ABSE: Adaptive Baseline Score-Based Election for Leader-Based BFT Systems

Xuyang Liu, *Graduate Student Member, IEEE*, Zijian Zhang, *Senior Member, IEEE*, Zhen Li, Hao Yin, Meng Li, *Senior Member, IEEE*, Jiamou Liu, Mauro Conti, *Fellow, IEEE*, and Liehuang Zhu, *Senior Member, IEEE*

***Abstract*—Leader-based BFT systems face potential disruption and performance degradation from malicious leaders, with current solutions often lacking scalability or greatly increasing complexity. In this paper, we introduce ABSE, an Adaptive Baseline Score-based Election approach to mitigate the negative impact of malicious leaders on leader-based BFT systems. ABSE is fully localized and proposes to accumulate scores for processes based on their contribution to consensus advancement, aiming to bypass less reliable participants when electing leaders. We present a formal treatment of ABSE, addressing the primary design and implementation challenges, defining its generic components and rules for adherence to ensure global consistency. We also apply ABSE to two different BFT protocols, demonstrating its scalability and negligible impact on protocol complexity. Finally, by building a system prototype and conducting experiments on it, we demonstrate that ABSE-enhanced protocols can effectively minimize the disruptions caused by malicious leaders, whilst incurring minimal additional resource overhead and maintaining base performance.**

***Index Terms*—Distributed consensus, Byzantine fault tolerance, leader election, score-based mechanism.**

Received 30 November 2024; revised 8 April 2025; accepted 18 May 2025. Date of publication 22 May 2025; date of current version 20 June 2025. This work was supported in part by National Key Research and Development Program of China under Grant 2023YFF0905300, and in part by the National Natural Science Foundation of China (NSFC) under Grant U2468205, Grant 62372149, and Grant U23A20303, and in part by the China Scholarship Council (CSC), and in part by the Key Laboratory of Knowledge Engineering with Big Data (the Ministry of Education of China) under Grant BigKEOpen2025-04. Recommended for acceptance by J. Xu. *(Corresponding authors: Zijian Zhang; Meng Li.)*

Xuyang Liu is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100811, China, and also with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: liuxuyang@bit.edu.cn).

Zijian Zhang and Liehuang Zhu are with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100811, China (e-mail: zhangzijian@bit.edu.cn; liehuangz@bit.edu.cn).

Zhen Li is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100811, China (e-mail: zhen.li@bit.edu.cn).

Hao Yin is with the Research Center of Cyberspace Security, Peking University Changsha Institute for Computing and Digital Economy, Changsha 410205, China (e-mail: yinhao@icode.pku.edu.cn).

Meng Li is with the Key Laboratory of Knowledge Engineering with Big Data, Hefei University of Technology, Hefei 230002, China, also with the Ministry of Education, School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230002, China, also with the Intelligent Interconnected Systems Laboratory of Anhui Province, Hefei University of Technology, Hefei 230002, China, and also with the HIT Center, University of Padua, 35122 Padova, Italy (e-mail: mengli@hfut.edu.cn).

Jiamou Liu is with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: jiamou.liu@auckland.ac.nz).

Mauro Conti is with the Department of Mathematics and HIT Center, University of Padua, 35131 Padua, Italy, and also with Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: mauro.conti@unipd.it).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2025.3572553, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2025.3572553

I. Introduction

The Byzantine Generals Problem [1] is a key challenge in distributed computing. It depicts a situation where generals, some potentially treacherous, must agree on a unified strategy. The key is to create a system where consensus can be reached despite potential deception. This metaphor speaks to unreliable processes in a distributed network [2]. Derived from this, Byzantine Fault Tolerance (BFT) allows distributed systems to operate accurately amidst Byzantine faults. BFT protocols [3] use a proposal and voting mechanism to reach consensus on system state or transaction validity, even if some processes fail or act maliciously (a scenario echoing the treacherous generals in the Byzantine problem).

Leader-based BFT: A class of such protocols that is particularly popular are leader-based protocols where one process (the leader) is tasked with proposing and broadcasting proposals of new data to be applied to the state machine, while other processes verify and vote on the leader's proposals. The leader is chosen through an election process, which can be deterministic [4], [5], random [6], [7], round-robin [8], [9], or other possible forms. In a leader-based BFT system, the consensus flow (the exact steps may vary depending on the specific protocol) typically starts with a "Leader Election". After that, the leader submits a proposal to the network. The processes vote on the proposal during the voting phase, with most protocols requiring a majority for acceptance. If the majority is achieved, the network commits to the value and subsequently moves to the next consensus round. If not achieved, the network aborts and may elect a new leader to restart the flow. Leader-based BFT systems provide various benefits, including reducing the complexity of messages and communication overheads due to the presence of a coordinating leader. Furthermore, a leader can organise and coordinate the system's work more effectively, increasing its efficiency.

Malevolence by Leaders: In BFT systems, it is generally assumed that all processes are equivalent, a principle which extends to leader-based BFT systems. Nevertheless, processes becoming leaders, due to their additional roles compared to normal processes, significantly impact the system. Specifically, if a malicious process ascends to leadership, it can drastically interrupt the consensus flow (e.g., through manipulation of data packing broadcasts) until a new leader is elected. At its most fundamental level, a malicious leader can delay the current consensus process, effectively reducing system throughput (as if it were down). Indeed, considerable performance degradation due to malicious processes has been observed in most leader-based BFT systems [10], [11], [12].

Additionally, in BFT systems related to trading [13] or finance [14], there has been a focus [15] on the role of the leader in ensuring fair transaction ordering [16]. In such systems, the data to be packaged, such as transactions or bids, are often highly susceptible to the effects of ordering. In such a context, malicious leaders can directly choose the final transaction ordering during the packaging proposal process [15]. Although proposals containing misordered transactions may not be accepted by the majority, they nonetheless pose a potential security risk to the system.

While some studies [17], [18], [19], [20], [21] propose solutions to the problems posed above, the majority of these solutions either lack scalability (i.e. only applicable to specific types of BFTs or based on specific network assumptions) or significantly increase system complexity, which in turn negatively impacts performance. Our goal is to develop a general approach for leader-based BFT systems that minimizes the effects of malicious leaders with minimal increase in system complexity and without compromising baseline performance.

Score-based Approach for Leader-based BFTs: Scoring strategies, commonly used in multi-party participatory systems to assess participant reliability, are applicable to BFT systems. We propose to establish a score-based approach for processes in leader-based BFTs, where the scores reflect the contribution of a process to consensus advancement (e.g., awarding points for votes that lead to a committed consensus). Processes hitting a certain score would be eligible for leadership. This approach, however, faces three primary challenges. First, there is no guarantee all processes consistently score a process, which could introduce additional inconsistencies. Besides, if the scoring process involves communication, it could significantly increase consensus complexity and potentially open the door to malfeasance. Second, it is challenging to design a score-based approach while minimizing additional complexity to the system. Finally, constructing a set of protocol-independent components for the approach remains challenging, i.e., applicable to most leader-based BFT systems.

A. Our Contributions

A formal treatment of ABSE: We propose ABSE - an Adaptive Baseline Score-based Election approach for Leader-based BFT systems. The fundamental concept of ABSE is that consensus-contributing processes accumulate scores indicative of their reliability. The objective is to filter out less reliable participants, bypassing rounds where they ascend to the leader position, thus enhancing protocol performance under unstable conditions or in the presence of adversaries. We aim to identify a relatively fair scoring method with a baseline score under the premise of independent process scoring and to mitigate the effects of malicious leaders without impacting the system's baseline performance. We also strive to create a solution that can be universally implemented, benefiting a wide range of BFTs. More crucially, our treatment has the following features:

- To ensure global consistency, we establish the rules that ABSE should adhere to. Specifically, we formally define three scenarios in which processes may diverge when choosing a leader. These scenarios cover all possible situations during leader election, and a protocol applying ABSE needs to maintain consistency across all three scenarios.
- We provide the generic components of ABSE, which include data structures and essential utilities. Using these components, we illustrate a generic procedure for ABSE. We show that these components and procedures are protocol-independent and are highly scalable according to the demands of different scenarios.
- ABSE is fully localized. All ABSE-related data structures are excluded from communication. ABSE minimally influences protocol complexity and is suitable for most leader-based BFT protocols.

Two ABSE application instances: To demonstrate the scalability, we apply ABSE to two different protocols: the Chained-HotStuff protocol under the partial synchronization assumption (Section V), and the DAG-Rider protocol under the asynchronous assumption (Section VII). We create ABSE modules based on the primary algorithms of the two protocols, offering a detailed explanation and progression of ABSE construction. We provide the modified protocols' content along with pseudocode. We show that ABSE has a negligible effect on protocol complexity and does not disrupt the protocol's normal flow or introduce further inconsistencies. Furthermore, we offer a formal proof of security for all protocols.

For code implementation, we have successfully applied ABSE to Chained-HotStuff and DAG-Rider, adding less than 300 lines of code for each. Our experimental results demonstrated that ABSE-enhanced versions can substantially mitigate the impacts of malicious leaders while incurring minimal additional resource overhead. Furthermore, the baseline performance of the original protocol is practically unaffected.

II. Related Work

BFT: BFT is a generic methodology that allows for the toleration of malicious participants. Depending on the network assumptions, BFT can be classified into synchronous BFT [22], [23], partially synchronous BFT [4], [24], and asynchronous BFT [25], [26]. The majority of BFTs employ a leader-based model. The progression of consensus per round necessitates a leader responsible for proposal initiation and voter coordination. However, the leader's role can also fluctuate depending on the BFT protocol, e.g. in DAG-based BFTs, where the leader serves as the cut-off point for ordering, symbolizing a topological point from a graph-theoretic viewpoint [6].

Investigations in Leader-Based BFT: The impact of leader misbehavior can significantly affect leader-based BFT, and a wealth of research has been dedicated to this issue, proposing various countermeasures. For example, Aequitas [15] prevents adversarial manipulation of transaction ordering in the log by introducing the transaction order-fairness property. Themis [27] achieves standard liveness with the fair ordering. Prosecutor [19] hinders Byzantine servers from attaining leadership by imposing hash computations on new election campaigns. Some investigations [28], [29], [30] have explored solutions at the communication level, drawing from high-performance computing and task-based distributed systems [31], [32]. We take a more flexible approach to address leader misbehavior by filtering out less reliable participants when electing leaders, retaining the role of the leader without introducing additional constraints. Our ABSE is a versatile 'plug-and-play' score-based enhancement for leader-based BFTs, offering flexibility with adjustable baselines, which can be set to zero to disable ABSE when desired.

Leaderless BFT: Several studies [33], [34] have scrutinized the significant influence of a leader on consensus performance. Some research [35] even suggests that the leader has become an obstacle to consensus scalability within large networks. Crain et al. [36] consider the Democratic BFT consensus algorithm as leaderless. HoneyBadgerBFT [37] operates in a fully asynchronous environment and offers better resilience to Byzantine faults as it does not rely on individual correctness. Since then, a series of leaderless asynchronous BFT protocols have been proposed [38], [39].

Previous schemes adopting reputation mechanisms: Scoring strategies or reputation mechanisms are frequently employed in multi-party participatory systems [40], [41], including BFTs [42], [43]. However, ensuring all participants consistently score an event or participant without augmenting system centrality is challenging and often necessitates the addition of extra consensus rounds. FCBFT [44] introduces a reputation score calculation formula to form independent consensus groups via feature grouping. Carousel [20] uses the available on-chain information to determine a leader locally without relying on consensus, but is too complex. Most recent Shoal [21] and Hammerhead [45] are simpler, and have the merit of requiring no changes to the consensus protocol itself, but only for DAG-based consensus. Moreover, most of these existing schemes are based only on synchronous or partially synchronous assumptions and lack support under asynchrony. ABSE, on the other hand, is simple, generic and protocol-independent, providing a reputation-based election mechanism for a wide range of leader-based BFTs. It relies only on voting information to score processes, and does not introduce any extra communication overhead or inconsistency.

