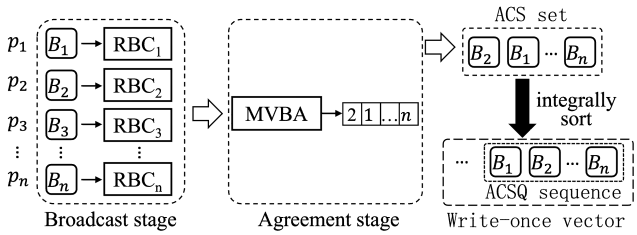
An asynchronous BFT protocol can be constructed by executing consecutive instances of *Asynchronous Common Sub-seQuence* (ACSQ), each generating a block sequence that is written to an append-only vector for execution. An ACSQ protocol comprises two components: an *Asynchronous Common Subset* (ACS) protocol and a block sorting mechanism. The ACS protocol produces a consistent set of blocks, referred to as the ACS set, which are then sorted into a sequence by the block sorting mechanism. The sorting of blocks is essentially the process of writing these blocks to the append-only vector, also known as committing blocks. Therefore, when there is no ambiguity, we use the terms "sort" and "commit" interchangeably. A more detailed explanation of the terminology for block operations is provided in Table 1 and Section 2.3.2.

The construction of ACS typically unfolds in two stages: the broadcast stage and the agreement stage. During the broadcast stage, each node disseminates blocks using the *Reliable Broadcast* (RBC) protocol [10, 12, 20]. The agreement stage diverges into two distinct paradigms: the BKR paradigm [8] and the CKPS paradigm [11].

An ACSQ protocol utilizing the BKR paradigm is illustrated in Figure 1a, where an *Asynchronous Binary Agreement* (ABA) instance [6, 36] is executed for each block. ABA is used to reach agreements on binary values. Each node inputs either 0 or 1 into

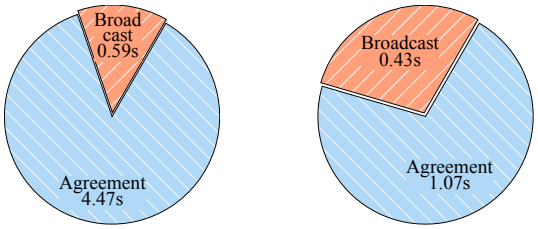


(a) An ACSQ instance utilizing the BKR paradigm



(b) An ACSQ instance utilizing the CKPS paradigm

Figure 1: The structure of ACSQ protocols



(a) BKR-ACS (b) CKPS-ACS

Figure 2: Latency decomposition

ABA, which then outputs 0 or 1 consistently. In the BKR paradigm, an output of 1 indicates that the corresponding block should be included in the ACS set, while an output of 0 signals rejection. As shown in Figure 1a, blocks B_2, B_1, ..., and B_n are included in the ACS set, whereas B_3 is excluded. On the other hand, the CKPS paradigm, depicted in Figure 1b, employs a single *Multi-valued Validated Byzantine Agreement* (MVBA) instance [4, 11] for the agreement stage. MVBA takes arrays of block numbers (i.e., identifiers of their creators) as inputs[1] and outputs a consistent array that selects blocks for inclusion in the ACS set. Following the agreement stage, blocks within the ACS set are sorted by block numbers.

However, existing protocols encounter three primary issues:

1.1.1 Issue 1: high latency. As analyzed in [27], the latency of BKR-ACS is defined by the sum of two stages: $t_r + \log(n) \cdot t_a$, where t_r represents the latency for a RBC instance, t_a for an ABA instance, and n denotes the node count. BKR-ACS's latency can be high for large n. As for CKPS-ACS, it has a latency of $t_r + t_m$, where t_m denotes the latency of an MVBA instance. While this protocol eliminates the logarithmic term, t_m introduces a significant term. Specifically, even the state-of-the-art MVBA protocol, sMVBA [26], has a best-case latency of six rounds, compared to just three rounds for t_r. We conduct preliminary experiments with 16 nodes to analyze the latency at different stages of the ACS protocol, with HBBFT [35] and Dumbo [27] representing BKR-ACS and CKPS-ACS. Experimental setups are provided in Section 6.1, and results are presented

[1]More accurately, the input is an array of block numbers plus proofs [27].

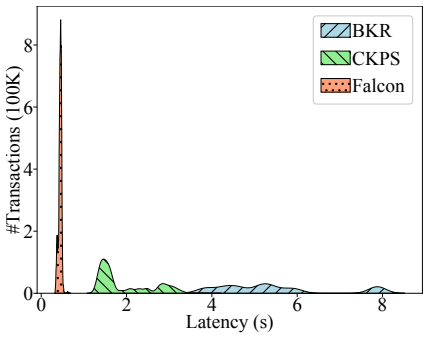


Figure 3: Latency distribution

in Figure 2. As shown, in both paradigms, **the agreement stage contributes significantly to the overall latency**.

1.1.2 Issue 2: latency instability. As illustrated in Figure 1, blocks in the ACS set are committed as a whole only after all blocks have been decided from the agreement stage. It is crucial to distinguish between the terms "decide" and "commit": deciding blocks refers to determining their inclusion or exclusion from the ACS set, while committing blocks involves sorting those. In the BKR protocol, blocks in the ACS set cannot be committed until the last ABA outputs. In the CKPS protocol, blocks are simultaneously included or excluded after MVBA outputs, and all blocks in the ACS set are then sorted. We refer to the mechanism of sorting only after all blocks have been decided as "integral sorting." **This integral-sorting mechanism introduces significant latency instability**, as evidenced by the experimental results depicting the latency distribution in Figure 3. Specifically, BKR's latency ranges from 3.1s to 8.3s, while CKPS's latency varies between 1.3s and 3.9s. Such latency instability can degrade user experience in higher-layer applications.

1.1.3 Issue 3: reduced throughput. In existing ACSQ constructions, an ACSQ instance outputs only a portion of all the broadcast blocks, typically $n-f$ (where f denotes fault tolerance), even when all nodes are correct and all blocks are well-broadcast. Specifically, in the BKR paradigm, once $n-f$ ABA instances output 1, nodes input 0 into the remaining f ABA instances, causing them to output 0 and resulting in the corresponding blocks being excluded from the ACS set and discarded. In the CKPS paradigm, each input array for the MVBA instance contains only $n-f$ block numbers, leading to an output array that also comprises $n-f$ block numbers. Consequently, a maximum of $n-f$ blocks can be committed. **This leads to reduced throughput, as up to f blocks may be discarded**.

1.2 Our solution & evaluation

To address the above issues, we propose Falcon, which delivers both low and stable latency as well as enhanced throughput. At a high level, Falcon utilizes ABA instances for the agreement stage, and is designed based on our three insights:

- **Insight 1:** If a node p_i delivering a block B_k can lead the corresponding ABA instance to eventually output 1, then p_i can include B_k in its ACS set as soon as B_k is delivered.
- **Insight 2:** For a specific block B_k, if a node p_i has decided on all blocks with smaller numbers, p_i can commit B_k immediately when it is included in the ACS set.

- **Insight 3:** A longer wait during the broadcast stage helps prevent the premature abandonment of blocks, thereby contributing to the committing of more blocks.

1.2.1 Solution to issue 1. Building on **insight 1**, we propose a new broadcast protocol, *Graded Broadcast* (GBC), for the broadcast stage, along with a novel binary agreement protocol, *Asymmetrical ABA* (AABA), for the agreement stage. With GBC, a node can deliver blocks with two grades, 1 and 2. The intuition behind introducing grades is that we want a correct node to deliver a block with grade 2 only after $f + 1$ correct nodes have already delivered it with a lower grade (i.e., grade 1). Nodes delivering with grade 1 will input 1 into AABA. In this case, AABA's biased-validity property ensures that it will output 1, thus satisfying the condition in **insight 1**.

In Falcon, once a node delivers a block with grade 2, it can directly include that block into its ACS set. For nodes that deliver the block with grade 1, they will input 1 to AABA. If a node outputs 1 from AABA, it will also include the corresponding block in the ACS set. Besides, we incorporate a shortcut mechanism to AABA, allowing it to output quickly if all nodes input 0.

In a favorable situation, each node can deliver all blocks with grade 2 during the broadcast stage and include them directly in its ACS set, bypassing the time-consuming agreement stage. This significantly reduces latency and effectively addresses **issue 1**.

1.2.2 Solution to issue 2. Based on **insight 2**, we devise a *partial-sorting mechanism* in Falcon, enabling a block to be committed directly if it is included in the ACS set and all preceding blocks are decided. For example, consider two blocks, B_1 and B_2, created by nodes p_1 and p_2. If B_1 has been decided, B_2 can be committed immediately once it is included in the ACS set. This eliminates the need to wait for decisions on all blocks, ensuring more stable latency and solving **issue 2**. The preliminary experimental results presented in Figure 3 indicate that Falcon's transaction latency fluctuates within a narrow range of [0.3s, 0.7s]. Besides, this mechanism allows blocks to be committed earlier, further reducing overall latency.

1.2.3 Solution to issue 3. Following **insight 3**, we introduce an *agreement trigger* that allows nodes to wait for more blocks to be delivered. Specifically, once a node has delivered $n - f$ blocks with grade 2 from the GBC instances, it checks whether the trigger has been activated. If so, the node proceeds directly to the agreement stage; otherwise, it waits until either all blocks are delivered with grade 2 or the trigger is activated. If all blocks are delivered with grade 2 before the trigger activation, they will be included in the ACS set, ensuring no blocks are discarded and resolving **issue 3**.

