

Byzantine Lattice Agreement in Synchronous Message Passing Systems

Xiong Zheng

Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Vijay Garg

Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Abstract

We propose three algorithms for the Byzantine lattice agreement problem in synchronous systems. The first algorithm runs in $\min\{3h(X) + 6, 6\sqrt{f_a} + 6\}$ rounds and takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages, where $h(X)$ is the height of the input lattice X , n is the total number of processes in the system, f is the maximum number of Byzantine processes such that $n \geq 3f + 1$ and $f_a \leq f$ is the actual number of Byzantine processes in an execution. The second algorithm takes $3 \log n + 3$ rounds and $O(n^2 \log n)$ messages. The third algorithm takes $4 \log f + 3$ rounds and $O(n^2 \log f)$ messages. All algorithms can tolerate $f < \frac{n}{3}$ Byzantine failures. This is the first work for the Byzantine lattice agreement problem in synchronous systems which achieves logarithmic rounds. In our algorithms, we apply a slightly modified version of the Gradecast algorithm given by Feldman et al [10] as a building block. If we use the Gradecast algorithm for authenticated setting given by Katz et al [12], we obtain algorithms for the Byzantine lattice agreement problem in authenticated settings and tolerate $f < \frac{n}{2}$ failures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Lattice agreement, Byzantine Failure, Gradecast

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.32

Related Version Full paper hosted on <https://arxiv.org/abs/1910.14141>.

Funding *Vijay Garg*: This work was supported in parts by the National Science Foundation Grants CNS-1812349, CNS-1563544, and the Cullen Trust Endowed Professorship.

1 Introduction

The lattice agreement problem, introduced by Attiya et al [2], is an important decision problem in shared-memory and message passing systems. In this problem, processes start with input values from a lattice and need to decide values which are comparable to each other. Specifically, suppose each process i has input x_i from a lattice (X, \leq, \sqcup) with the partial order \leq and the join operation \sqcup , it has to output a value y_i also in X such that the following properties are satisfied. 1) **Downward-Validity**: $x_i \leq y_i$ for each correct process i . 2) **Upward-Validity**: $y_i \leq \sqcup\{x_i \mid i \in [n]\}$. 3) **Comparability**: for any two correct processes i and j , either $y_i \leq y_j$ or $y_j \leq y_i$.

In shared-memory systems, algorithms for the lattice agreement problem can be directly applied to solve the atomic snapshot problem [1, 2]. This was the initial motivation for studying this problem. The application of lattice agreement in message passing systems has been explored only recently. Failero et al [9] were the first to apply lattice agreement for building a special class of linearizable replicated state machines, which can support query operation and update operation, but not mixed query and update operation. Traditionally, consensus based protocols are applied to build linearizable replicated state machines. However, consensus based protocols do not provide deterministic termination guarantee in the presence of failures in the system, since the consensus problem cannot be solved with even one failure



© Xiong Zheng and Vijay Garg;

licensed under Creative Commons License CC-BY

34th International Symposium on Distributed Computing (DISC 2020).

Editor: Hagit Attiya; Article No. 32; pp. 32:1–32:16



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in an asynchronous system [11]. The lattice agreement problem instead has been shown to be a weaker decision problem than consensus. It can be solved in an asynchronous system when a majority of processes is correct. Thus, linearizable replicated state machines built based on lattice agreement protocols have the advantage of termination even with failures. Another application of lattice agreement in distributed systems is to build atomic snapshot objects. Efficient implementation of atomic snapshot objects in crash-prone asynchronous message passing systems is important because they can make design of algorithms in such systems easier. The paper [2] presents a general technique for applying algorithms for the lattice agreement problem to solve the atomic snapshot problem. By using the same technique, algorithms for lattice agreement problem in distributed systems can be directly applied to implement atomic snapshot objects in crash-prone message passing systems. Essentially, an atomic snapshot object needs to provide linearizability for all processes, which decides on some total ordering of operations. In the lattice agreement problem, processes need to output values which lie in a chain of the input lattice, which is also a total ordering.