Directed Acyclic Graph (DAG): The DAG approach, introduced chiefly by Aleph [46], marks a notable progression in BFTs, utilizing a graph-based structure to address the scalability issues of conventional chain-based protocols. On this basis, DAG-Rider [6] and Narwhal [7] managed to achieve zero communication overhead and high throughput under asynchrony by dividing the protocol into two layers: processes reliably broadcast proposals and construct a structured DAG of the communication between them in the first layer, and observe their DAG locally and fully sort proposals without extra communication in the second layer. Further practical solutions under partial synchronization have been proposed by Bullshark [12], which also introduce garbage collection mechanisms. ABSE is also applicable to DAG-based BFTs.

III. Overview

A. Model

The system incorporates a collection of n processes, designated as $\Pi = \{p_1, ..., p_n\}$. These processes manage transactions dispatched by the client to maintain the global consistency of operation logs. Transactions are batched after a specified period or upon reaching the maximum batch size, and subsequently processed. The system can contain up to f processes under the control of an adversary, provided that $f < n/3$. For simplicity, we define $n = 3f + 1$ as the total number of processes. Processes compromised by the adversary exhibit Byzantine faults with unpredictable behaviour. The remaining processes communicate authentically and adhere to the protocol. When a message is transmitted from one correct process to another, it eventually reaches the recipient, who can then validate the sender's identity using cryptographic primitives. We assume each process p_i possesses a public/private key pair designated as (pk_i, sk_i) and uses the public key as its identity. BFT protocols are subject to different assumptions in partially synchronous and asynchronous environments, respectively, which we discuss separately below.

Partial Synchronization Assumption: Under partial synchronization, a valid proposer must await vote messages from other validators before proposing. This introduces a time delay, bound by a known finite value Δ, and a significant event known as the Global Stabilization Time (GST) [47]. Post-GST, messages transmitted among correct processes are delivered within the time frame of Δ. We take into account an adversary capable of corrupting up to f processes dynamically, with potential for arbitrary behaviour, including protocol violation or non-response. The adversary may intercept network communication but lacks the capacity to impede or alter messages during point-to-point transmission. Additionally, it must induce the GST event after an unknown but finite time period. For the protocol to be effective, it must adhere to: **(1) Safety.** All correct processes share a uniform view of the system and maintain an identical, ordered log of transactions. **(2) Liveness.** Requests from a correct client are eventually delivered and committed by all correct processes.

Asynchronous Assumption: Messages are only bound to reach eventually in asynchrony. There is no restriction on message delivery time. We consider an adaptive adversary, with the ability to dynamically corrupt up to f processes during the operational period. Once a process has been corrupted, any undelivered messages previously sent from that process to others can be dropped. The adversary has control over the arrival times of messages. Reliable broadcast [25], [48], [49] serves as an effective building block in asynchronous consensus protocols, providing a basic communication primitive that ensures reliable message delivery and agreement. The reliable broadcast abstraction guarantees the following properties:

- *Agreement:* If a correct process p_i delivers a message m, then all correct processes eventually deliver the same m.

- *Integrity:* Each correct process delivers a message a maximum of one time, irrespective of the message content.
- *Validity:* If a correct process p_i broadcasts a message m, then every correct process eventually deliver the same m.

Byzantine Atomic Broadcast (BAB) [50] is one of the problems that asynchronous consensus focus on, enabling processes to agree on a message sequence required for State Machine Replication [51]. FLP result [52] demonstrates that BAB cannot be solved in an asynchronous setting deterministically (note that the original only considers crash fault assumptions, since what is impossible under crash is also impossible for the Byzantine fault model). Therefore, additional building blocks are required to ensure randomness that guarantees liveness. A BAB protocol satisfies reliable broadcast (agreement, integrity, and validity) as well as: **Total order**: If a correct process p_i delivers the message m before the message m', then no correct process delivers m' without first delivering m.

B. ABSE Specification

ABSE is suitable for leader-based BFTs where all (or most) processes are eligible to be leaders, especially protocols where leaders rotate frequently (e.g., random or round-robin, and we use these two modes as instances since the protocols used in this paper employ these types of leader rotation, respectively). The fundamental concept of ABSE is consensus-contributing processes accumulate scores indicative of their reliability. The objective is to filter out less reliable participants, bypassing rounds where they ascend to the leader position, thereby enhancing protocol performance under unstable conditions or in the presence of adversaries. For each process, we maintain a matrix $Scores$ where $Scores[r]$ represents the scores of all processes in round r. Specifically, $Scores[r][j]$ denotes the score of process p_j in r. As we will explain shortly, each score in $Scores$ is associated with a unique number of rounds and a source process. At any given time, p_i's $Scores[r]$ for $r \in N$ are the scores of all processes linked with r, as perceived by p_i. $Scores$ does not partake in message transmission; instead, each process locally preserves its own $Scores$.

Score accumulation: The $Scores[r]$ of the current round r is derived from $Scores[r-1]$. Specifically, when the process gains access to the voting information of a consensus round r, $Scores[r]$ is determined by updating the scores of consensus-promoting processes[1] (which can include both the voting processes and the consensus-promoting leader) based on $Scores[r-1]$. If process p_j is a consensus-promoting process in round r, then its score is updated as $Scores[r][j] = Scores[r-1][j] + X_j^{(r)}$, where $X_j^{(r)}$ is a configurable value associated with p_j's specific contribution type in r. Once $Scores[r]$ has accumulated the required number (a preset value, e.g. 2in the example of Fig. 1) of confirmations (later denoted as ct), it becomes the reference score for the current process to evaluate other processes, and is subsequently used to determine process reliability in leader election.

We implement $Scores[]$ as a First-In-First-Out (FIFO) queue that stores score arrays for the most recent rounds. The queue has a fixed size equal to ct plus 1. Only the most recent $ct + 1$ rounds of scoring information need to be maintained, with older data being removed as new rounds are processed. In each round r, the current array $Scores[r]$ is pushed into the queue. When round $r + ct$ is reached, $Scores[r]$ is popped from the queue, indicating it has accumulated the required number of confirmations, and is then used to update the reference score that determines process reliability in leader election. Initially, $Scores[]$ is set to all [0]. Fig. 1 provides a detailed example of how $Scores[]$ evolves over multiple rounds with 4 processes and 2 confirmations ($ct = 2$) from the perspective of p_1.

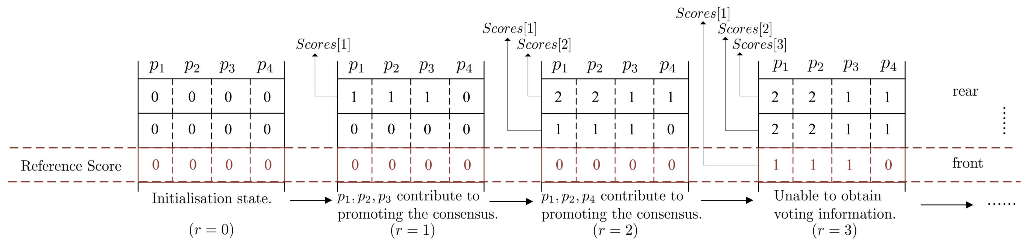
Leader choices above baseline scores: Baseline scores are computed based on r and increase concurrently with r. In each **leader election round** (we emphasize this point because not all BFT protocols choose leaders on a 1-round basis. E.g., DAG-Rider contains 4 rounds per wave, with the leader randomly elected each wave, so leader is elected on a 4-round basis), each process initially verifies whether the leader has a reference score that surpasses the baseline score. If so, the process endorses the selected process as the leader and casts vote for it; otherwise, it triggers a view jump to search for a reliable leader in the upcoming rounds. It should be noted that in BFT protocols, a process can be elected as the leader only with the agreement of the majority of processes(i.e., $2f + 1$), whereas the processes, which only keep their individual $Scores[]$, are likely to diverge in their choice of leader. This divergence can be divided into three scenarios and needs to be fully satisfied in order to ensure consistency:

(I) At least $2f + 1$ processes agree on the leader, while the remaining processes trigger view jumps to search for a reliable leader in the upcoming rounds. In this scenario, the leader is successfully elected and completes the proposal (in partially synchronous protocols, processes that failed in their early attempts to proceed to the next round did not update their local locked value, so this did not affect their ability to follow the normal process of the protocol properly. The remaining processes not agreeing initially on the new leader will follow post-timeout, entering next round. Since $2f + 1$ majority has reset timers upon entering new round, these processes can vote without causing timeouts, maintaining progress).

(II) At least $2f + 1$ processes deem the leader as unreliable and already find the same reliable leader in the upcoming rounds. In this scenario, the original leader is bypassed and the new reliable leader is successfully elected. The remaining processes not agreeing initially on the new leader can catch up by obtaining a valid proposal from the higher round and then vote in the jumped-to round.

(III) No more than $2f + 1$ (at most $2f$) processes agree on the same leader. under partial synchronization, the current round cannot elect a valid leader, triggering a timer timeout and no proposal is made. Subsequently, the next round is entered

[1] The identification of consensus-promoting processes is configurable based on specific scenarios or requirements, which is a key component of scoring mechanism design. In the basic and generic ABSE model we construct later (and use in ABSE-Chained-HotStuff and ABSE-DAG-Rider), consensus-promoting processes are defined as those that contribute valid votes leading to successful consensus in a given round, as well as the leader of the corresponding round (also referred to as a consensus-promoting leader). Furthermore, for both behaviors, $X = 1$.



Scores[1]
Scores[2]
Scores[3]
p1 p2 p3 p4
rear
front
Reference Score
Initialisation state. (r = 0)
p1, p2, p3 contribute to promoting the consensus. (r = 1)
p1, p2, p4 contribute to promoting the consensus. (r = 2)
Unable to obtain voting information. (r = 3)
......

Fig. 1. An example of iterations of $Scores[]$ with $n = 4, ct = 2$ from the perspective of p_1 (for simplicity, we do not distinguish between consensus-promoting behaviors here and set $X_i^{(k)} = 1$ if p_i promotes the consensus in round k.).

normally according to the protocol process. In asynchronous protocols, processes put the the proposal for the current round on hold, temporarily skipping the current round and completing the proposal that has been put on hold when a reliable leader is elected in a subsequent round.

Scenarios I-II guarantee that even if the $2f + 1$ processes agreeing on the leader contain malicious processes, the remaining f processes can eventually synchronize the process and continue to vote, thus ensuring consensus. Scenario III[2] ensures a return to the normal flow of protocols when a leader cannot be elected with ABSE.

In all scenarios, we need to obtain definitive results among all correct processes, rather than ambiguous outcomes. This is critical because ABSE introduces new round flow patterns where processes may (temporarily or permanently) skip multiple rounds and enter higher rounds ahead of others due to divergent local scoring. Without proper handling, such behavior could result in processes operating in different rounds. A universal correctness proof for ABSE as a leader election mechanism is currently impractical (but remains a future possibility) because it involves replacing the existing mechanisms of protocols, meaning it's tied to many factors including the protocol's original mechanisms and system model. Therefore, when applying ABSE to specific protocols, providing new sets of proofs (demonstrating consistency in all three scenarios when ABSE serves as the protocol's new leader election mechanism, and proving correctness when the protocol follows ABSE flow) is essential.