The trigger is activated by system events, particularly the grade-2 delivery of a block from the next ACSQ instance. In favorable situations, blocks from the current ACSQ will always be delivered before those from the next ACSQ, keeping the trigger inactive. Conversely, if the blocks from the next ACSQ are delivered first, it indicates an unfavorable situation, prompting the system to transition to the agreement stage based on the trigger activation.

1.2.4 Evaluation. To evaluate Falcon's performance, we implement a prototype system and conduct a series of comparisons with four representative protocols: HBBFT [35], Dumbo [27], MyTumbler [31], and Narwhal&Tusk [19]. We examine both favorable situations, where all nodes are correct, and unfavorable ones, which include some faulty nodes. The results show that Falcon achieves lower latency in both situations. In favorable situations, this advantage arises from its use of GBC for direct block decisions, bypassing the agreement stage. In unfavorable situations, its low latency is attributed to AABA's shortcut mechanism, enabling faster outputs. Falcon also surpasses others in throughput, as it facilitates the delivery and committing of more blocks before the trigger is activated. Additionally, Falcon achieves more stable latency due to the partial-sorting mechanism that enables continuous block committing.

2 MODEL AND PRELIMINARIES

2.1 System model

We consider a system consisting of n nodes, of which up to f ($3f + 1 \leq n$) can exhibit Byzantine behavior. These Byzantine nodes are controlled by an adversary capable of coordinating their actions. The other nodes, termed correct, strictly adhere to the protocol. Each node is uniquely identified by a distinct number, denoted as p_i ($1 \leq i \leq n$). The system operates over an asynchronous network, where no assumptions are made about network delays. Furthermore, the adversary is assumed to have the ability to delay and reorder message delivery, though all messages are eventually delivered [35]. Each pair of nodes is connected via an authenticated network link.

A *Public Key Infrastructure* (PKI) and a threshold signature scheme are established within the system. All messages are signed and verified through PKI. For conciseness, we omit the description of the PKI signature process in the rest of the paper. The threshold signature scheme is employed to generate proofs in the GBC. The adversary is assumed to be computationally bounded, implying it cannot compromise PKI or the threshold signature.

2.2 SMR and BFT protocols

We focus on a *State Machine Replication* (SMR), where each node p_i maintains a vector denoted as C_i. Each slot within C_i is indexed by r. Initially, C_i is empty, with $C_i[r] = \perp$ for each r (where $r \geq 1$). Without loss of generality, we assume elements to be written in C_i are blocks, and each block comprises multiple transactions. Blocks, particularly their contained transactions, in the append-only vector, are eligible for execution to change the machine state.

A block is considered committed by p_i when it is written to C_i. Besides, a block ready for committing is written to the first available empty slot. In other words, a block B is written at $C_i[r]$ only if, for each index l prior r, $C_i[l] \neq \perp$. We define the length of a chain C_i as the maximum index of its non-empty slots, denoted by $|C_i|$.

Additionally, each node p_i maintains a local buffer, denoted as buf_i. Transactions are initially cached in this buffer. When proposing a block, a node extracts a group of transactions, typically those at the prefix of the buffer. Transactions that are contained in a committed block are then removed from the buffer. Specifically, a correct BFT protocol has to satisfy the following two properties:

- **Safety:** For two nodes p_i and p_j, if $C_i[r] \neq \perp$ and $C_j[r] \neq \perp$, then $C_i[r] = C_j[r]$.
- **Liveness:** If a transaction tx is included in every correct node's buffer, each one will eventually commit tx.

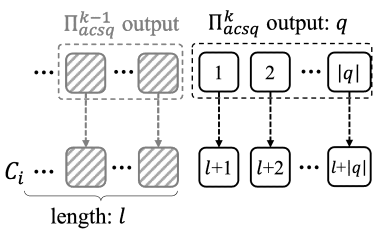


Figure 4: BFT construction based on ACSQ

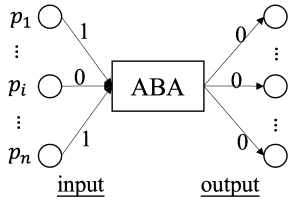


Figure 5: Schematic diagram of ABA

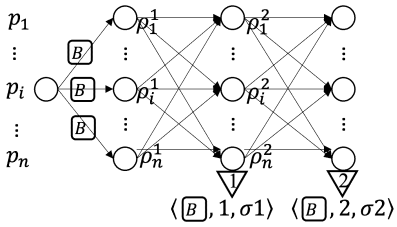


Figure 6: Construction of GBC

Table 1: Terminology for block operations. I, II, and III denote the broadcast, agreement, and sorting stages, respectively.

Operation	Explanation	Stage
Deliver	Accept a block	I
Decide	Include/exclude a block in/from ACS set	I or II
Sort/Commit	Sort blocks in the ACS set and write sorted blocks into C_i	III

2.3 Asynchronous common sub-sequence

2.3.1 Definition of ACSQ. Before introducing *Asynchronous Common Sub-seQuence* (ACSQ) protocol, we first introduce a basic concept named *Asynchronous Common Subset* (ACS) [7, 8, 27, 35]. Generally speaking, the ACS protocol is a single-shot protocol, where each node inputs a block and then outputs an identical set of blocks. The ACS protocol must satisfy the following three properties:

- **Agreement:** If a correct node outputs a set s, then every correct node outputs s.
- **Validity:** If a correct node outputs a set s, then $|s| \geq n - f$.
- **Totality:** If each correct node receives an input, all correct nodes will eventually output.

The ACSQ protocol can be considered a sorted version of the ACS protocol, in that the output from ACSQ is a sequence of blocks after sorting. Concretely speaking, the ACSQ protocol is defined by the following properties:

- **Agreement:** If a correct node outputs a sequence q, then every correct node outputs q.
- **Validity:** If a correct node outputs a sequence q, then $|q| \geq n$-f.
- **Totality:** If each correct node receives an input, all correct nodes will eventually output.

The validity property of ACSQ mandates the inclusion of at least $n - f$ blocks. However, as discussed in **issue 3** of Section 1.1, this permits the discarding of up to f blocks, even under favorable situations. To address this, we introduce a new property termed 'optimistic validity' for ACSQ, defined as follows:

- **Optimistic validity:** If all nodes are correct and the network conditions are favorable, then each correct node can output a sequence comprising inputs from all nodes, such that $|q| = n$.

A favorable network refers to a network where, if a correct node first broadcasts block $B1$ and then broadcasts $B2$ after delivering $B1$, all correct nodes are guaranteed to deliver $B2$ after $B1$.

2.3.2 ACSQ construction & terminology. As described in Section 1.1, ACSQ can be implemented by adding a sorting stage to ACS, which itself is comprised of the broadcast and agreement stages. To clarify the terms related to block operations at each stage, we provide explanations in Table 1. During the broadcast stage, the operation of a node accepting a block is referred to as "deliver". Through the agreement stage, a decision is made about a block's inclusion or exclusion in/from the ACS set, which is termed "decide". Since Falcon can bypass the agreement stage, the "decide" operation may also be completed at the end of the broadcast stage. Blocks in the ACS set are sorted in the sorting stage, which is essentially the process of writing blocks to C_i, also referred to as "commit". Thus, we interchangeably use the terms "sort" and "commit" in this paper.

2.3.3 Constructing BFT from ACSQ. The BFT protocol can be effectively constructed by operating consecutive ACSQ instances, each denoted as Π^k_{acsq}. As shown in Figure 4, a node p_i writes the sequence q outputted from Π^k_{acsq} to C_i only after all outputs from previous ACSQ instances Π^m_{acsq} (where $m < k$) have been written. Blocks in q are written to C_i in their original order in q, specified as $C_i\,[l+1 : l+|q|] = q$, where l denotes C_i's length before writing q.

Using the induction method, it is straightforward to demonstrate that the BFT construction from ACSQ upholds the safety property as outlined in Section 2.2. Concerning the liveness property, each correct node will include the transaction tx in its input block for the next Π_{acsq} instance as long as tx remains uncommitted. Consider that in instance Π^k_{acsq} where tx is still uncommitted, every correct node will include tx in its block input. Given the validity and totality properties, the output q from Π^k_{acsq} will contain at least $n-f$ blocks, with at least $n-2f$ of these blocks inputted by correct nodes. Thus, q will definitely include tx, ensuring tx to be committed. In other words, the BFT construction from ACSQ is shown to guarantee the liveness property as defined in Section 2.2.

2.4 Asynchronous binary agreement

Asynchronous Binary Agreement (ABA) is recognized as one of the simplest Byzantine agreement protocols. Within an ABA instance, each node inputs a binary value and expects a consistent binary value as output. As exemplified in Figure 5, p_1 and p_n input 1 to ABA, while p_i inputs 0. Finally, all nodes output the same bit 0. Specifically, an ABA protocol is defined by the following properties:

- **Agreement:** If a correct node outputs b, then every correct node outputs b.
- **Validity:** If a correct node outputs b, then at least one correct node inputs b.
- **Termination:** If each correct node receives an input, all correct nodes will eventually output.

Over the past decades, various constructions of ABA protocols have been developed [2, 6, 23, 36]. We utilize the ABA protocol in a black-box manner.

3 BUILDING BLOCKS

3.1 Graded broadcast (GBC)

3.1.1 Intuition behind GBC. As outlined in Section 1.2, Falcon allows nodes to include delivered blocks to the ACS set at the end of the broadcast stage, thus reducing latency. However, due to network asynchrony, some nodes may not deliver the block by then and must rely on the binary agreement for a decision. To ensure that the binary agreement decides to include the block to the ACS set (i.e., outputs 1), a mechanism is required. This mechanism guarantees that if a correct node delivers the block, a sufficient number of correct nodes must have delivered it *in some weaker form*. This directs them to input 1 into the binary agreement. The GBC broadcast protocol is used to implement this mechanism. It guarantees that if a correct node delivers a block with grade 2, then at least $f+1$ correct nodes have delivered the block with a lower grade of 1.

3.1.2 Definition of GBC. Similar to other broadcast protocols, a node in GBC acts as the broadcaster to disseminate a block, while the others act to deliver the block. The block can be delivered with two grades, namely 1 and 2. Besides a block, a node will also deliver proof that certifies the receipt/delivery situation among the nodes. Therefore, a node will deliver a block in the format of $\langle B, g, \sigma \rangle$, where B signifies the block data, g ($g \in \{1, 2\}$) denotes the grade, and σ represents the proof. In the context of GBC, we differentiate the terms 'receive' and 'deliver.' A node is said to receive a block B once it obtains B from the broadcaster. By contrast, a node is said to deliver B if some accompanying proof σ is also generated. The GBC protocol has to satisfy the following properties:

- **Consistency:** If two correct nodes p_i and p_j deliver $\langle B_i, g_i, \sigma_i \rangle$ and $\langle B_j, g_j, \sigma_j \rangle$ respectively, then $B_i = B_j$.
- **Delivery-correlation:** If a correct node delivers $\langle B, 2, \sigma \rangle$, at least $f+1$ correct nodes have delivered B with grade 1.
- **Receipt-correlation:** If a correct node delivers $\langle B, 1, \sigma \rangle$, then at least $f+1$ correct nodes have received B.

Note that while GBC shares some similarities with Abraham et al.'s Gradecast [1] and Malkhi et al.'s BBCA [34] protocols, there are also key differences. First, Gradecast is designed for synchronous networks and defines a 'validity' property, whereas GBC makes no assumptions about network synchrony and does not define such a property. Additionally, Gradecast's 'agreement' property requires all correct nodes to output, while GBC's 'delivery-correlation' property only requires $f+1$ correct nodes to output. Second, BBCA introduces a probe mechanism that returns a NOADOPT result, used by BBCA-Chain for new block generation. GBC does not define a similar probe mechanism or NOADOPT result. In addition to defining the 'delivery-correlation' property, which is similar to the 'complete-adopt' property in BBCA, GBC also introduces the 'receipt-correlation' property, which plays a crucial role in the proof of Lemma 1 in Section 5.1.2.

3.1.3 Construction of GBC. The construction of GBC is inspired by the normal-case protocol of PBFT [13]. As illustrated in Figure 6, after receiving the block B, each node p_j will broadcast a partial threshold signature ρ_j^1 on the concatenation of B's digest and a tag 1. After receiving $n-f$ messages of ρ_j^1, a node can construct a complete threshold signature $\sigma 1$ and deliver $\langle B, 1, \sigma 1 \rangle$. Besides, it will further broadcast a partial threshold signature ρ_j^2 on the concatenation of B's digest and a tag 2. Similarly, a node can construct a complete threshold signature $\sigma 2$ based on $n-f$ received ρ_j^2 and deliver $\langle B, 2, \sigma 2 \rangle$. Note that, since the GBC proof is used to prove endorsements from nodes, it can alternatively be constructed by packaging $n-f$ PKI signatures, which is more efficient than using threshold signatures. In this case, the proof verification process involves validating all PKI signatures included in the GBC proof.

The above construction achieves all GBC properties, even in the presence of a Byzantine broadcaster. Specifically, consider the Byzantine broadcaster sending two contradictory blocks, B and B'. Since the delivery requires the collection of signatures from $n-f$ nodes for the same block, and given that $n \geq 3f+1$, at most one of B and B' can be delivered, thus ensuring consistency. On the other hand, a Byzantine node can remain silent, meaning it refuses to broadcast any block. However, this only reduces the consensus throughput without compromising consistency or safety.

3.2 Asymmetrical ABA (AABA)

3.2.1 Intuition behind AABA. Our intuition behind AABA mainly unfolds two aspects. First, as briefly described in Section 3.1, GBC guarantees that if a correct node delivers a block with grade 2, then at least $f+1$ correct nodes have delivered it with grade 1. We need to design an ABA variant that ensures, under these conditions, it will decide to include the block in the ACS set. Second, we hope to introduce a shortcut outputting mechanism for the situation where all nodes input 0. This mechanism can accelerate the agreement when some nodes crash before the start of the broadcast stage.

3.2.2 Definition of AABA. Different from ABA, AABA accepts asymmetrical inputs. Specifically, it accepts 0 directly while accepting 1 only if an externally defined value v and the certified proof σ are provided in the format of a triplet $\langle 1, v, \sigma \rangle$. Also, an external validation predicate Q is defined regarding v and σ. In the context of Falcon, v is the digest of a block, σ is the proof $\sigma 1$ when delivering the block with grade 1, while Q is a signature verification function. Similar to ABA, AABA also produces the output of a bit.

AABA defines asymmetrical properties on bits of 0 and 1. It requires AABA to output 1 if at least $f+1$ correct nodes receive the input $\langle 1, v, \sigma \rangle$. Furthermore, if AABA outputs 1, at least one node, whether Byzantine or correct, must have received $\langle 1, v, \sigma \rangle$ such that $Q(v, \sigma)$=true. Besides, AABA enables a shortcut to commit 0: if all nodes receive the input 0, AABA can output 0 quickly.

To be more specific, AABA is defined by the following properties:

- **Agreement:** If a correct node outputs b, then every correct node outputs b.
- 1-**validity:** If a correct node outputs 1, then at least one node, whether Byzantine or correct, must have received the input $\langle 1, v, \sigma \rangle$ s.t. $Q(v, \sigma)$=true.
- **Biased-validity:** If $f+1$ or more correct nodes receive the valid input $\langle 1, v, \sigma \rangle$, each correct node will output 1.
- **Termination:** If each correct node receives an input, all correct nodes will eventually output.
- **Shortcut** 0**-output:** If all nodes receive the input 0, each correct node can output after three communication rounds.

3.2.3 Construction of AABA. We propose a construction of AABA, named Π_{aaba}, which leverages any existing ABA protocol in a

Algorithm 1: Construction of AABA: Π_{aaba} (for p_i)

1 **Let** I_i denote the input received by p_i; $S \leftarrow \emptyset$; $cnt_0 \leftarrow 0$;
// amplification phase
2 **broadcast** $\langle \texttt{amp}, I_i \rangle$;
3 **on** *receiving* $\langle \texttt{amp}, I_j \rangle$ *from* p_j**:**
4 | **if** I_j *is decoded as* $\langle 1, v, \sigma \rangle$ ***and*** $Q(v, \sigma)$=`true` **then**
5 | | **if** p_i *has not broadcast a* `sho1` *message* **then**
6 | | | **broadcast** $\langle \texttt{sho1}, 1 \rangle$;
7 | **else if** I_j *is decoded as* 0 **then**
8 | | $cnt_0 \leftarrow cnt_0 + 1$;
9 | | **if** cnt_0=n-f ***and*** p_i *has not broadcast* `sho1` *msg.* **then**
10 | | | **broadcast** $\langle \texttt{sho1}, 0 \rangle$;
// shortcut phase
11 **on** *receiving* $\langle \texttt{sho1}, b \rangle$ *from* $f + 1$ *nodes*:
12 | **if** p_i *has not broadcast* $\langle \texttt{sho1}, b \rangle$ **then**
13 | | **broadcast** $\langle \texttt{sho1}, b \rangle$;
14 **on** *receiving* $\langle \texttt{sho1}, b \rangle$ *from* $n - f$ *nodes*:
15 | $S \leftarrow S \bigcup \{b\}$;
16 | **if** p_i *has not broadcast a* `sho2` *message* **then**
17 | | **broadcast** $\langle \texttt{sho2}, b \rangle$;
18 **on** *receiving* $n - f$ `sho2` *messages whose bits are in* S**:**
19 | **if** *all these messages contain* 0 **then**
20 | | **output** 0;
21 | **if** *at least one message contains* 0 **then**
22 | | **input** 0 to the subsequent ABA protocol
23 | **else**
24 | | **input** 1 to the subsequent ABA protocol
// output from ABA
25 **on** *outputting b from ABA*:
26 | **output** b if not yet

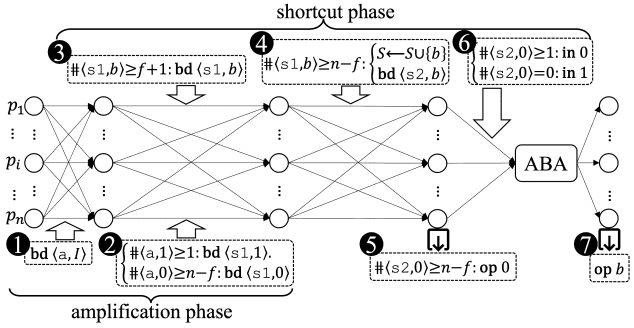


Figure 7: Schematic diagram of Π_{aaba}, where 'bd,' 'in,' and 'op' denote the actions of 'broadcast,' 'input,' and 'output.' Besides, 'a,' 's1,' and 's2' represent the tags of 'amp,' 'sho1,' and 'sho2.'

black-box manner. Specifically, Π_{aaba} introduces a pre-processing component before running an existing ABA protocol.