We then analyse the effectiveness of ABSE, which provides probabilistic rather than deterministic guarantees for failure detection. Let's assume a system with $n = 3f + 1$ processes. Consider a malicious process p_i that participates in consensus. In each round: When p_i behaves correctly, it earns an expected score $E[C]$ per round in correct processes; When p_i exhibits malicious behavior, it earns an expected score $E[F]$ per round in correct processes; Since malicious processes can strategically behave correctly in some rounds to accumulate score, we have $0 \le E[C], E[F] \le E[C]$. Let $Pr(M|p_i)$ represent the probability that p_i exhibits malicious behavior in a given round.

[2] Frequent occurrence of Scenario III typically indicates that even correct processes cannot keep pace with the baseline increase, which would prevent the system from electing a reliable leader. This can lead to no performance advantage or even performance degradation compared to the original protocol without ABSE. Therefore, the specific design of the scoring mechanism and baseline is critical for ABSE's effectiveness.

The expected score accumulation for p_i after r rounds can be expressed as: $E[Score(p_i, r)] = r \times (Pr(M|p_i) \times E[F] + (1 - Pr(M|p_i)) \times E[C])$.

The baseline typically needs to be set as an r-based calculation function to ensure it remains effective at reducing the election probability of unreliable processes as consensus rounds progress. Let $baseline{=}rule(r)$, $rule(r)$ should $< r \times E[C]$ for large enough r (to ensure correct processes eventually exceed the baseline). To better illustrate how to achieve practicality in design, we provide a formal analysis based on a basic and generic baseline and scoring mechanism.

Suppose correct processes earn scores (at correct processes) with probability $Pr[C]$, while malicious processes may deviate from protocol with earning $Pr[F] \le Pr[C]$. Let RULE(r)= $\beta \cdot r$ where $\beta < Pr[C]$ (i.e., the growth rate is β). For any process p_i, define its cumulative score at round r as $S_i^{(r)} = \sum_{k=1}^{r} X_i^{(k)}$, where $X_i^{(k)}$ is a Bernoulli random variable: $X_i^{(k)} = 1$ if p_i contributes a valid vote in k that leads to a successful consensus or is the consensus-promoting leader in k, otherwise 0. This is a basic and generic model because we don't consider more complex baseline growth functions or scoring mechanism (definition of promoting consensus) or non-binary score possibilities per round (which could even include penalties or score deduction rule), though more complex models can be derived from this foundation.

For correct processes p_i: $E[X_i^{(k)}] = Pr[C] \Rightarrow E[S_i^{(r)}] = Pr[C] \cdot r$. For malicious processes p_j: $E[X_j^{(k)}] = Pr[F] \Rightarrow E[S_j^{(r)}] = Pr[F] \cdot r$. There should exist a critical round r^* such that: $\forall r > r^* : \beta \cdot r < Pr[C] \cdot r - \epsilon_c$ and $\beta \cdot r > Pr[F] \cdot r + \epsilon_c$ where ϵ_c is a convergence precision parameter. For correct processes p_i, the misclassification probability decreases exponentially: $Pr(S_i^{(r)} < \beta r) \le e^{-r \cdot D(Pr[C]||\beta)}$ where $D(\cdot||\cdot)$ is the KL divergence.

$rule(r)$ can be chosen based on the specific application scenario. For instance, its growth rate can be set higher if the network is better, and vice versa. Following the basic and generic model considered above, a straightforward approach is to consider the expected number of valid votes per process observed from one's own perspective after r rounds. Since the voting threshold is $2f + 1$, assuming replicas vote with equal probability of $Pr[V] = (2f + 1)/(3f + 1)$ per round. If every process broadcasts its vote in each round, then the expected cumulative number of votes per process at round r is $E[S_i^{(r)}] = (r - ct) \cdot Pr[V]$, where ct is the number of confirmations required. If each process only casts vote to the leader in each round, then $Scores[]$ is only updated through votes when a process becomes the leader. In this case, if the leader rotates every round, then the expected cumulative number of votes per process at round r is $E'[S_i^{(r)}] = (r - ct) \cdot Pr[V]/(3f + 1)$. Similar scenarios can be calculated in this manner (note this expectation is usually high to use in practice). The baseline score can also be set lower (i.e., a low increase rate) in certain scenarios: e.g., if processes are allowed to join midway (e.g., in the permissionless systems [3], participants can change dynamically), the baseline score can be used to ensure that the process must successfully consensus a sufficient number of times before it is eligible to be elected as the leader.

Note that ABSE is highly scalable, and the above basic and generic ABSE model is just an instance, which can be modified as needed. For example, a score deduction rule can be introduced to implement targeted penalties for leader misbehavior. Building on our basic model, we can define a deduction factor $\alpha < 0$ or $\gamma < 1$. When process p_i serves as leader in round r but fails to achieve consensus as expected, $X_i^{(r)} - = \alpha$ or $X_i^{(r)} \cdot = \gamma$. Additionally, a "probation period" can be implemented for p_i who will face a temporary multiplier on their baseline requirement, making it harder for them to be elected again immediately after disrupting consensus. Such mechanisms further mitigate the risk of processes that vote validly but lead maliciously. However, it should be noted that even without such mechanisms, the role of ABSE remains evident as it increases the cost for malicious processes to disrupt consensus as leaders (in the original protocols, malicious processes can simply wait for their turn in the rotation without any correct behavior required). In the remainder of this paper, we continue with the basic ABSE model to demonstrate its fundamental effectiveness.

Utilities: The data structures and essential utilities for ABSE are defined in Algorithm 1. The generate utility is utilized to generate a new round of scores (based on the previous round's scores) derived from the replicas in $info$. The Update utility are employed to update score-related data structures, including the score queue $Scores$, the reference score ref_s, and the baseline score $baseline$, when a new round commences. The Judge utility is used to determine if the specified process score surpasses the $baseline$. These utilities are generic for ABSE, but specific implementations can be designed according to particular scenarios.

Communication measurement: Since each process maintains its $Scores$ locally, $Scores$ is not implicated in message transmission, thereby preventing the introduction of extra communication complexity. However, each process does require extra space to store these additional data structures. When processes are corrupted by the adversary, the performance may be significantly affected, as a malicious process that assumes leadership can severely disrupt the consensus in the current round, or even in adjacent rounds. The integration of ABSE effectively mitigates this problem, as it discourages the election of malicious processes based on their historical scores. Under conditions where all processes operate correctly, ABSE's impact on protocol performance is minimal.

Algorithm 1: Data Structures and Basic Utilities for ABSE.

```
Local variables:
    $r \leftarrow 0$  ▷ Record the number of rounds, initially 0
    $s \leftarrow$ array $(n)$  ▷ $s[i]$ - the score of $p_i$ in current $r$, initially 0
    $ref\_s \leftarrow$ array $(n)$  ▷ $ref\_s[i]$ - $p_i$'s reference score, initially 0.
    $Scores \leftarrow queue(ct)$  ▷ A queue of $s$, initially all [0]. Size $ct$ is the number of confirmations required.
    $info \leftarrow \perp$  ▷ Record replicas promoting consensus, initially $\perp$.
    $baseline \leftarrow 0$  ▷ Baseline score of leader election, initially 0.
function generate ()
    $rear \leftarrow Scores.back()$  ▷ Get the rear data in Scores
    On the basis of $rear$, completing the addition of points based on $info$ to obtain the new score array $s\_new$. If $info = null$, then $s\_new \leftarrow rear$.
    $s \leftarrow s\_new$ return
function update( $round$)
    $r \leftarrow round$
    $ref\_s \leftarrow Scores.pop()$
    $Scores.push(s)$
    $baseline \leftarrow$ rule$(r)$  ▷ A new $baseline$ is obtained.
    // RULE is a manually set $r$-based calculation function, and can vary by scenario
    return
function judge ( $j$)
    if $ref\_s[j] \geq baseline$ then return $true$
    return $false$
```

$Scores[]$ and $baseline$ of correct processes cannot be tampered due to fully localization. While malicious replicas can collude to change scores among themselves, this doesn't affect their scores in correct replicas. In protocols where all votes are sent only to the leader, a malicious leader can selectively choose as many votes from faults as possible when forming a quorum certificate. However, only the malicious leader's own $Scores[]$ is affected in this case, as previously discussed. If a malicious process strategize to accumulate scores and become a leader, it must validly participate in multiple rounds of consensus to increase its score in other processes.

In the following sections, we demonstrate the usage and scalability of ABSE by integrating it into two different leader-based BFT protocols that operate under different network assumptions: Chained-HotStuff [8] (partial synchronization) and DAG-Rider [6] (asynchrony).

IV. Preliminaries: HotStuff Protocol

For self-containment, we provide a brief description of HotStuff. Further details can be found in the original literature.

In HotStuff, the leader communicates with the voters on proposal and collects votes from them, which continues

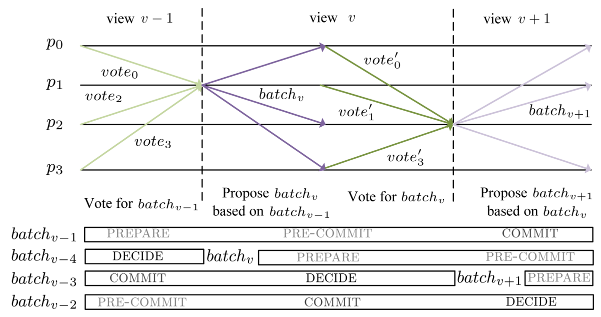

view $v-1$
view v
view $v+1$
p_0
p_1
p_2
p_3
$vote_0$
$vote_2$
$vote_3$
$batch_v$
$vote'_0$
$vote'_1$
$vote'_3$
$batch_{v+1}$
Vote for $batch_{v-1}$
Propose $batch_v$ based on $batch_{v-1}$
Vote for $batch_v$
Propose $batch_{v+1}$ based on $batch_v$
$batch_{v-1}$ PREPARE PRE-COMMIT COMMIT
$batch_{v-4}$ DECIDE $batch_v$ PREPARE PRE-COMMIT
$batch_{v-3}$ COMMIT DECIDE $batch_{v+1}$ PREPARE
$batch_{v-2}$ PRE-COMMIT COMMIT DECIDE

Fig. 2. The flow of Chained-HotStuff ($n = 4$).

three times before processes commits the corresponding proposal. Although this introduces additional phases compared to PBFT [4]-like protocols to guarantee safety and liveness, it maintains a communication pattern limited to one-to-all or all-to-one messaging, effectively reducing communication complexity.

Chained-HotStuff leverages pipelining techniques to improve parallelization. After completion of the first phase of proposal $batch_0$, optimistically assuming $batch_0$ will be finalized, the subsequent leader can immediately start the consensus for $batch_1$ based on this not-yet-finalized proposal. This establishes a chained and pipelined consensus pattern, whereby when $batch_1$ completes its first phase, $batch_0$ completes its second phase simultaneously. This allows for up to 4 proposals at different phases to be advanced in the same consensus round, as illustrated in Fig. 2. The leader election is typically implemented via a deterministic round-robin approach, where each view v has a pre-determined leader based on the view number: $leader(v) = p_{v \bmod n}$. If a leader fails to make progress within a timeout period, processes trigger the NEXTVIEW interrupt to initiate a view change.

Now we present the core components of HotStuff that we'll build upon for our ABSE integration:

1) Data Structures:

A view in HotStuff corresponds to the round we mentioned before, and we use $viewNumber$ or v to record it.

Tree branches and nodes: A tree is a hierarchical structure where each node has one parent, apart from the root node. A branch is a sequence of connected nodes from the root to a leaf node in the tree. Branches conflict if neither one is an extension of the other. A node $node$ contains its parent $node.parent$ and the command (batched transactions) ($node.cmd$). Nodes conflict if the branches led by them conflict.