The pre-processing component of Π_{aaba} is designed in two phases, with the first amplifying the input of 1, named the amplification phase, and the second facilitating shortcut output of 0, named the shortcut phase. The pseudocode and schematic diagram of Π_{aaba} are presented as Algorithm 1 and Figure 7, respectively. In the amplification phase, each node broadcasts its input in the

Algorithm 2: Early-stopping mechanism for Π_{aaba} (for p_i)

// after outputting 0 in Line 20 of Algorithm 1
1 **broadcast** $\langle \texttt{stop}, 0 \rangle$;
2 **on** *receiving* $f + 1$ `stop` *messages containing* 0**:**
3 | **if** p_i *has not broadcast a* `stop` *message yet* **then**
4 | | **broadcast** $\langle \texttt{stop}, 0 \rangle$;
5 | **if** p_i *has not outputted yet* **then**
6 | | **output** 0;
7 **on** *receiving* $n - f$ `stop` *messages containing* 0**:**
8 | **exit** from current Π_{aaba} instance

amp message (Line 2 of Algorithm 1 and ❶ in Figure 7) and waits to receive at least $n - f$ amp messages. If a node receives at least one amp message with the valid bit of 1, it will broadcast the sho1 message of 1 for the shortcut phase. Conversely, when it receives $n - f$ amp messages with 0, it will broadcast the sho1 message of 0 (Lines 3-10 and ❷ in Figure 7). Through the amplification phase, as long as $f + 1$ correct nodes have the initial input of 1, all correct nodes will input 1 to the subsequent shortcut phase, thereby amplifying the input of 1.

The shortcut phase consists of two steps, with the first step mirroring a filter functionality designed by MMR-ABA [36]. Within the first step, if a node receives b from at least $f + 1$ nodes and it has not yet broadcast a sho1 message containing b, it will also broadcast a $\langle \texttt{sho1}, b \rangle$ (Lines 11-13 and ❸ in Figure 7), even though it has broadcast $1-b$ before. In the meanwhile, once a node receives $\langle \texttt{sho1}, b \rangle$ from at least $n - f$ nodes, it will add b to a local set S. If it has not broadcast a sho2 message, it will broadcast one containing b (Lines 14-17 and ❹ in Figure 7). Intuitively, this step (Lines 11-17) guarantees that a correct node will not include b in its set S if b is input to the shortcut phase only by Byzantine nodes.

In the second step, the node will wait to receive $n - f$ sho2 messages, each containing a bit in the set S. If all these sho2 messages contain 0, the node will output 0 directly (Lines 19-20 and ❺ in Figure 7). If at least one sho2 message contains 0, the node will input 0 to the subsequent ABA protocol; otherwise, it will input 1 (Lines 21-24 and ❻ in Figure 7). It will then take the output from ABA as its output from Π_{aaba} (Lines 25-26 and ❼ in Figure 7).

If all nodes input 0 to Π_{aaba}, each correct node can output 0 through the shortcut mechanism at Line 20, resulting in a latency of three communication rounds. Note that the shortcut mechanism cannot be exploited by the adversary to undermine the protocol's correctness. In simple terms, when a correct node outputs via the shortcut mechanism, it must have received $n - f$ sho2 messages containing 0 (Line 19). Since $n \geq 3f + 1$, each correct node will receive at least one sho2 message with 0 and then input 0 into the ABA protocol (Lines 21-22). Due to ABA's validity property, it will output 0, ensuring consistency with the shortcut output. Due to space limitations, a detailed correctness analysis of Π_{aaba} is given in Appendix A.1 of our full version [15].

Early-stopping mechanism. In Π_{aaba}, a correct node must continue to execute the subsequent ABA even if it has outputted 0 (Line 20). This is necessary because ABA promises to terminate only if each correct node receives an input. To address this, we introduce an early-stopping mechanism, described in Algorithm 2, which

Algorithm 3: Falcon protocol (for p_i)

Let $\Pi^k_{acsq}.v$ denote the variable v in Π^k_{acsq}; $k \leftarrow 1$;
while `true`:
  if Π^k_{acsq} *has not been activated* **then**
    activate Π^k_{acsq};
  on $\left|\Pi^k_{acsq}.M_2\right| = n - f$:
    activate Π^{k+1}_{acsq};
  on $\left|\Pi^{k+1}_{acsq}.M_2\right| \geq 1$:
    activate $\Pi^k_{acsq}.trigger$;
  wait until Π^k_{acsq} returns;
  $k \leftarrow k + 1$;

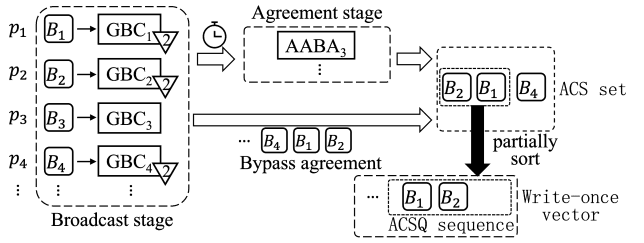


Figure 8: Π_{acsq} **with the partial sorting mechanism**

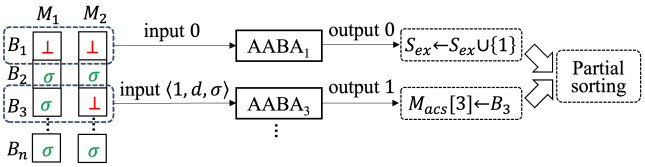


Figure 9: Schematic diagram of the agreement stage

enables a node to exit early without finishing the subsequent ABA if $f + 1$ correct nodes output 0. Specifically, a node will broadcast a `stop` message after outputting 0 in Line 20 of Algorithm 1. Once receiving $f + 1$ `stop` messages, a node will also broadcast a `stop` message with 0 (Lines 2-4 of Algorithm 2). Besides, it will output 0 if it has not done yet (Lines 5-6). Once receiving $n - f$ `stop` messages, a node can exit from the current Π_{aaba} instance. Particularly, it will stop its participation in the subsequent ABA protocol. Owing to space constraints, the correctness of the early-stopping mechanism is analyzed in Appendix A.2 of our full version [15].

4 FALCON DESIGN

4.1 Overall design

Like in other ACSQ-based protocols, Falcon also advances by executing successive Π_{acsq} instances, which is described in Algorithm 3. The ACSQ protocol comprises two parts: the ACS protocol and the block partial-sorting mechanism, as described in Algorithm 4. The ACS protocol consists of the broadcast and agreement stages. A schematic diagram of Π_{acsq} is illustrated in Figure 8.

4.1.1 Broadcast stage. Within this stage, each node broadcasts its block through the GBC protocol (Lines 3-8). If the node p_i delivers the block B_j with the grade 2, p_i can directly include B_j into its ACS, without running AABA for B_j, as shown by B_2, B_1, and B_4 in Figure 8. The node waits for all blocks in this ACSQ instance to be grade-2

Algorithm 4: Π_{acsq} with the instance identity k (for p_i)

Let B_i denote the block proposed by p_i;
$M_1 \leftarrow []$; $M_2 \leftarrow []$; $M_{acs} \leftarrow []$; $S_a \leftarrow \emptyset$; $idx \leftarrow 0$; $S_{ex} \leftarrow \emptyset$;
// `broadcast stage`
broadcast B_i through $\mathtt{GBC}_i$;
on *delivering* $\langle B_j, 1, \sigma 1\rangle$ *from* $\mathtt{GBC}_j$:
  $M_1[j] \leftarrow \langle B_j, 1, \sigma 1\rangle$;
on *delivering* $\langle B_j, 2, \sigma 2\rangle$ *from* $\mathtt{GBC}_j$:
  $M_2[j] \leftarrow \langle B_j, 2, \sigma 2\rangle$;
  $M_{acs}[j] \leftarrow B_j$; // `include it to the ACS set directly`
  $PartialSort(k, M_{acs}, S_{ex}, j)$;
wait until $|M_2| = n$ **or** the *trigger* activation;
if *the trigger is activated* **then**
  wait until $|M_2| \geq n - f$;
  stop sending partial signatures in the broadcast stage;
// `agreement stage`
foreach $j \in [1..n]$ *s.t.* $M_2[j] = \perp$ **do**
  $S_a \leftarrow S_a \cup \{j\}$;
  if $M_1[j] \neq \perp$ **then**
    $(B, g, \sigma) \leftarrow M_1[j]$;
    input $\langle 1, B.d, \sigma\rangle$ to $\mathtt{AABA}_j$; //$B.d$ `denotes B's digest`
  else
    input 0 to $\mathtt{AABA}_j$;
on $\mathtt{AABA}_j$ *outputs* b:
  if $b = 1$ **then**
    $M_{acs}[j] \leftarrow B_j$; // `include it to the ACS set`
  else
    $S_{ex} \leftarrow S_{ex} \bigcup \{j\}$;
  $PartialSort(k, M_{acs}, S_{ex}, j)$;
// `delivery-assistance mechanism`
on *receiving an* $\mathtt{AABA}_j$ *message from* p_t:
  if $M_2[j] \neq \perp$ **then**
    send $M_2[j]$ to p_t;
on *receiving* $\langle B_j, 2, \sigma 2\rangle$:
  $M_2[j] \leftarrow \langle B_j, 2, \sigma 2\rangle$;
  $M_{acs}[j] \leftarrow B_j$; // include it to the ACS set
  $PartialSort(k, M_{acs}, S_{ex}, j)$;
  stop participating in $\mathtt{AABA}_j$;
wait until $\mathtt{AABA}_j$ outputs **or** stops through *Line 34*, $\forall j \in S_a$;
return from current Π_{acsq} instance

delivered or until the agreement trigger is activated. If it is the latter, the node will continue to wait until at least $n - f$ blocks have been grade-2 delivered if not yet. After that, the node stops generating or sending the partial threshold signatures in the broadcast stage (Lines 10-13). If all blocks are grade-2 delivered before the trigger activation, node p_i can skip the agreement stage, thus reducing latency and addressing **issue 1** outlined in Section 1.1.