Messages: A message m consists of fixed fields populated with MSG. Each message has a type ($m.type$) set by the phase and a proposed node ($m.node$). $m.justify$ is an optional field used by the leader to carry the QC and by processes in NEW-VIEW messages to carry the highest $prepareQC$. Messages sent by a process include a partial signature ($m.partialSig$) by the sender, added in the VOTEMSG utility.

Quorum certificates (QC): A QC over a tuple $<$ $type$, $viewNumber, node$ $>$ is a set of signed votes on a node from $2f+1$ processes, implemented using threshold signatures or aggregate signatures.

2) Phases:

Hotstuff protocol employs four phases: PREPARE, PRE-COMMIT, COMMIT, and DECIDE. A new leader begins by collecting NEW-VIEW messages from (n- f) processes. The NEW-VIEW message includes a QC referencing the highest QC ($\perp$ if none) seen so far. In Chained Hotstuff, the four phases merge into a GENERIC phase.

GENERIC phase: During the generic phase, the leader collects $n-f$ generic messages from processes in the previous phase, aggregating them into QC for current phase broadcasting. Processes can perform any necessary operations to maintain the integrity and consistency of the system.

NEXTVIEW interrupt: A process waits for a message in the view for a timeout period in generic phase, determined by an auxiliary NEXTVIEW utility. If NEXTVIEW interrupts waiting, it will increment $viewNumber$ and start the next view.

3) Utility Functions:

MSG $(type, node, qc)$: This function creates a message with given $type$, proposed node $node$, qc and current $viewNumber$.

VOTEMSG $(type, node, qc)$: For the given parameters, this function calls MSG to generate message m and generates a partial signature of the caller stored in $m.partialSig$.

CREATELEAF $(parent, cmd, qc)$: This function creates a new node with a given parent node, a command (or commands) from client and a qc.

QC (V): This function creates a qc from a set of messages V. The qc includes the message type $type$, view number $viewNumber$, and proposed nodes $node$ from the message in V, and a combined signature from all the messages.

MATCHINGMSG (m, t, v): This function checks if a message m has a given type t and view number v. It returns true if the message matches the given type and view number.

MATCHINGQC (qc, t, v): Similar to MATCHINGMSG.

SAFENODE $(node, qc)$: This function checks if a proposed node satisfies the safety or liveness rules. It returns true if the node extends from the $lockedQC$ node, or if the $qc.viewNumber$ is greater than the $lockedQC.viewNumber$.

V. ABSE-CHAINED-HOTSTUFF

To illustrate the integration of ABSE, we first describe how we incorporate its utility into the original HotStuff protocol [8]. The pseudocode for ABSE-Chained-HotStuff is presented in Algorithm 2. We highlight the ABSE related parts in purple.

Before proceeding, it is first necessary to emphasise the challenges addressed in integrating ABSE with Chained-HotStuff, thereby ensuring the subsequent comprehension of the modifications implemented. First, view jumping disrupts the linear view progression model (in the original protocol, processes move sequentially through views with each view having a predetermined leader), requiring careful management of consistency when bypassing multiple views. Second, view jumping necessitates handling multiple leader candidates simultaneously, which adds complexity. Finally, we must ensure view jumping doesn't interfere with Three-Chain formation, which is crucial for command execution in Chained-HotStuff.

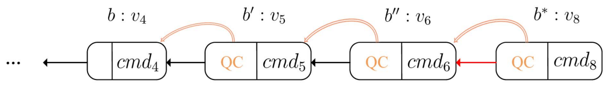
Algorithm 2: ABSE-Chained-HotStuff Protocol.

```
1: Initialization:
     $max\_jmp$    ▷ The upper limit that a view can jump
2: for $curView \leftarrow 1, 2, 3 \cdots$ do
3:     $curView' = curView$
4:     $LC = \{Leader(curView+1), Leader(curView+2), Leader(curView+3), ..., Leader(curView+max\_jmp)\}$
     ▷ GENERIC phase
5:     as a leader do    ▷ $r =$ LEADER ( $curView$)
         ▷ $M$ is the set of messages collected at the end of previous view
6:         $highQC \leftarrow (\arg\max_{m \in M}\{m.justify.viewNumber\}).justify$
7:         if $highQC.viewNumber > genericQC.viewNumber$ then $genericQC \leftarrow highQC$
8:         $curProposal \leftarrow createLeaf(genericQC.node$, client's command, $genericQC)$
9:         broadcast Msg ( $generic$, $curProposal$, $\perp$)    ▷ PREPARE phase
10:    as a process do
11:        generate(), $info \leftarrow null$
12:        update($curView$)
13:        wait for message $m$: MATCHINGMSG ( $m$, $generic$, $curView$) from Leader $curView$
14:        $b^* \leftarrow m.node; b'' \leftarrow b^*.justify.node; b' \leftarrow b''.justify.node; b \leftarrow b'.justify.node$
15:        if SAFENODE( $b^*$, $b^*.justify$) then
16:            $info \leftarrow info \cup leader(curView)$    ▷ Add only 1 point when a leader has both a valid vote and a valid proposal.
17:            for $jmpView \leftarrow 1, 2, 3 \cdots, max\_jmp$ do    ▷ find a reliable process as the next leader within the $max\_jmp$
18:                if judge($Leader(curView + jmpView)) = true$ then
19:                    send $voteMsg(generic, b^*, \perp)$ to $leader(curView + jmpView)$
20:                    $curView \leftarrow curView + jmpView - 1$
21:                    Break
22:        if $b^*.parent = b''$ then $genericQC \leftarrow b^*.justify$    ▷ start PRE-COMMIT phase on $b^*$'s parent
23:        if $(b^*.parent = b'') \wedge (b''.parent = b')$ then $lockedQC \leftarrow b''.justify$    ▷ start COMMIT phase on $b^*$'s
         grandparent
24:        if $(b^*.parent = b'') \wedge (b''.parent = b') \wedge (b'.parent = b)$ then    ▷ start DECIDE phase on $b^*$'s
         great-grandparent
25:            Execute new commands through $b$, respond to clients
26:    as the leader candidates in $LC$ do
27:        wait for all messages: $M \leftarrow \{m | matchingMsg(m, generic, curView)\}$ until there are $(n - f)$ votes:
         $V \leftarrow \{v \mid v.partialSig \neq \perp \wedge v \in M\}$
28:        $genericQC \leftarrow QC(V)$
29:        $curView \leftarrow curView + jmp - 1$, if process is the leader in view $curView + jmp$
30:        $info \leftarrow info \cup V$
     ▷ Finally
31:    NEXTVIEW interrupt: goto this line if NEXTVIEW( $curView'$) is called during "wait for" in any phase
32:    Send MSG GENERIC, $\perp$, $genericQC$ to LEADER ( $curView' + 1$)
33:    $curView \leftarrow curView'$
```

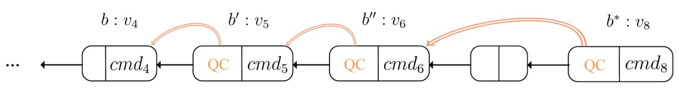
A. The Protocol

HotStuff assumes that each $viewNumber$ has a unique designated leader known to all, enabling processes to predict whether they will be the leaders in subsequent views. The introduction of ABSE can trigger a view jump larger than 1, necessitating the definition of an upper limit on this jump (Line 1). This allows leader processes within this view range to enter the leader candidate queue to await messages from the processes in the current view. Once a process has collected a sufficient number of valid votes, it synthesizes a new $genericQC$ and waits to become the leader of the new view (Line 4, Line 26-29).

All processes need to update $Scores$, ref_s, and $baseline$ every view (Line 11-12). Since each only casts vote to the leader in each round, a process's $Scores[]$ is only updated through votes when it becomes the leader. In addition, we also award additional points to the leader who proposes a valid proposal (Line 16), as it also promote consensus. During the voting phase, the process identifies a reliable process within the scope of the view that can act as the next leader (Line 17-21). It should be noted that there may be instances where a reliable leader cannot be selected from the leaders of $curView + 1$ to $curView + max\_jump$. Such situations will eventually cause the process timer in the HotStuff protocol to time out, triggering a NEXTVIEW interrupt. In this case, the view will proceed according to the original protocol and the ABSE mechanism is suspended (Line 31-33, of course, an alternative approach is possible where the process moves directly to the next view using the original process if a reliable leader is not found until the $curView + max\_jump$, thereby preventing any potential timeouts). The frequency of such instances usually indicates that $baseline$ or max_jump was not chosen appropriately, for example, a lower increase rate of $baseline$ or a higher value of max_jmp may be required. In general, it is appropriate to set max_jmp to $f + 1$.



(a) The nodes at views v_5, v_6, v_8 form a Three-Chain, because v_8 directly references the node of the previous view. Although the view number is increased by more than 1 because of the view jump, there is no dummy node between b'' and b^*, and $b^*.justify.node$ remains equal to $b^*.parent$.



(b) The nodes at views v_5, v_6, v_8 do not form a Three-Chain because the node at view v_8 does not make a valid One-Chain. The reason is that the leader of a preceding view fails to obtain a valid QC, either because there are conflicting proposals, or due to a benign crash. This creates a dummy node between b'' and b^*, resulting in $b^*.justify.node$ being unequal to $b^*.parent$.

Fig. 3. View jumping does not interfere with the formation of a Three-Chain.

In Chained-HotStuff, when a node b^* carries a QC that refers to a direct parent, i.e., $b^*.justify.node = b^*.parent$, we say that it forms a One-Chain. Denote by $b'' = b^*.justify.node$. Node b^* forms a Two-Chain, if in addition to forming a One-Chain, $b''.justify.node = b''.parent$. It forms a Three-Chain, if b'' forms a Two-Chain. The formation of a Three-Chain is crucial for the execution of commands (Line 24). As shown in Fig. 3(a), view jumping does not interfere with the formation of a Three-Chain as it does not lead to the creation of dummy nodes. For comparison, the scenario where dummy nodes interfere with the formation of a three-chain is shown in Fig. 3(b).

The following step is to demonstrate consistency in all three scenarios laid out in Section III-B when ABSE serves as Chained-HotStuff's new leader election mechanism.

Lemma V.1: In ABSE-Chained-HotStuff, if at least $2f + 1$ processes agree on the leader of the next view and the remaining processes try to make a view jump, in which case all processes are able to proceed to the next view normally.

Proof: If at least $2f + 1$ processes have voted for the next view leader, their votes can be combined into a valid $genericQC$, enabling the system to transition to the next view. Since at most f replicas are malicious, it is impossible for a leader candidate in the subsequent view to be able to collect $2f + 1$ votes as well to form a valid $genericQC$, as this would require one of the correct replicas to vote for two different leader candidates simultaneously, which is not possible.

Any leader candidates in the subsequent views acknowledged by the remaining processes but not receiving sufficient votes to initiate proposing will not be elected. The remaining processes eventually experience a timeout, leading to a NEXTVIEW interrupt, thus proceeding to the next view. ■

Lemma V.2: In ABSE-Chained-HotStuff, if at least $2f + 1$ processes consider the next view leader unreliable and unanimously agree on a new leader of the subsequent view, in this case all processes are able to complete the view jumping successfully.

Proof: If at least $2f + 1$ processes consider the next view leader to be unreliable and unanimously agree on a new leader of the subsequent view, then this new leader can gather sufficient votes to form a valid $genericQC$ and proceed to the corresponding view. Similarly to Lemma V.1, it is impossible for another leader candidate in LC to collect $2f + 1$ votes as well. The remaining processes synchronize with the most recent progress by obtaining subsequent messages from higher views when the view jump occurs. ■

Lemma V.3: In ABSE-Chained-HotStuff, if no more than $2f + 1$ processes agree on the same leader, in this case all processes proceed to the next view normally.