4.1.2 Agreement stage. p_i will execute an AABA instance for each block B_j that has not been grade-2 delivered, as exemplified in

Algorithm 5: Partially sorting protocol (for p_i)

```
1  doneACSQId ← 0;
2  define PartialSort(k, M, S, idx):
3      wait until doneACSQId = k − 1;
4      while idx < n and (M[idx + 1] ≠ ⊥ or idx + 1 ∈ S):
5          if M[idx + 1] ≠ ⊥ then
6              l ← |C_i|; C_i[l + 1] ← M[idx + 1];
7          idx ← idx + 1;
8      if idx = n then
9          doneACSQId = k
10     return idx;
```

Figure 9. Specifically, p_i constructs an input to AABA instance AABA_j based on its delivery situation of B_j. If it has delivered B_j with the grade 1 (e.g., B_3 in Figure 9), it will input $\langle 1, d, \sigma \rangle$ to AABA_j, where d denotes the digest of B_j. Otherwise, if B_j has not been delivered at all (e.g., B_1 in Figure 9), it will input 0 to AABA_j (Lines 14-20).

p_i will wait for each AABA to terminate, either outputting eventually or delivering the corresponding blocks through the delivery-assistance mechanism. The delivery-assistance mechanism is detailed in Section 4.3. If AABA_j outputs 1 (e.g., AABA_3 in Figure 9), the node will also include B_j to the ACS set (Lines 21-23). Otherwise, such as AABA_1, it will be marked in a set S_{ex} (Lines 24-25), which will be later accessed during the partial sorting process. Due to the conciseness, we omit a trivial query process. Specifically, p_i may need to query B_j since it may not have received B_j. This can be done by broadcasting a query request containing the digest of B_j. A correct node will respond with the data of B_j after receiving this request. We will prove in Lemma 1 of Section 5.1 that at least one correct node must have received B_j and can complete the response.

4.1.3 Partial sorting. A node in Falcon can sort blocks without waiting for all blocks to be decided, which is named a partial sorting mechanism. Anytime a new block is decided as included or excluded in the ACS set, the node can call the partial sorting function to commit blocks, as shown in Line 9, Line 26, and Line 33 of Algorithm 4. The partial sorting function is described in Algorithm 5, which takes four parameters: k represents the identity of the Π_{acsq} instance, M is the set of blocks that have been decided as included, S is the set marking all blocks being excluded, and idx denotes the last processed index.

An index can be processed if all smaller indices have been processed and a block with this index has been decided. Furthermore, if a block with this index is included in the ACS set, this block can be sorted immediately. A one-larger index $idx + 1$ will be processed after another idx until all indices have been processed or a block with $idx + 1$ has not been decided (Lines 4-7). The partial-sorting mechanism allows blocks to be committed continuously, eliminating the need to wait for the slowest block to be decided. This not only improves latency stability, addressing **issue 2** from Section 1.1, but also reduces the overall latency. As a practice common in the asynchronous BFT protocol and having been introduced in Section 2.3.3, blocks in an Π_{acsq} instance can be committed only after all Π_{acsq} instances with smaller identities have been processed, which is ensured by Line 3 and Lines 8-9 of Algorithm 5.

4.2 Agreement trigger

At a high level, the agreement trigger in an Π_{acsq} instance is designed based on an event in the subsequent Π_{acsq} instance. To be more specific, if a node grade-2 delivers $n - f$ blocks in an instance Π^k_{acsq}, it will activate the next instance Π^{k+1}_{acsq}, as shown by Lines 5-6 of Algorithm 3. If some block in Π^{k+1}_{acsq} has been grade-2 delivered, the trigger in Π^k_{acsq} is activated (Lines 7-8 of Algorithm 3).

If it is in a favorable situation, all blocks in Π^k_{acsq} can be grade-2 delivered before any block in Π^{k+1}_{acsq} is grade-2 delivered, and the trigger will not be activated at all. Additionally, under this favorable situation, all blocks can be committed, leading to higher throughput and resolving **issue 3** outlined in Section 1.1.

4.3 Delivery-assistance mechanism

As described in Section 4.1.2, a node will not activate the AABA_j instance for a block B_j if it has grade-2 delivered B_j. Other nodes that activate the AABA_j instance may not output from AABA_j, since the AABA protocol only promises termination if each correct node receives an input. To address this, we introduce a delivery-assistance mechanism, as described by Lines 27-34 of Algorithm 4. Specifically, if a correct node p_i receives an AABA_j message from p_t and p_i has grade-2 delivered B_j, p_i will send $\langle B_j, 2, \sigma 2 \rangle$ to p_t (Lines 27-29). After receiving $\langle B_j, 2, \sigma 2 \rangle$, p_t deals with B_j as if B_j is grade-2 delivered through the GBC_j instance. Besides, p_t will stop its participation in AABA_j (Lines 30-34).

4.4 Performance analysis

The ABA protocol absorbed in Π_{aaba} is implemented using the ABY-ABA [2], which has an expected worst latency of 9 communication rounds. Since it is hard to perform a quantitative analysis of the sorting process, our primary focus is on analyzing the latency of generating the ACS set, in terms of the communication rounds.

In a favorable situation where all nodes are correct, and the network condition is well, all blocks can be grade-2 delivered and included into the ACS set before the trigger activation, resulting in a latency of 3 communication rounds to generate the ACS set.

In a less favorable situation where nodes can only crash before the Π_{acsq} instance activates, some blocks fail to be grade-2 delivered before the trigger activation. However, all correct nodes will input 0 to these AABA instances, which takes 3 rounds to output through the shortcut outputting mechanism. Since the trigger is activated by grade-2 delivering a block in the next Π_{acsq} instance and the next Π_{acsq} instance is activated after grade-2 delivering $n - f$ blocks in the current Π_{acsq} instance, the time for the trigger activation equals the sum of two GBC instances, namely 6 rounds. Therefore, the latency to generate the ACS set in this situation is 9 rounds.

In the worst situation, f AABA instances are activated and cannot output through the outputting mechanism. It takes 12 rounds in expectation to output from AABA. Therefore, the latency to generate the ACS set in this situation is $6 + 12 \cdot log(n)$ rounds.

5 CORRECTNESS ANALYSIS

5.1 Analysis on Π_{acsq}

We prove that Π_{acsq} correctly implements an ACSQ protocol, beginning with an intuitive analysis and followed by a rigorous one.

5.1.1 Intuitive analysis. The intuitive analysis involves an assessment of Π_{acsq}'s resilience against three common types of attacks.

Attack 1: send inconsistent blocks. In the broadcast stage, a Byzantine node may send conflicting blocks in an attempt to make nodes deliver inconsistently. However, in the GBC protocol, correct nodes must gather endorsements (i.e., signatures) for the same block from at least $n - f$ nodes before delivering that block. Moreover, each correct node will vote for only one block in a given GBC instance. Given that $n \geq 3f + 1$, at most one block can be voted by $n - f$ nodes. In other words, if two correct nodes deliver blocks B and B', then B and B' must be identical, effectively thwarting the attack.

Attack 2: delay messages. In a GBC instance with a correct broadcaster, the adversary may delay messages to cause only a subset of correct nodes to deliver the block with grade 2 and add it to the ACS set. Meanwhile, other correct nodes might either deliver the block with grade 1 or not deliver it at all. The goal is to cause discrepancies between ACS sets, compromising consistency. However, GBC's delivery-correlation property ensures that at least $f + 1$ correct nodes must have delivered the block with grade 1, which will input 1 in AABA. Under these conditions, AABA's biased-validity property guarantees that all nodes will output 1, thereby adding the block to the ACS set. This maintains the consistency of ACS sets across all nodes, thus protecting against the attack.

Attack 3: input wrong bits to AABA. In AABA, a Byzantine node p_b may deliberately input a wrong bit in an attempt to induce unreasonable outputs, in two ways: (1) p_b delivers a block with grade 1 and broadcasts the second partial signature ρ_j^2. After receiving ρ_j^2, a correct node delivers the block with grade 2 and adds it to the ACS set. However, p_b then inputs 0 to AABA, hoping that others will exclude the block. (2) p_b does not deliver the block but instead inputs 1 to AABA, attempting to trick correct nodes into committing a nonexistent block. We subsequently analyze how Π_{acsq} defends against these attacks. In the first case, GBC's delivery-correlation property ensures at least $f+1$ correct nodes input 1 to AABA. Combined with AABA's biased-validity property, AABA will output 1, regardless of p_b's input. In the second case, AABA requires a proof σ for inputting 1, preventing p_b from inputting an invalid 1.

5.1.2 Rigorous analysis. This analysis focuses on proving whether Π_{acsq} satisfies various ACSQ properties defined in Section 2.3.

Lemma 1. *Within a Π_{acsq} instance, if a correct node includes B in its* ACS *set, then all correct nodes will include B in their* ACS *sets.*

Proof. As shown in Algorithm 4, a block will be decided at three points: at the end of GBC instance (Line 8), at the end of the AABA instance (Line 23), and through the delivery-assistance mechanism (Line 32). We refer to the decision at these three points as GBC-decide, AABA-decide, and DA-decide, respectively, for short. Consider the following two cases:

Case 1: At least one correct node GBC-decides to include B. Denote this node as p_i. By GBC's delivery-correlation property, at least $f + 1$ correct nodes must have delivered B with grade 1. Each of these nodes will input 1 to AABA if it has not grade-2 delivered B. On the other hand, for another correct node p_j, if p_j GBC-decides to include a block B', then B' must be identical to B based on GBC's consistency property. If p_j AABA-decides on a block, according to AABA's biased-validity property, the AABA instance must output 1. Given AABA's 1-validity property, some node must have inputted $\langle 1, v, \sigma \rangle$ s.t. $Q(v, \sigma)$=true. Since σ is a proof of grade-1 delivery, by the receipt-correlation property, at least $f + 1$ correct nodes have received B. p_j can acquire B by broadcasting a query request if it has not received B. In other words, p_j can definitely include B in the ACS set. If p_j has not GBC-decided or AABA-decided on a block, it will eventually receive from p_i a delivery-assistance message (Line 30 of Algorithm 4), thus DA-deciding to include B.

Case 2: No correct node GBC-decides to include B. In this case, all nodes can only AABA-decide on a block. If a correct node includes B in the ACS set, it must output 1 from AABA. According to AABA's agreement property, each correct node will output 1 from AABA. By AABA's 1-validity property, some node have inputted $\langle 1, v, \sigma \rangle$ s.t. $Q(v, \sigma)$=true. Similar to the analysis in Case 1, each correct node can eventually receive B and include B in the ACS set. □

Theorem 2 (Agreement). *Within a Π_{acsq} instance, if a correct node outputs q, then every correct node outputs q.*

Proof. If a correct node excludes a block from the ACS set, it must output 0 from AABA. By AABA's agreement property, each correct node will output 0 from AABA and exclude the block from its ACS set. Conversely, if a correct node includes a block into the ACS set, then, by Lemma 1, each correct node will include this block. When blocks in the ACS set are sorted by their block numbers, the resulting sequences will also match, ensuring agreement. □

Theorem 3 (Validity). *Within a Π_{acsq} instance, if a correct node outputs a sequence q, then $|q| \geq n - f$.*

Proof. As described by Line 10 and Line 12 of Algorithm 4, a correct node will wait to deliver at least $n - f$ blocks with grade 2, each of which will be included in the ACS set and the Π_{acsq} sequence. Thus, the sequence will contain at least $n - f$ elements. □

Theorem 4 (Totality). *Within a Π_{acsq} instance, if each correct node receives an input, all correct nodes will eventually output.*

Proof. Once receiving an input, a correct node will broadcast the block through GBC. Since there are at least $n - f$ correct nodes, each correct node can deliver at least $n - f$ blocks with grade 2, thereby completing the broadcast stage in Algorithm 4. Next, we prove that for each block without being decided on in the broadcast stage, a node can decide on it during the agreement stage.

Assume a node p_i does not decide on a block B in the broadcast stage. We consider the following two cases. First, if no correct node decides on B in the broadcast stage, all correct nodes will activate an AABA instance for B. According to AABA's termination property, p_i will output from AABA and decide on B. Second, if some correct node decides on B in the broadcast stage, it will send a delivery-assistance message to p_i. Therefore, p_i can decide on B after receiving this delivery-assistance message if it has not made a decision based on AABA's output. □

Theorem 5 (Optimistic validity). *Within a Π_{acsq} instance, if all nodes are correct and the network conditions are favorable, each correct node can output a sequence comprising inputs from all nodes, such that $|q| = n$.*

Proof. If all nodes are correct and the network conditions are favorable, each correct node can grade-2 deliver all blocks before the trigger activation during the broadcast stage. Therefore, the outputted sequence will be in size of n. □

5.2 Analysis on Falcon

In this section, we prove that Falcon correctly implements BFT.

Theorem 6 (Safety). *For two nodes p_i and p_j, if $C_i[r] \neq \perp$ and $C_j[r] \neq \perp$, then $C_i[r] = C_j[r]$.*

Proof. As described in Section 4.1.3, blocks in the Π^h_{acsq} instance can be committed only if all Π^m_{acsq} $(m < h)$ instances have been processed. For ease of presentation, we denote the Π_{acsq} sequence generated by node p_i through Π^m_{acsq} as $\mathcal{A}^m_i$. Additionally, the vector updated by node p_i after completing the Π^m_{acsq} instance is denoted as C^m_i. According to Theorem 2, $\mathcal{A}^m_i$ must be equal to $\mathcal{A}^m_j$, and thus C^m_i must also be equal to C^m_j.

Without loss of generality, we assume that $C_i[r]$ and $C_j[r]$ are committed by p_i and p_j in $\mathcal{A}^{h1}_i$ and $\mathcal{A}^{h2}_j$, respectively. If $h1 = 1$, then $h2$ must also equal 1. In other words, $C_i[r]$ and $C_j[r]$ are committed in the first Π_{acsq} instance. According to Theorem 2, $C_i[r]$ and $C_j[r]$ must be identical.

If $h1 > 1$, then $h2$ must be greater than 1, and moreover, $h1$ and $h2$ will be equal. Thus, C^{h1-1}_i must equal to C^{h2-1}_i. Denote the lengths of C^{h1-1}_i and C^{h2-1}_j as l_i and l_j, respectively. l_i and l_j will also be identical. Thus, $C_i[r]$ is committed by p_i in Π^{h1}_{acsq} with index $r - l_i$, and $C_j[r]$ is committed by p_j in Π^{h2}_{acsq} with index $r - l_j$. Since $h1 = h2$ and $l_i = l_j$, $C_i[r]$ and $C_j[r]$ must be identical. □

Theorem 7 (Liveness). *If a transaction* tx *is included in every correct node's buffer, each one will eventually commit* tx.

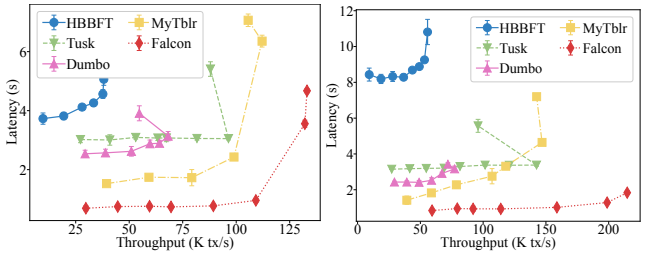
Proof. According to Theorem 3 and Theorem 4, in a Π_{acsq} instance, each correct node will output a sequence q where $|q| \geq n - f$. In this sequence, at least $n - 2f$ blocks are proposed by correct nodes. When tx is present in every correct node's buffer, each one will include tx in its newly proposed block if tx has not already been committed. Therefore, if tx remains uncommitted, the sequence q must include blocks containing tx, which will be committed in this Π_{acsq} instance. Thus, Falcon guarantees the liveness property. □

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation & settings

To evaluate Falcon's performance, we implement the prototype system. We select HBBFT [35], Dumbo[2] [27], MyTumbler (abbreviated as MyTblr) [31], and Narwhal&Tusk (abbreviated as Tusk) [19] as baselines. HBBFT and MyTblr represent the BKR paradigm, Dumbo embodies CKPS, and Tusk represents a *Directed Acyclic Graph* (DAG)-based protocol. Implementations of HBBFT, Dumbo, MyTblr, and Falcon are coded in Rust within the same code framework, without utilizing threshold encryption or erasure codes [35]. The decision not to use erasure codes is also consistent with the recommendation from the BEAT paper [21]. We use ed25519-dalek[3] for PKI signatures and threshold_crypto[4] for threshold signatures.

[2]We utilize sMVBA [26] as the MVBA protocol of Dumbo to achieve good performance.
[3]https://github.com/dalek-cryptography/ed25519-dalek
[4]https://github.com/poanetwork/threshold_crypto



(a) 7 nodes **(b) 16 nodes**

Figure 10: Latency *v.s.* throughput in favorable situations

The ABA protocol is implemented based on the MMR version [36]. For Tusk, we utilize the open-source codebase[5] directly. The experiments are conducted on AWS, with each node deployed on an m5d.2xlarge instance. These instances are equipped with 8 vCPUs and 32 GB of memory and run Ubuntu 20.04, with a maximum network bandwidth of 10 Gbps. To simulate a decentralized deployment, nodes are distributed across five regions globally: N. Virginia, Stockholm, Tokyo, Sydney, and N. California.

To minimize the impact of experimental errors, each experiment is repeated three times, and we plot the average or error bars for each data point in the experiment. We consider two situations: favorable and unfavorable. The favorable situation refers to a system without faulty nodes, while the unfavorable situation includes some faulty nodes. As in most mainstream consensus evaluations [9, 16, 24], we simulate faulty nodes in the system using crash failures.

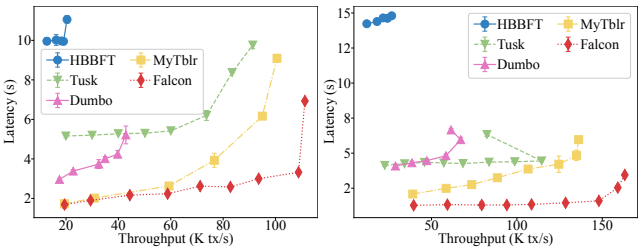
Our prototype implementation is built on a mempool that broadcasts transactions via payloads, a method widely adopted by various blockchain systems [9, 16, 24, 25]. In essence, a payload refers to a package that bundles multiple transactions together. Each payload is configured to be 500 KB. Besides, as in many other consensus studies [9, 16, 27, 35], we generate mock transactions for evaluation. Each transaction is set to 512 bytes, with each block referencing up to 32 payloads. These parameters are also commonly used in recent BFT evaluation works [16, 19, 24, 39].