Proof: If no more than $2f + 1$ processes agree on the same leader, then no leader will be able to gather sufficient votes to form a valid $genericQC$, resulting in no new proposals being proposed, leaving a dummy node. In this case, all processes will timeout, triggering the NEXTVIEW interrupt, which then synchronizes to the next view. ■

Due to the Optimistically Responsive of HotStuff, the leader only needs to gather $2f + 1$ votes to reach consensus. Therefore, the protocol's performance is not affected by the downtime or timer timeout of up to f processes (situations described in Lemma V.1 and V.2). Processes failing to agree on a same leader (situations described in Lemma V.3, the occurrence indicates the parameters $baseline$ or max_jmp is not justified) or malicious processes becoming leaders can result in the emergence of dummy nodes, adversely affecting the protocol's performance. The introduction of ABSE can reduce the occurrence of faults becoming leaders, thus enhancing the performance when dealing with malicious processes. Our protocol maintains the same communication complexity as HotStuff. During each view, the leader transmits messages to all processes. Processes, in turn, certify the vote with partial signatures. The leader's message contains a proof of the $n - f$ votes previously collected, encoded by a single threshold signature. The sole authenticator of a process's response is its unique partial signature. As a result, each view receives $O(n)$ authenticators. The overall complexity per view is $O(n)$.

We show that ABSE-Chained-HotStuff satisfies the Safety and Liveness properties (as defined in Section III-A) in Appendix A of the supplementary material.

VI. Preliminaries: DAG-Rider Protocol

Similarly, before integrating ABSE, it is first necessary to provide a brief description of DAG-Rider for self-containment.

DAG-Rider is an asynchronous BAB protocol that operates in two distinct layers. In the communication layer, processes reliably broadcast proposals to form a structured DAG of messages. In the ordering layer, processes locally interpret their DAGs to total-order all proposals without additional communication. The protocol organizes rounds into waves, with each wave consisting of four consecutive rounds. During each round, every process broadcasts a vertex containing a batch of transactions

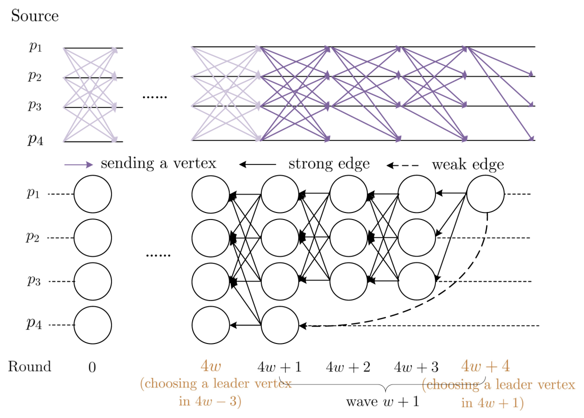


Fig. 4. The flow of DAG-Rider ($n = 4$).

and references to previous vertices. Fig. 4 depicts the DAG structure and wave organization in DAG-Rider. Each horizontal line represents vertices from a single process, while vertical columns represent rounds. The strong edges connect vertices to at least $2f + 1$ vertices from the previous round, while weak edges ensure all vertices are eventually included in the causal history.

A key distinction of DAG-Rider is its leader election mechanism, which uses a global perfect coin (detailed later within this section) to randomly select a leader for each wave retrospectively after wave completion. Now we present the core components of DAG-Rider that we'll build upon for our ABSE integration:

1) Building Blocks:

Reliable broadcast: Each sender process p_k can send messages by calling $r\_bcast_k(m, r)$, where m is a message, $r \in \mathbb{N}$ is a round number. Every process p_i has an output $r\_deliver_i(m, r, p_k)$, where p_k is the process that called the corresponding $r\_bcast_k(m, r)$.

BAB: Each correct process $p_i \in \Pi$ can call $a\_bcast_i(m, r)$ and output $a\_deliver_i(m, r, p_k)$, $p_k \in \Pi$.

Global perfect coin: An instance $w, w \in \mathbb{N}$, of the coin is invoked by a process $p_i \in \Pi$ by calling $choose\_leader_i(w)$. This call returns a process $p_j \in \Pi$, which is the chosen leader for w. The global perfect coin provides several guarantees. First, it ensures agreement between correct processes that call $choose\_leader_i(w)$ and $choose\_leader_j(w)$, such that the return values p_1 and p_2 are equal. Second, it guarantees termination of the $choose\_leader(w)$ call when at least $f + 1$ correct processes make the call. Third, the return value of the $choose\_leader(w)$ call is indistinguishable from a random value (unpredictability), except with negligible probability ε, as long as less than $f + 1$ processes make the call. Finally, the coin is fair. For any w and any p_j in the system, the probability that p_j is returned by the coin is equal to $1/n$ (fairness). Global perfect coin ensures the liveness of DAG-Rider.

2) Data Structures:

Vertex: A vertex v represents a vertex in the DAG, contains the round number of v in the DAG ($v.round$), the process id that broadcast v ($v.source$), a batch of transactions ($v.block$), strong edges pointing to at least $2f + 1$ vertices in $v.round - 1$ ($v.strongEdges$) and weak edges pointing to vertices in previous rounds with no path from v ($v.weakEdges$).

DAG: $DAG[r]$ contains the set of all vertices in round r that the process currently sees. It is initialized as $DAG[0]$ containing $2f + 1$ "genesis" vertices. $\forall r > 0$, $DAG[r]$ is initially empty.

3) Basic Utilities:

$path(v, u)$: Checks if there are paths from vertex v to vertex u in the DAG consisting of strong and weak edges and returns all such paths.

$strong\_path(v, u)$: Checks if there are paths from vertex v to vertex u in the DAG consisting only of strong edges and returns all such paths.

$create\_new\_vertex(r)$: Creates a new vertex for round r containing a block of transactions, set strong edges to vertices in $r - 1$, and calls set_weak_edges to set weak edges.

$set\_weak\_edges(v, r)$: Sets weak edges from vertex v in round r to vertices in previous rounds $r' < r - 1$ such that there is no path from v to those vertices. This ensures all vertices are eventually included to satisfy BAB validity.

VII. ABSE-DAG-RIDER

ABSE allows reliable processes in DAG-Rider [6] to have more influence, while reducing the potential impact of malicious processes and enhancing effective throughput of the system. The randomness and fairness of the original protocol are retained. Moreover, we prove that ABSE does not cause any additional inconsistencies and maintains the communication complexity of the original protocol. Based on these results, we verify the correctness of the ABSE-DAG-Rider. The pseudocode for this protocol is provided in Algorithm 3.

Similarly, we first emphasise the unique challenges addressed in integrating ABSE with DAG-Rider. First, unlike in Chained-HotStuff where leaders actively coordinate consensus, DAG-Rider leaders primarily serve as ordering anchors, requiring adaptation of the scoring mechanism. Second, the asynchronous model complicates reliability assessment, requiring carefully designed scoring criteria based on strong path relationships as there is no constraint on message delivery time. Third, the wave-based approach with retrospective leader selection necessitates special handling for unreliable leaders, allowing processes to efficiently bypass waves where problematic leaders would otherwise slow consensus progress.

A. The Protocol

DAG-Rider organizes each wave into four rounds, referred to as r_1, r_2, r_3, r_4. During each round, every process create a new vertex in DAG to transmit a proposal to all other processes, containing the current round's batched transactions, along with at least $2f + 1$ strong edges to the prior round vertices and weak edges to vertices in rounds $r' < r - 1$. A process switches to the following round once $DAG_i[r]$ has at least $2f + 1$ vertices. This four-round process is repeated until all correct processes finish r_4, signifying the completion of a wave. Then, all processes use the global perfect coin to select a random leader of this wave, who corresponds to the proposal of a correct process in the first round (r_1). A leader is considered committed if it has at least $2f + 1$ references (strong edges) to it from r_4. Once committed, the causal history contained

Algorithm 3: ABSE-DAG-Rider, Pseudocode for Process p_i.

```
1:  Local variables:
        $r \leftarrow 0,\ buffer \leftarrow \{\},\ decidedWave \leftarrow 0,\ deliveredVertices \leftarrow \{\}$
        $leadersStack \leftarrow$ initialize empty stack with $isEmpty(), push()$, and $pop()$ functions
2:  procedure $choose\_leader'(w)$
3:    if at least $2f+1$ processes call $choose\_leader'(w)$ then return $true$
4:    return $false$
5:  procedure $get\_wave\_vertex\_leader(w)$
6:      $p_j \leftarrow choose\_leader_i(w)$
7:    if judge$(p_j) = true$ then $boolean_i \leftarrow choose\_leader'_i(w)$
8:      if $boolean_i = true \wedge \exists v \in DAG[round(w,1)] : v.source = p_j$ then return $v$
9:    return $\perp$
10: procedure $order\_vertices(leadersStack)$
11:   while $\neg leadersStack.isEmpty()$ do
12:       $v \leftarrow leadersStack.pop()$
13:       $verticesToDeliver \leftarrow v' \in \bigcup_{r>0} DAG[r] | path(v, v') \wedge v' \notin deliveredVertices$
14:     for $v' \in verticesToDeliver$ in some deterministic order do
15:       output $a\_deliver_i(v'.block, v'.round, v'.source)$
16:         $deliveredVertices \leftarrow deliveredVertices \cup \{v'\}$
      ▷ DAG Construction
17: while true do
18:   for $v \in buffer : v.round \leq r$ do
19:     if $\forall v' \in v.strongEdges \cup v.weakEdges : v' \in \bigcup_{k \geq 1} DAG[k]$ then
20:         $DAG[v.round] \leftarrow DAG[v.round] \cup \{v\},\ buffer \leftarrow buffer \backslash \{v\}$
21:   if $|DAG[r]| \geq 2f+1$ then
22:     if $r \bmod 4 = 0$ then $wave\_ready(r/4)$  ▷ A new wave is complete
23:       $r \leftarrow r+1$
24:       generate(), $info \leftarrow null$
25:       update$(r)$, $v \leftarrow create\_new\_vertex(r)$
26:       $r\_bcast_i(v, r)$
27: upon $r\_deliver_i(v, round, p_k)$
28:     $v.source \leftarrow p_k,\ v.round \leftarrow round$
29:   if $|v.strongEdges| \geq 2f+1$ then $buffer \leftarrow buffer \cup \{v\}$
    ▷ ABSE-DAG-Rider, DAG-based asynchronous BAB protocol with ABSE
30: upon $a\_bcast_i(b, r)$
31:     $blocksToPropose.enqueue(b)$    ▷ Correct processes call this with sequential $r$, starting at 1
32: upon $wave\_ready(w)$
33:     $v \leftarrow get\_wave\_vertex\_leader(w)$
34:   if $v = \perp \vee |\{v' \in DAG[round(w,4)] : strong\_path(v', v)\}| < 2f+1$ then return    ▷ No commit
35:     $info \leftarrow \{v' \in DAG[round(w,4)] : strong\_path(v', v)\}$
36:     $leadersStack.push(v)$
37:   for wave $w'$ from $w-1$ down to $decidedWave+1$ do $v' \leftarrow get\_wave\_vertex\_leader(w')$
38:     if $v' \neq \perp \wedge strong\_path(v, v')$ then
39:         $info \leftarrow info + \{v'' \in DAG[round(w',4)] : strong\_path(v'', v')\}$ ▷ Processes that appear multiple times in
          $info$will receive additional scores based on the number of occurrences
40:         $leadersStack.push(v')$
41:         $v \leftarrow v'$
42:     $decidedWave \leftarrow w$
43:     $order\_vertices(leadersStack)$
```

by the leader is consistently sorted and output by all correct processes.