6.2 Basic performance

We employ two settings: one with 7 nodes and the other with 16 nodes. In each setting, we progressively increase the input rate of transactions until the system reaches saturation.

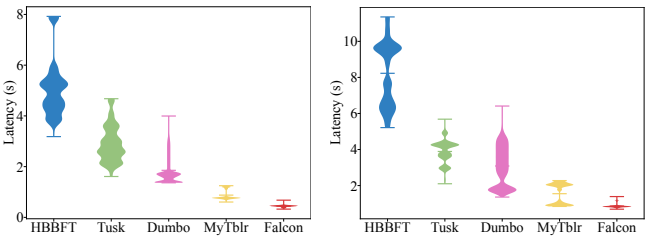
6.2.1 Performance in favorable situations. The trade-off between latency and throughput in favorable situations is shown in Figure 10. It is evident from the graph that Falcon exhibits lower latency and higher peak throughput compared to other protocols. Specifically, when the system consists of 7 nodes, Falcon's latency is only 38.4% of MyTblr, 24.2% of Dumbo, 17.3% of HBBFT, and 23.9% of Tusk. This advantage arises from two factors. First, Falcon avoids running the agreement protocol in favorable situations, while Dumbo and HBBFT always need to execute it. As for Tusk, it relies on the leader-block's reference count mechanism to reach agreements. In other words, it also requires the execution of an agreement protocol. Second, Falcon's partial-sorting mechanism allows faster

[5]https://github.com/facebookresearch/narwhal



(a) 7 nodes (b) 16 nodes

Figure 11: Latency *v.s.* throughput in unfavorable situations



(a) No faulty nodes (b) 3 faulty nodes

Figure 12: Comparison of latency stability

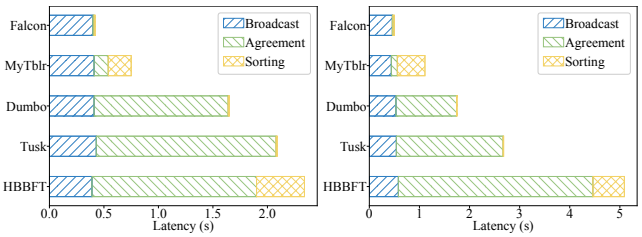
block committing, whereas HBBFT's integral-sorting requires all blocks to be decided before committing. MyTblr, with its timestamp-based sorting, requires blocks to wait until the timestamp reaches a specific threshold for committing. Regarding peak throughput, Falcon achieves 132.1K tx/s, surpassing MyTblr's 112.1K tx/s, Dumbo's 68.2K tx/s, HBBFT's 38.3K tx/s, and Tusk's 96.4K tx/s. The high throughput of Falcon is attributed to its ability to commit more blocks for each ACSQ instance in favorable situations.

6.2.2 Performance in unfavorable situations. In unfavorable situations, we set the number of faulty nodes to 2 and 3 in the 7-node and 16-node settings, respectively. The experimental results are shown in Figure 11. As expected, the performance of all protocols declines when faulty nodes are present in the system compared to Figure 10. However, Falcon still demonstrates better performance than HBBFT, Dumbo, and Tusk. Specifically, in the 7-node setting, Falcon's latency is only 29.4%, 70.7%, 49.7% of that of HBBFT, Dumbo, and Tusk, respectively. This is due to the shortcut mechanism introduced in Falcon's AABA protocol during the agreement stage, which allows AABA instances corresponding to faulty nodes to output more quickly. MyTblr exhibits a latency performance similar to Falcon's because it also incorporates a fast path during the agreement stage, akin to the shortcut mechanism.

6.3 Latency stability

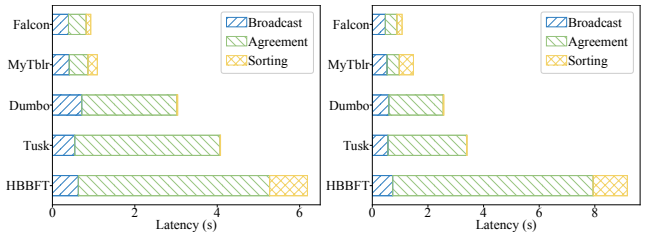
In this set of experiments, we consider 16 nodes and examine both favorable and unfavorable situations. The experimental results are shown in Figure 12.

As shown in the figure, regardless of the situation, Falcon consistently achieves a smaller latency range, due to its ability to continuously commit blocks through the partial-sorting mechanism. In contrast, Dumbo, HBBFT, and Tusk only commit blocks intermittently, experiencing periods of no activity followed by large bursts of commits, which cause significant latency instability. MyTblr also



(a) 7 nodes (b) 16 nodes

Figure 13: Latency decomposition in favorable situations



(a) 7 nodes (b) 16 nodes

Figure 14: Latency decomposition in unfavorable situations

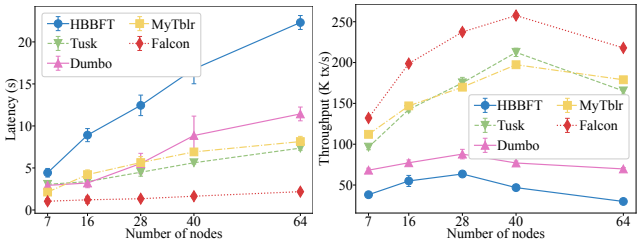
achieves relatively stable latency, owing to its timestamp-based sorting mechanism. However, this mechanism requires the timestamp to reach a specific threshold before committing a block, which leads to higher overall latency and reduced throughput.

6.4 Latency decomposition

We break down the latency to analyze the time consumed by each stage, namely the broadcast, agreement, and sorting. Note that while Tusk does not employ the ABA protocol, it achieves agreement through the leader-block's reference count mechanism. Therefore, we decompose the latency in Tusk as follows: broadcast time (block dissemination), agreement time (from the completion of block dissemination to when the leader block meets the reference count condition), and sorting time (from the leader block meeting the condition to the completion of block sorting).

The experimental results for favorable situations are shown in Figure 13. It is evident that, regardless of the setup, the latency incurred by Falcon during the agreement stage is negligible. This validates Falcon's design of using GBC for direct block decision, eliminating the agreement stage. In contrast, the agreement stage in the other four protocols takes a significant amount of time, particularly in HBBFT, Dumbo, and Tusk. In the 7-node system, the time spent in the agreement stage of HBBFT, Dumbo, and Tusk is 3.9, 2.7, and 3.8 times that of its broadcast stage, respectively. Moreover, Falcon's sorting stage is also negligible due to its partial-sorting mechanism, which accelerates block committing by eliminating the need to wait for all blocks to be decided.

The results for unfavorable situations are shown in Figure 14. Due to the presence of faulty nodes, Falcon needs to run the agreement stage to commit blocks. However, with the shortcut mechanism introduced by AABA, Falcon's agreement stage takes less time than others. Specifically, in the 16-node setting, Falcon's agreement stage consumes only 80.1%, 26.2%, 6.1%, and 14.4% of the time



(a) Latency comparison **(b) Throughput comparison**

Figure 15: Performance comparison as node count increases

required by MyTblr, Dumbo, HBBFT, and Tusk, respectively. As for the sorting stage, Falcon's partial-sorting mechanism continues to reduce latency compared with MyTblr and HBBFT.

An interesting observation is that both Dumbo and Tusk exhibit almost no sorting time, regardless of whether the situations are favorable or unfavorable. This is because Dumbo employs the MVBA agreement protocol, where all blocks are decided simultaneously. Similarly, Tusk's reference count mechanism allows all blocks referenced by the leader to be simultaneously decided. The sorting process following simultaneous decisions is trivial and introduces minimal delay. However, the agreement process itself tends to incur relatively high latency.

6.5 Scalability

We evaluate the scalability of each protocol by increasing the number of nodes and analyzing the corresponding performance variations. Specifically, we focus on the favorable situation and, for each experiment, record the latency and throughput at the point of system saturation. The results are shown in Figure 15.

As shown in Figure 15a, while the latency of all protocols increases with node count, Falcon exhibits a significantly smaller increase compared to the others. For instance, when the number of nodes increases from 7 to 64, Falcon's latency increases by only 2.1 times, whereas MyTblr, Dumbo, HBBFT, and Tusk experience increases of 3.8, 3.9, 5.1, and 2.4 times, respectively. Even with 64 nodes, Falcon's latency remains at 2.2 seconds, which is notably lower than MyTblr's 8.2 seconds, Dumbo's 11.4 seconds, HBBFT's 22.3 seconds, and Tusk's 7.4 seconds.

In terms of throughput, Falcon consistently outperforms others across different node scales. At 64 nodes, Falcon achieves a throughput of 217.9K tx/s, surpassing MyTblr's 178.9K tx/s and Tusk's 165.3K tx/s, and significantly exceeding Dumbo's 69.7K tx/s and HBBFT's 29.8K tx/s. Figure 15b shows that the throughput of all protocols initially increases and then decreases as the number of nodes grows. This pattern is linked to the mempool mechanism in Falcon, MyTblr, Dumbo, and HBBFT, where each node generates and broadcasts payloads. In the early period, more nodes result in more payloads, boosting throughput. However, as the number of nodes increases further, the surge in consensus and payload messages leads to network congestion, which ultimately reduces throughput. For Tusk, a similar reasoning applies, as it depends on the Narwhal mempool. Despite this, Falcon consistently maintains higher throughput than the other protocols throughout all periods.