Unlike ABSE-Chained-HotStuff, we don't need to set a maximum jump round because the role of the leader in DAG-Rider differs from that of the regular protocol, as they are no longer accountable for proposing and coordinating other processes to vote on the proposals. In DAG-Rider, the function of the leader is to act as a marker at the ordering level. They serve as the cut-off point of the ordering, representing a topological point from a graph-theoretic perspective. Each process designates its own leader, who only decides the ordering of its own causal history. Their influence on other processes is indirect.

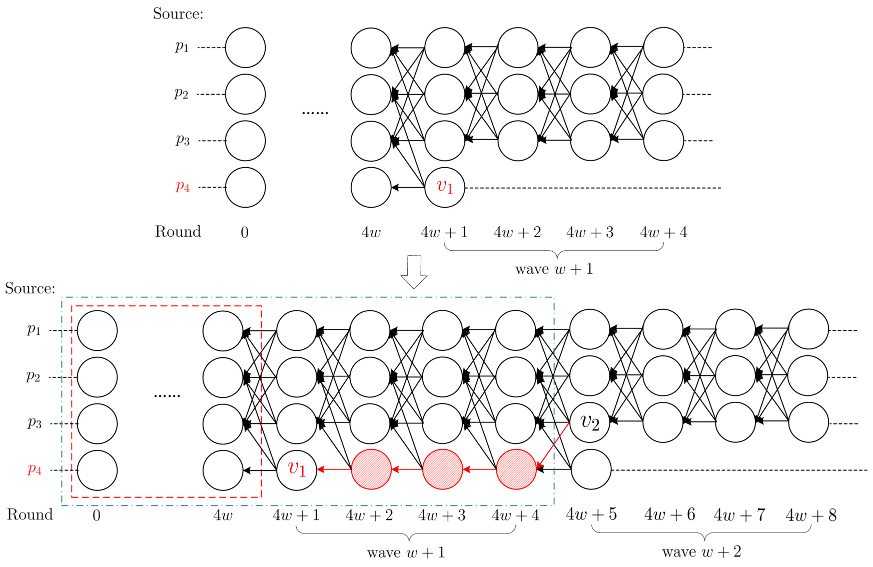


Fig. 5. Illustration of DAG_1, i.e., the DAG at process 1, out of 4 processes in total. Assume p_4 is a process corrupted by an adversary that broadcasts vertices or tries to make obfuscated proposals at a rate that lags behind other processes in each wave. At the end of wave $w+1$, v_1 is elected as leader. The commit rule is not met for v_1 since there are less than $2f+1$ vertices in round $4w+4$ with a strong path to v_1. However, if p_4 promptly releases the vertices of rounds $4w+2$ to $4w+4$ and makes any of them referenced by the leader vertex of the next wave (e.g., v_2), then commit rule is met in wave $w+2$ since there are $2f+1$ vertices in round $4w+8$ with a strong path to v_2 and there is a strong path from v_2 to v_1 (highlighted in red). In DAG-Rider v_1 at this point succeeds in getting elected before v_2, which causes the $order\_vertices$ function to first traverse the vertices in the red dashed box for ordering based on v_1 as the cut-off point, and subsequently traverse the vertices in the blue dotted box to order them based on the previous ordering. This is also the minimum cost method for malicious processes to be elected in DAG-Rider. ABSE-DAG-Rider mitigates this by excluding the vertex of an unreliable process as a leader in wave $w+1$, with only the vertices in the blue box traversed for ordering in wave $w+2$.

DAG-Rider enables all processes to randomly determine a globally consistent leader under a specific wave through global perfect coin. A process can only know the leader of wave w if at least $f+1$ processes have called $choose\_leader(w)$ in w. We use this building block to construct a similar procedure $choose\_leader(w)'$, which returns $true$ only when $2f+1$ ($f+1$ correct) processes have called $choose\_leader(w)'$ in w (Line 2-4, it is actually simpler to implement than $choose\_leader(w)$, e.g. by broadcast & validation). This procedure helps to achieve a globally consistent agreement on whether the current leader need to be skipped.

Specifically, a leader can only be successfully elected if $2f+1$ processes consider the leader of w reliable and there exists a vertex v' of the leader in r_1. If v' meets the commit condition, each process subsequently assigns scores to the source process that has to v' strong edges in r_4 according to its own DAG view (Line 7-8, Line 24-25, Line 35, Line 39). A leader p_j that is deemed unreliable will have its wave temporarily skipped until the next reliable leader is elected, at which point p_j will have one last chance to be judged as reliable, and if it is still unreliable, then the functions for which it is the cut-off point of the ordering will all be delivered to the reliable leader (Line 37-43). This approach improves the effective throughput because, at the very least, in DAG-Rider, if an adversary is down or trying to be bad, the ability of the ABSE to directly skip its ordering round would save a lot of extra judgement.

In addition, ABSE can also deal with special adversary patterns: in DAG-Rider, if the next valid leader vertex exists a strong edge to the unreliable leader vertex p_j (Line 38-40), then p_j can be elected and it takes at least two rounds of loops to fully order the vertices within the $order\_vertices$ procedure (Line 11-12), one of which is based on the ordering of p_j. ABSE-DAG-Rider avoids this issue. On the other hand, each round of ordering with $order\_vertices$ requires traversal of all vertices, from round 0 to the current round (Line 13), which greatly affect the efficiency when vertices are huge in number. Therefore, reducing the rounds of ordering can also optimize the performance, as illustrated in Fig. 5.

It is worth noting that the randomness and fairness of the original protocol was preserved after the implementation of ABSE. This is due to the fact that even if there is a skip in leader during wave w, processes cannot determine the leader of wave $w+1$ until at least $f+1$ processes call $choose\_leader(w+1)$, which matches the original protocol. The following step is to demonstrate that ABSE does not introduce any further inconsistencies.

Lemma VII.1. In ABSE-DAG-Rider, if no more than $f+1$ correct processes deem the leader of a wave to be reliable, then all correct processes will skip the leader.

Proof: If no more than $f+1$ correct processes deem a leader to be reliable, the $get\_wave\_vertex\_leader$ procedure will return null for all processes. Based on the code, all processes will refrain from committing any proposals and proceed to the next round. ■

Lemma VII.1 covers both the second and third cases referenced in Section III-B. ABSE-DAG-Rider maintain consistency behavior for these cases.

Lemma VII.2: In ABSE-DAG-Rider, a wave w leader follows the election process in DAG-Rider if at least $f+1$ correct processes consider it reliable.

Proof: If at least $f+1$ correct processes consider the leader of w reliable, then the $choose\_leader'$ returns $true$ for all processes. If the leader has vertex v_1 in the $DAG[round(w,1)]$ and there are $2f+1$ strong edges to v_1 in $DAG[round(w,4)]$ or there is a strong edge to v_1 of a reliable leader vertex in the subsequent round, v_1 is pushed into $leadersStack$ to wait for ordering. This is identical to that in DAG-Rider. ■

Since each process maintains its $Scores$ locally, thereby preventing introducing extra communication complexity. Using Cachin and Tesero's [53] information dispersal protocol instantiation calculates the DAG's bit complexity by dividing the bits sent by correct processes per round by the total ordered values. The complexity is $O(n^2\log(n)+nM)$, where M is the message size, including a set of proposed values and n references with a process id of size $\log(n)$. If each message batches $n\log(n)$ values, the bit complexity of a broadcast is $O(n^2\log(n))$. Each process can broadcast once per round, limiting a correct process to n reliable broadcasts, hence the total bit complexity per round is $O(n^3\log(n))$. A minimum of $2f+1=O(n)$ vertices are ordered each round, ordering $O(n^2\log(n))$ values. Therefore, the amortized communication complexity is $O(n)$, the same as the original protocol.

We show that ABSE-DAG-Rider satisfies the properties of the BAB problem (as defined in Section III-A) in Appendix B of the supplementary material.

VIII. Evaluation

Overview: We implemented ABSE-Chained-HotStuff and ABSE-DAG-Rider in about 4 K and 2.5 K lines of Rust code respectively, with the ABSE module contributing around 240 lines to each. We utilize Ed25519 and BLAKE3 for cryptographic tasks such as digital signatures, and TCP connections for inter-process communication via the Tokio library. The client, operating as an independent single-threaded process, dispatches transactions to processes at a tunable frequency. Configuration is retrieved from JSON files and command line arguments. We developed modules namely, CONFIG-GEN and GENERATE, to automate the generation of configuration files for diverse experiments. The codebase is hosted on GitHub repositories to ensure reproducibility.[3, 4]

Our evaluation emphasizes latency and throughput as primary performance indicators. Latency is calculated from the time a client initiated a transaction to the time it was committed. Throughput is the rate of committed transactions per second. Additional metrics such as memory usage, and the percentage of malicious leaders are also considered. Our evaluations reveal that the ABSE-enhanced versions effectively mitigate the effects of malicious leaders with negligible resource overhead, leaving the original protocol's baseline performance virtually unaffected.[5]

This section evaluates ABSE-Chained-HotStuff with Chained-HotStuff regarding performance under malicious processes, base performance & scalability, and the impact of various baselines, illustrating ABSE's advantages. We also executed similar experiments for ABSE-DAG-Rider and DAG-Rider. Due to space constraints, they are not included in this primary text for the following reasons: the existing open source code of HotStuff has been tested over time, whereas there is no reliable open source code for DAG-Rider so far; HotStuff exhibits stable performance, whereas DAG-Rider suffers from unstable performance as the protocol advances; the existing open-source code for the DAG protocol does not fully implement the Global Perfect Coin, necessitating alternative methods for simulating $choose\_leader(w)'$. The evaluation of ABSE-DAG-Rider is included in Appendix C, available online in the supplementary material.

Setup: We conducted the experiments in this section on a cluster of up to 16 Aliyun's Cloud Elastic Compute Service (ECS) u1-c1m2.4xlarge instances, each with 16 vCPUs and 32 GB of memory, running Ubuntu 22.04 LTS and deployed rustc 1.67.0-nightly. All CPU cores are supported by Intel(R) Xeon(R) Platinumc. We use n to represent the network size, and use f to represent the number of malicious processes. All processes are evenly distributed across instances and connect others to build a consensus network. The labels CH, and ACH represent the experimental results for Chained-HotStuff, ABSE-Chained-HotStuff respectively.

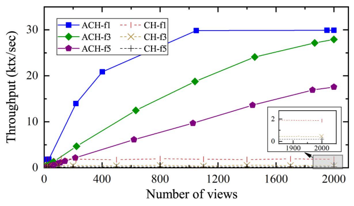
Performance under malicious processes.[6] We studied the systems' performance variations under malicious processes as the consensus advances (view grows). Our experimental parameters were $n=16$, f ranging from 1 to 5 (notated as "f1", "f3", "f5"), a batch size of 400, a payload size of 128 bytes, network delay of 0 ms, and a timeout of 1000 ms. The client sends packets to each process at a rate of 2000 transactions per second (note that the client sending rate for this experiment is fixed and does not bring the system to its performance limit under present parameter settings, which is different from the rate set in the subsequent experiments). The baseline score was set at 1/2 expectation, with a maximum view jump of 6. These malicious processes were simulated by enabling certain processes to intentionally discard all messages and disregard any requests. Figs. 6 and 7 illustrate the system's (average) performance fluctuations under malicious processes for the first 2000 views. In Fig. 6, malicious processes are concentrated, while in Fig. 7, they are uniformly distributed. The former scenario is more detrimental in leader-based BFT systems as it results in consecutive timer timeouts.

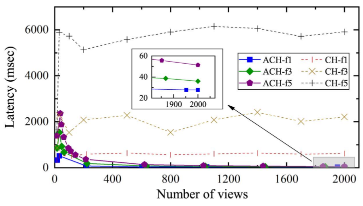
[3] https://github.com/BerserkRugal/ABSE-Chained-HotStuff

[4] https://github.com/BerserkRugal/ABSE-DAG-Rider

[5] We compare ABSE-integrated protocols with their original counterparts rather than across different protocol families. This approach isolates ABSE's specific impact on leader election quality by keeping other protocol variables constant. We selected Chained-HotStuff and DAG-Rider as representative protocols because they form the foundation for many state-of-the-art leader-based BFT systems in partially synchronous and asynchronous settings, respectively. By demonstrating ABSE's effectiveness on these foundational protocols, we establish its potential adaptability across different protocol families (thus illustrating its generality). While comparing with other reputation mechanisms would be valuable, practical limitations including protocol-specific dependencies (i.e., designed for specific protocol types or network assumptions and often cannot be adapted to both Chained-HotStuff and DAG-Rider simultaneously), implementation availability (i.e., lack of open source implementations), and complementary rather than competitive designs (to ABSE) make such comparisons challenging for this study.

[6] Since we use a basic and generic ABSE module in the present algorithms, more complex adversary capabilities are not simulated here, e.g., vote validly but lead maliciously, which requires the addition of score deduction mechanisms for the leader after a failed consensus or an increase in the score weight of leader's valid proposal, as discussed in Section III-B. Nevertheless, the ultimate anticipated effectiveness of ABSE are largely analogous.





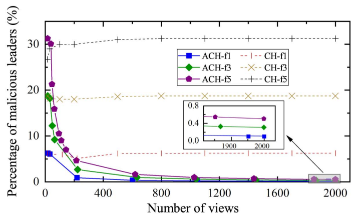
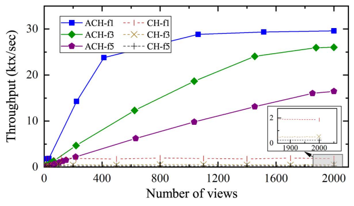
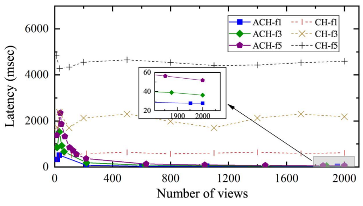


Fig. 6. Variations in system performance under concentrated faults distribution in the first 2 k views. 128-byte payload, 400 batch size, 0 ms network delay and 2 ktx/s injection rate per process. The subfigures from left to right indicate throughput, latency, percentage of malicious leaders (frequency of malicious leader elections).





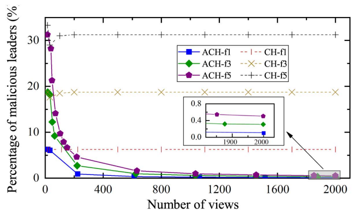


Fig. 7. Variations in system performance under uniform faults distribution in the first 2 k views. 128-byte payload, 400 batch size, 0 ms network delay and 2 ktx/s injection rate per process. The subfigures from left to right indicate throughput, latency, percentage of malicious leaders (frequency of malicious leader elections).

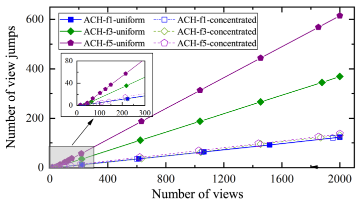


Fig. 8. Number of view jump changes in the first 2000 views for various malicious processes distributions.

We observe that ABSE-Chained-HotStuff's performance initially mirrors the original protocol's. However, as the consensus progresses, the ABSE module enables processes to gradually identify and label the malicious ones. Consequently, the proportion of malicious processes being elected as a leader diminishes, leading to a significant surge in overall system throughput and a corresponding reduction in latency. For ABSE-Chained-HotStuff, assuming all other factors remain constant, a concentrated distribution of malicious processes (denoted as "concentrated", with the corresponding uniform distribution referred to as "uniform") results in enhanced throughput and reduced latency. This outcome can be attributed to the reduced number of view jumps, as multiple consecutive malicious processes can be bypassed in a single action, as demonstrated in Fig. 8. However, this effect is minute and can be considered negligible for small systems.

Base Performance and Scalability (fault-free cases): We examined the base performance and scalability of the systems under various transaction sizes, system sizes, network delays, and batch sizes. The first experiment was conducted with $n = 4, 16, 64, 256$, a payload size of either 128 bytes or 1024 bytes (denoted as "p128", "p1024"), a batch size of 400, a network latency of 0 ms, and a timeout of 1000 ms. The second experiment was conducted with batch sizes of 400, 800, 1200, 1600, network latencies of 10 ms or 50 ms (denoted as "d10", "d50", we use the traffic control tool tc in Linux to simulate network delay), $n = 16$, a payload size of 128 bytes, and a timeout of 1000 ms. Throughput and latency data were obtained by adjusting the client sending rate from underload to overload conditions and recording the data at the inflection points. The results of these experiments are displayed in Figs. 9 and 10, respectively. Note that when not explicitly stated, the performance is measured during the systems' steady state rather than measured within a specific view range subsequent to the initiation of the protocol.

On the whole, ABSE-Chained-HotStuff exhibits minimal performance degradation (nearly identical performance) compared to the original protocol base, with ABSE module only consuming minimal extra memory. This demonstrates that ABSE does not add significant overhead when there are no faults. Specifically, since all extra data structures of the ABSE module are stored locally by each process and not communicated among them, varying batch sizes, network latencies, and payload sizes have negligible effects on the impact of ABSE introduction on the original protocol's throughput, latency, and memory consumption. However, an increase in system size does cause the latency gap between ABSE-Chained-HotStuff and the original protocol to gradually widen - as more processes need to be scored - but this gap remains relatively small even at 256 processes. The extra memory usage introduced is negligible at larger system scales, and this holds true for batch size as well.

Impact of various baselines: We explored how different baselines affect system performance under various conditions. The experiment parameters were set with $n = 16$, a baseline ranging from 0.5 to 1.25 times the expected value ($(r-2) \cdot Pr[V]/(3f+1)$, as calculated in Section III-B), a batch size of 400, a payload size of 128 bytes, and a timeout of 1000 ms. The results are presented in Fig. 11. In general, **the baseline must not surpass the expected value**; otherwise, the process will be unable to keep pace with the baseline increase, even if it earns a score in every round. This situation would prevent the system from electing a reliable leader within the maximum jump range, leading to most rounds timing out, severely degrading system performance.

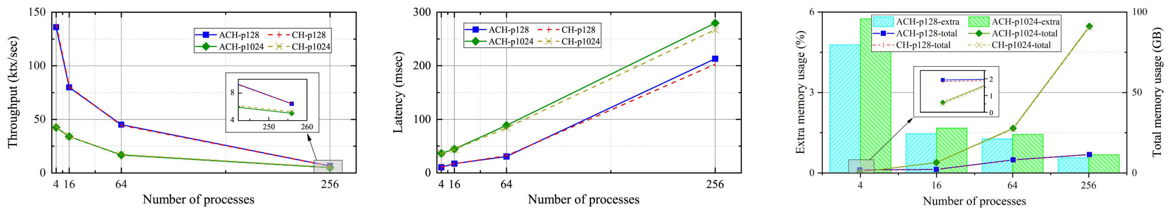


Fig. 9. System performance comparison under various system sizes and payload sizes with 400 batch size and 0 ms network delay. The subfigures from left to right indicate throughput, latency, and memory usage.

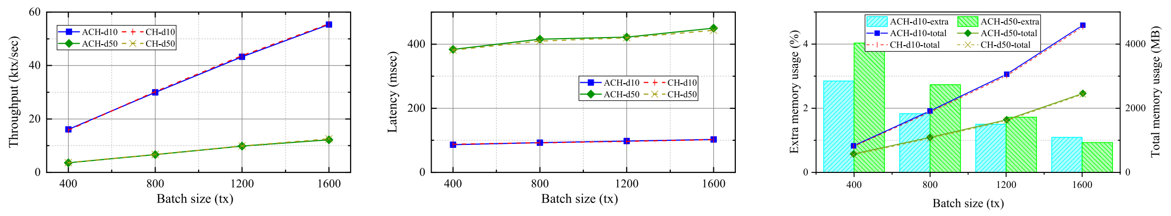


Fig. 10. System performance comparison under different batch sizes and network delays with 16 processes and 128-byte payload. The subfigures from left to right indicate throughput, latency, and memory usage.

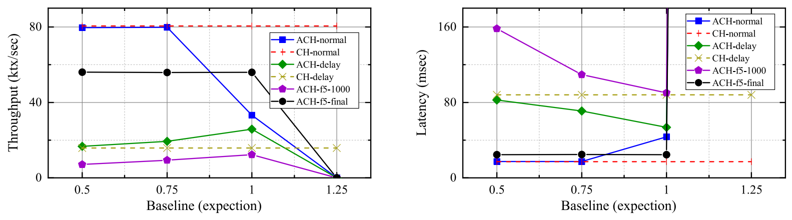


Fig. 11. Impact of various baselines on system throughput (left) and latency (right) under different conditions: normal situation, presence of faults, delay differences among processes with 16 processes, 128-byte payload, and 400 batch size.

When the baseline score is within the expected value, under normal circumstances (denoted as "normal"), an excessively high baseline score can be problematic (throughput falls below what it would be without ABSE usually indicates too high a baseline). It may cause some rounds where processes cannot agree on the same leader, thereby triggering a timeout and affecting system performance (since not all processes can score per round). However, this is not the case when there is a delay difference among processes (denoted as "delay"). We manually added a delay of 1–16 ms to each of the 16 processes. Interestingly, a slightly higher baseline resulted in improved performance, as higher-delay processes are identified as "unreliable" during consensus and excluded from the leader election. As a result, processes with lower delay are usually elected as leaders. Furthermore, a higher baseline score is also appropriate when dealing with malicious processes. We conducted an experiment with $f = 5$, finding that although the chosen baseline scores had almost no impact on the system's final performance (denoted as "f5-final"), higher baseline scores enabled the system to detect malicious processes more quickly in the early stages of consensus. This effect is well illustrated by the system's performance at the 1000th view (denoted as "f5-1000").

IX. Conclusion

We explored ABSE, an Adaptive Baseline Score-based Election approach, aimed at reducing the adverse impact of malicious leaders in leader-based BFT systems. ABSE focuses on accumulating scores for processes based on their consensus contribution, strategically bypassing less reliable participants during leader election. We highlighted the implementation challenges and provided a formal description of ABSE, defining its generic components and adherence rules to ensure global consistency. We apply ABSE to two different BFT protocols, demonstrating its scalability and negligible impact on protocol complexity. Our results indicate that ABSE effectively mitigates the effects of malicious leaders with minimal resource overhead in systems deploying up to 16 servers and 256 processes, leaving protocols' base performance largely intact. We believe ABSE is a powerful and practical approach, and will be applicable to a diverse range of BFT systems due to its high scalability to customize the scoring mechanism and baseline score for specific scenes and needs.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Concurrency: The Works of Leslie Lamport*. San Rafael, CA, USA: Morgan & Claypool, 2019, pp. 203–226.
[2] A. Segall, "Distributed network protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 1, pp. 23–35, Jan. 1983.
[3] X. Wang, S. Duan, J. Clavin, and H. Zhang, "BFT in blockchains: From protocols to use cases," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–37, 2022.
[4] M. Castro et al., "Practical Byzantine fault tolerance," in *Proc. Symp. Operating Syst. Des. Implementation*, 1999, pp. 173–186.
[5] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, Univ. Guelph, Guelph, ON, Canada, 2016.