7 RELATED WORK

In this section, we summarize the related work on asynchronous BFT consensus. The discussion of (partially) synchronous protocols is deferred to Appendix B of [15] due to space constraints.

Asynchronous BFT consensus protocols rely on two key components: ABA [2, 23, 36] and MVBA [4, 26, 33]. ABA is designed for binary values, while MVBA handles arbitrary values. When combined with RBC or the GBC protocol proposed in this paper, ABA and MVBA enable the implementation of asynchronous BFT consensus, known as the BKR and CKPS paradigms, respectively. Among these, HBBFT, a representative of BKR, is recognized as the first asynchronous BFT protocol that can be practically deployed [35].

Building on HBBFT, Duan et al. introduce five versions of the BEAT protocol to improve it [21]. BEAT0 and BEAT2 optimize the cryptographic protocols, which are orthogonal to our work. BEAT3 and BEAT4 focus on storage system optimizations, which diverge from our goal of the general SMR. BEAT1 suggests replacing the encoded RBC in HBBFT with a non-encoded RBC, which aligns with our implementation in Section 6.1. Liu et al. also proposed a new protocol, MyTumbler, to improve BKR's performance [31]. However, even in favorable situations, this protocol still requires running the ABA protocol. In contrast, Falcon avoids the ABA instances in optimistic situations, resulting in better performance.

Some works have introduced an optimistic path into asynchronous protocols to enhance their performance in favorable conditions [30, 38]. Depending on how the optimistic and pessimistic paths are structured, these approaches can be classified into the serial path paradigm [24, 32] and the parallel path paradigm [9, 16]. They, however, can only commit a single block in optimistic situations, resulting in lower throughput. In contrast, Falcon can commit n blocks in favorable conditions, offering higher throughput.

Additionally, works like DAGRider [29] have incorporated the topology of *Directed Acyclic Graph* (DAG) into the design of consensus protocols. Building on DAGRider, protocols such as Tusk [19], BullShark [40], GradedDAG [18], and Wahoo [17] aim to optimize the latency of DAG-based BFT consensus. However, these protocols focus primarily on leader blocks and neglect non-leader blocks, which results in higher latency for non-leader blocks.

8 CONCLUSION

To address the key challenges of existing BFT consensus, we present Falcon, a protocol that offers both low and stable latency, as well as improved throughput. Specifically, by introducing the GBC protocol, Falcon effectively bypasses the agreement stage in favorable situations, significantly reducing latency. The AABA protocol ensures consistency between decisions made via GBC and those made during the agreement stage. Besides, Falcon's partial-sorting mechanism enhances latency stability by enabling continuous block committing. Integrating an agreement trigger further boosts throughput by allowing nodes to deliver and commit more blocks. Experimental results demonstrate Falcon's superiority over existing protocols, positioning it as a promising solution for advancing BFT consensus.

ACKNOWLEDGMENTS

This work was supported by National Science and Technology Major Project 2022ZD0115301.

REFERENCES

[1] Ittai Abraham and Gilad Asharov. 2022. Gradecast in Synchrony and Reliable Broadcast in Asynchrony with Optimal Resilience, Efficiency, and Unconditional Security. In *Proceedings of the 41st ACM Symposium on Principles of Distributed Computing*. ACM, 392–398.

[2] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proceedings of the 41st ACM Symposium on Principles of Distributed Computing*. ACM, 381–391.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync Hotstuff: Simple and Practical Synchronous State Machine Replication. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 106–118.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 337–346.

[5] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. 2022. Qanaat: A Scalable Multi-enterprise Permissioned Blockchain System with Confidentiality Guarantees. In *Proceedings of the 48th International Conference on Very Large Data Bases*, Vol. 15. VLDB Endowment, 2839–2852.

[6] Michael Ben-Or. 1983. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*. ACM, 27–30.

[7] Michael Ben-Or, Ran Canetti, and Oded Goldreich. 1993. Asynchronous Secure Computation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. ACM, 52–61.

[8] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous Secure Computations with Optimal Resilience. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 183–192.

[9] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. 2023. Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus. In *Proceedings of the 30th ACM Conference on Computer and Communications Security*. ACM, 519–533.

[10] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Information and Computation* 75, 2 (1987), 130–143.

[11] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Proceedings of the 21st Annual International Cryptology Conference*. Springer, 524–541.

[12] Christian Cachin and Stefano Tessaro. 2005. Asynchronous Verifiable Information Dispersal. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. IEEE, 191–201.

[13] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 173–186.

[14] TH Hubert Chan, Rafael Pass, and Elaine Shi. 2018. Pili: An Extremely Simple Synchronous Blockchain. *Cryptology ePrint Archive* (2018).

[15] Xiaohai Dai, Chaozheng Ding, Wei Li, Jiang Xiao, Bolin Zhang, Chen Yu, Albert Y. Zomaya, and Hai Jin. 2025. Falcon: Advancing Asynchronous BFT Consensus for Lower Latency and Enhanced Throughput (full version). arXiv:2504.12766 https://arxiv.org/abs/2504.12766

[16] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. In *Proceedings of the 30th ACM Conference on Computer and Communications Security*. ACM, 504–518.

[17] Xiaohai Dai, Zhaonan Zhang, Zhengxuan Guo, Chaozheng Ding, Jiang Xiao, Xia Xie, Rui Hao, and Hai Jin. 2024. Wahoo: A DAG-Based BFT Consensus With Low Latency and Low Communication Overhead. *IEEE Transactions on Information Forensics and Security* 19 (2024), 7508–7522.

[18] Xiaohai Dai, Zhaonan Zhang, Jiang Xiao, Jingtao Yue, Xia Xie, and Hai Jin. 2023. GradedDAG: An Asynchronous DAG-based BFT Consensus with Lower Latency. In *Proceedings of the 42nd International Symposium on Reliable Distributed Systems*. IEEE, 107–117.

[19] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. In *Proceedings of the 17th European Conference on Computer Systems*. ACM, 34–50.

[20] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and its Applications. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2705–2721.

[21] Sisi Duan, Michael Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2028–2041.

[22] Sisi Duan, Xin Wang, and Haibin Zhang. 2023. FIN: Practical Signature-free Asynchronous Common Subset in Constant Time. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 815–829.

[23] Roy Friedman, Achour Mostefaoui, and Michel Raynal. 2005. Simple and Efficient Oracle-based Consensus Protocols for Asynchronous Byzantine Systems. *IEEE Transactions on Dependable and Secure Computing* 2, 1 (2005), 46–56.

[24] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-adaptive Efficient Consensus with Asynchronous Fallback. In *Proceedings of the 26th International Conference on Financial Cryptography and Data Security*. Springer, 296–315.

[25] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-stm: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, 232–244.

[26] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. *Cryptology ePrint Archive* (2022).

[27] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster Asynchronous BFT Protocols. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 803–818.

[28] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. An In-depth Look of BFT Consensus in Blockchain: Challenges and Opportunities. In *Proceedings of the 20th International Middleware Conference*. ACM, 6–10.

[29] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*. ACM, 165–175.

[30] Klaus Kursawe and Victor Shoup. 2005. Optimistic Asynchronous Atomic Broadcast. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*. Springer, 204–215.

[31] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. 2023. Flexible Advancement in Asynchronous BFT Consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, 264–280.

[32] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo Transformer: Asynchronous Consensus as Fast as the Pipelined BFT. In *Proceedings of the 29th ACM Conference on Computer and Communications Security*. ACM, 2159–2173.

[33] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. 2020. Dumbo-MVBA: Optimal Multi-valued Validated Asynchronous Byzantine Agreement, Revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*. ACM, 129–138.

[34] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-CHAIN: Low latency, High Throughput BFT Consensus on a DAG. In *Proceedings of the 28th International Conference on Financial Cryptography and Data Security*. Springer, 51–73.

[35] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 31–42.

[36] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free Asynchronous Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*. ACM, 2–9.

[37] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. 2022. Neuchain: A Fast Permissioned Blockchain System with Deterministic Ordering. In *Proceedings of the 48th International Conference on Very Large Data Bases*, Vol. 15. VLDB Endowment, 2585–2598.

[38] HariGovind Ramasamy and Christian Cachin. 2005. Parsimonious Asynchronous Byzantine-fault-tolerant Atomic Broadcast. In *Proceedings of the 9th International Conference On Principles of Distributed Systems*. Springer, 88–102.

[39] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards Improving Latency of DAG-based BFT. In *Proceedings of the 46th IEEE Symposium on Security and Privacy*. IEEE.

[40] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of ACM Conference on Computer and Communications Security*. ACM, 2705–2718.

[41] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. 2022. BFT in Blockchains: From Protocols to Use Cases. *Comput. Surveys* 54, 10 (2022), 1–37.

[42] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: An Elastic Disaggregated Blockchain. In *Proceedings of the 48th International Conference on Very Large Data Bases*, Vol. 16. VLDB Endowment, 23–36.

[43] Yang Xiao, Ning Zhang, Wenjing Lou, and Thomas Hou. 2020. A Survey of Distributed Consensus Protocols for Blockchain Networks. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 1432–1465.

[44] Maofan Yin, Dahlia Malkhi, Michael Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of ACM Symposium on Principles of Distributed Computing*. ACM, 347–356.

[45] Haibin Zhang and Sisi Duan. 2022. Pace: Fully Parallelizable BFT from Reproposable Byzantine Agreement. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 3151–3164.

[46] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain Challenges and Opportunities: A Survey. *International Journal of Web and Grid Services* 14, 4 (2018), 352–375.