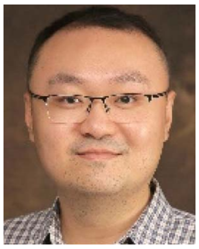
[6] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proc. 2021 ACM Symp. Princ. Distrib. Comput.*, 2021, pp. 165–175.
[7] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient bft consensus," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 34–50.
[8] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 347–356.
[9] X. Liu et al., "Dolphin: Efficient non-blocking consensus via concurrent block generation," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 11824–11838, Dec. 2024.
[10] S. Duan and H. Zhang, "Foundations of dynamic BFT," in *Proc. 2022 IEEE Symp. Secur. Privacy*, 2022, pp. 1317–1334.
[11] X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase BFT with linearity," in *Proc. 52nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2022, pp. 54–66.
[12] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag BFT protocols made practical," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 2705–2718.
[13] J. Abdella, Z. Tari, R. Mahmud, N. Sohrabi, A. Anwar, and A. Mahmood, "HiCoOB: Hierarchical concurrent optimistic blockchain consensus protocol for peer-to-peer energy trading systems," *IEEE Trans. Smart Grid*, vol. 14, no. 5, pp. 3927–3943, Sep. 2023.
[14] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance Byzantine fault tolerant settlement," in *Proc. 2nd ACM Conf. Adv. Financial Technol.*, 2020, pp. 163–177.
[15] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for Byzantine consensus," in *Proc. 40th Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, Springer, Aug. 17–, 2020, pp. 451–480.
[16] M. Lewis, *Flash Boys: A Wall Street Revolt*. New York, NY, USA: WW Norton & Company, 2014.
[17] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without Byzantine oligarchy," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 633–649.
[18] C. Cachin, J. Mićić, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, Springer, 2022, pp. 316–333.
[19] G. Zhang and H.-A. Jacobsen, "Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 52–63.
[20] S. Cohen et al., "Be aware of your leaders," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, Springer, 2022, pp. 279–295.
[21] A. Spiegelman, B. Aurn, R. Gelashvili, and Z. Li, "Shoal: Improving DAG-BFT latency and robustness," 2023, arXiv: *2306.03058*.
[22] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *Proc. 2020 IEEE Symp. Secur. Privacy*, 2020, pp. 106–118.
[23] P. Feldman and S. Micali, "An optimal probabilistic protocol for synchronous Byzantine agreement," *SIAM J. Comput.*, vol. 26, no. 4, pp. 873–933, 1997.
[24] Z. Zhang et al., "HCA: Hashchain-based consensus acceleration via re-voting," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 775–788, Mar./Apr. 2024.
[25] G. Bracha, "Asynchronous Byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, 1987.
[26] X. Dai, B. Zhang, H. Jin, and L. Ren, "ParBFT: Faster asynchronous BFT consensus with a parallel optimistic path," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2023, pp. 504–518.
[27] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in Byzantine consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2023, pp. 475–489.
[28] N. Alhaddad et al., "Balanced Byzantine reliable broadcast with near-optimal communication and improved computation," in *Proc. 2022 ACM Symp. Princ. Distrib. Comput.*, 2022, pp. 399–417.
[29] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "DispersedLedger : High-Throughput Byzantine consensus on variable bandwidth networks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 493–512.
[30] I. Kaklamanis, L. Yang, and M. Alizadeh, "Poster: Coded broadcast for scalable leader-based BFT consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 3375–3377.
[31] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A flexible high performance MPI," in *Proc. 6th Int. Conf. Parallel Process. Appl. Math.*, Poznań, Poland, Springer, Sep. 11–14, 2006, pp. 228–239.
[32] S. Zhuang et al., "Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems," in *Proc. ACM SIGCOMM 2021 Conf.*, 2021, pp. 641–656.
[33] F. Borran and A. Schiper, "A leader-free byzantine consensus algorithm," in *Proc. Int. Conf. Distrib. Comput. Netw.*, Springer, 2010, pp. 67–78.
[34] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: Redundant Byzantine fault tolerance," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 297–306.
[35] K. Antoniadis et al., "Leaderless consensus," *J. Parallel Distrib. Comput.*, vol. 176, pp. 95–113, 2023.
[36] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "DBFT: Efficient leaderless Byzantine consensus and its application to blockchains," in *Proc. IEEE 17th Int. Symp. Netw. Comput. Appl.*, 2018, pp. 1–8.
[37] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The Honey Badger of BFT protocols," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 31–42.
[38] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 466–483.
[39] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless BFT consensus through metastability," 2019. [Online]. Available: https://arxiv.org/abs/1906.08936
[40] L. Zhang, Y. Wu, L. Chen, L. Fan, and A. Nallanathan, "Scoring aided federated learning on long-tailed data for wireless IoMT based healthcare system," *IEEE J. Biomed. Health Inform.*, vol. 28, no. 6, pp. 3341–3348, Jun. 2024.
[41] M. Allahbakhsh and A. Ignjatovic, "An iterative method for calculating robust rating scores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 2, pp. 340–350, Feb. 2015.
[42] R. Almakki, L. AlSuwaidan, S. Khan, A. R. Baig, S. Baseer, and M. Singh, "Fault tolerance Byzantine algorithm for lower overhead blockchain," *Secur. Commun. Netw.*, vol. 2022, 2022, Art. no. 1855238.
[43] X. Wang and Y. Guan, "A hierarchy Byzantine fault tolerance consensus protocol based on node reputation," *Sensors*, vol. 22, no. 15, 2022, Art. no. 5887.
[44] Y. Chen et al., "An improved algorithm for practical Byzantine fault tolerance to large-scale consortium chain," *Inf. Process. Manage.*, vol. 59, no. 2, 2022, Art. no. 102884.
[45] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "Hammerhead: Leader reputation for dynamic scheduling," 2023, arXiv: *2309.12713*.
[46] A. Gągol, D. Leśniak, D. Straszak, and M. Świętek, "Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes," in *Proc. 1st ACM Conf. Adv. Financial Technol.*, 2019, pp. 214–228.
[47] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.
[48] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2508–2530, Jun. 2006.
[49] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling," in *Proc. 27th ACM Symp. Princ. Distrib. Comput.*, 2008, pp. 145–154.
[50] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proc. Annu. Int. Cryptol. Conf.*, Springer, 2001, pp. 524–541.
[51] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 331–342.
[52] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[53] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *Proc. 24th IEEE Symp. Reliable Distrib. Syst.*, 2005, pp. 191–201.

Xuyang Liu (Graduate Student Member, IEEE) received the BE degree in computer science and technology from the Beijing Institute of Technology in 2022. He is currently working toward the PhD degree both with the School of Cyberspace Science and Technology, Beijing Institute of Technology and the School of Computer Science, the University of Auckland. His research interests include applied cryptography, blockchain technology, distributed consensus.

Zijian Zhang (Senior Member, IEEE) is a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is also a research fellow with the School of Computer Science, the University of Auckland. He was a visiting scholar with the Computer Science and Engineering Department of the State University of New York at Buffalo in 2015. His research interests include design of authentication and key agreement protocol and analysis of entity behavior and preference.

Zhen Li is currently working toward the PhD degree with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include privacy computing, AI security, and blockchain.

Hao Yin received the BE degree in information security and the MS degree in software engineering from the University of Science and Technology Beijing, in 2015 and 2018, respectively and the PhD degree in cyberspace security from the Beijing Institute of Technology, in 2022. He is currently an assistant researcher with the Research Center of Cyberspace Security, Peking University Changsha Institute for Computing and Digital Economy. His main research interests include applied cryptography, blockchain technology, and distributed consensus, privacy computing, fuzz testing, etc.



Meng Li (Senior Member, IEEE) received the PhD degree in computer science and technology from the School of Computer Science and Technology, Beijing Institute of Technology (BIT), China, in 2019. He is an associate professor and personnel secretary with the School of Computer Science and Information Engineering, Hefei University of Technology (HFUT), China. He was a postdoc researcher with the Department of Mathematics and HIT Center, University of Padua, Italy, where he is with the Security and PRIvacy Through Zeal (SPRITZ) research group led by Prof. Mauro Conti (IEEE Fellow). He was sponsored by ERCIM 'Alain Bensoussan' Fellowship Programme (from 2020.10.1 to 2021.3.31) to conduct postdoc research supervised by Prof. Fabio Martinelli with CNR, Italy. He was sponsored by China Scholarship Council (CSC) as a Joint PhD student (from 2017.9.1 to 2018.8.31) supervised by Prof. Xiaodong Lin (IEEE Fellow) with the Broadband Communications Research (BBCR) Lab, University of Waterloo and Wilfrid Laurier University, Canada. He is supported by CSC as a visiting scholar (from 2025.3.1 to 2026.2.28) collaborating with Prof. Mauro Conti (IEEE Fellow) with the HIT Center, University of Padua, Italy. His research interests include security, privacy, applied cryptography, blockchain, TEE, and Internet of Vehicles. In this area, he has published 106 papers in topmost journals and conferences, including IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, *ACM Transactions on Database Systems*, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, *IEEE Communications Surveys & Tutorials*, S&P, MobiCom, and ISSTA. He is a senior member of CIE, CIC, and CCF. He is an associate editor for IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, *EURASIP Journal on Audio, Speech, and Music Processing*, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, and *Computer Networks*. He has served as a TPC member for conferences, including ICDCS, TrustCom, ICC, Globecom, and HPCC. He is the recipient of 2024 IEEE HITC Award for Excellence (Early Career Researcher).

Jiamou Liu received the PhD degree in computer science from the University of Auckland, Auckland, New Zealand, in 2010. He is currently an associate professor with the School of Computer Science, University of Auckland. He was a senior lecturer with the Auckland University of Technology, Auckland, from 2011 to 2015 and a researcher with the Department of Computer Science, Leipzig University, Leipzig, Germany, from 2009 to 2010. His current research interests include social network analysis, multiagent systems, and algorithms.

Mauro Conti (Fellow, IEEE) received the PhD degree from the Sapienza University of Rome, Italy, in 2009. He is full professor with the University of Padua, Italy. He is also affiliated with TU Delft and University of Washington, Seattle. After his PhD, he was a postdoc researcher with Vrije Universiteit Amsterdam, The Netherlands. In 2011 he joined as assistant professor with the University of Padua, where he became associate professor in 2015, and full professor in 2018. He has been visiting researcher with GMU, UCLA, UCI, TU Darmstadt, UF, and FIU. He has been awarded with a Marie Curie Fellowship (2012) by the European Commission, and with a Fellowship by the German DAAD (2013). His research is also funded by companies, including Cisco, Intel, and Huawei. His main research interest is in the area of Security and Privacy. In this area, he published more than 400 papers in topmost international peer-reviewed journals and conferences. He is editor-in-chief for IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, area editor-in-chief for *IEEE Communications Surveys & Tutorials*, and has been associate editor for several journals, including *IEEE Communications Surveys & Tutorials*, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. He was program chair for TRUST 2015, ICISS 2016, WiSec 2017, ACNS 2020, CANS 2021, and general chair for SecureComm 2012, SACMAT 2013, NSS 2021 and ACNS 2022. He is senior member of the ACM, and fellow of the Young Academy of Europe.

Liehuang Zhu (Senior Member, IEEE) is a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is selected into the Program for New Century Excellent Talents in University from Ministry of Education, China. His research interests include cryptographic algorithms and secure protocols, Internet of Things security, cloud computing security, big data privacy, mobile and Internet security, and trusted computing.