The lattice agreement problem in crash failure model has been studied both in synchronous and asynchronous systems. In synchronous systems, a $\log n$ rounds recursive algorithm based on “branch-and-bound” approach is proposed by Attiya et al [2] with message complexity of $O(n^2)$. The basic idea of their algorithm is to divide processes into two groups based on ids and let processes in the first group send values to processes in the second group. Each process in the second group takes join of received values. Then, this procedure continues within each subgroup. Their algorithm can tolerate at most $n - 1$ process failures. Later, the paper by Mavronicolasa et al [14] gave an algorithm with round complexity of $\min\{1 + h(X), \lfloor (3 + \sqrt{8f + 1})/2 \rfloor\}$, for any execution where at most $f < n$ processes may crash and $h(X)$ denotes the height of the input lattice X . Their algorithm has the early-stopping property and is the first algorithm with round complexity that depends on the actual height of the input lattice. The best upper bound for the lattice agreement problem in crash-failure model is given by Xiong et al [21], which is $O(\log f)$ rounds. The basic idea of their algorithm is again to divide processes into two groups based height of received values at each round and trying to achieve agreement within each group recursively.

In asynchronous systems, the lattice agreement problem was first studied by Faleiro et al in [9]. They present a Paxos style protocol when a majority of processes are correct. Their algorithm needs $O(n)$ asynchronous rounds in the worst case. The basic idea of their algorithm is to let each process repeatedly broadcast its current value to all at each round and update its value to be the join of all received values until all received values at a certain round are comparable with its current value. Later, Xiong et al [21] propose an algorithm which improves the round complexity to $O(f)$ rounds. For the best upper bound, Xiong et al [20] present an algorithm for this problem with round complexity of $O(\log f)$, which applies similar idea as [21] but with extra work to take care of possible arbitrary delay of messages in asynchronous systems.

The Byzantine failure model was first considered by Lamport et al [13] for the study of the Byzantine general agreement problem. For the lattice agreement problem in Byzantine failure model, Nowak et al [15] give an algorithm for a variant of the lattice agreement problem on cycle-free lattices that tolerates up to $f < \frac{n}{(h(X)+1)}$ Byzantine faults in asynchronous systems, where $h(X)$ is the height of the input lattice X . In their problem, the original Downward-Validity and Upward-Validity requirement are replaced with a different validity definition, which only requires that for each output value y of a correct process, there must be some input value x of a correct process such that $x \leq y$. With their validity definition, however, corresponding algorithms are not suitable for applications in atomic snapshot

objects and linearizable replicated state machines, since each process would like to have their proposal value included its output value. A more closely related work is the preprint by Di Luna et al [8], which proposes a reasonable validity condition and presents the first algorithm for asynchronous systems. Their algorithm takes $O(f)$ rounds. The basic idea of their algorithm is to first use the asynchronous Byzantine reliable broadcast primitive [5, 17] to let all correct processes disclose their input values to each other, based on which each correct process constructs a set of safe values. This set of safe values are the only values a correct process will possibly deliver in future rounds. After the disclosure phase, the remaining steps are similar to the algorithms given in [9, 21] for the lattice agreement problem in crash failure model, except that each process delivers a message only if all the values included in it are contained in its safe value set. Our algorithms assume synchronous systems but achieve exponential improvement in terms of round complexity.

For related works on application of lattice agreement, Faleiro et al [9] give procedures to build a linearizable and serializable replicated state machine which only supports query operation and update operation but not mixed queryte operation, based on lattice agreement protocols. Later, Xiong et al [20] propose some optimizations for their procedure for implementing replicated state machines in practice, specifically, they proposed a method to truncate the logs maintained. The recent paper by Skrzypczak et al [16] improves the procedure given in [9] in terms of memory consumption, at the expense of progress, and also demonstrates higher throughput.

Our main contribution in this paper is summarized in Table 1. Our first algorithm is early stopping because its round complexity depends on f_a : the actual number of Byzantine processes in an execution. Its basic idea is to let processes communicate using a modified Gradecast primitive and detect Byzantine processes along the way. The second and third algorithm are not early stopping but take logarithmic number of rounds. A building block of both algorithms is to construct a classifier procedure as in [2, 3, 21, 20] using a variant of the Gradecast primitive, but now the classifier procedure is Byzantine-tolerant and needs to guarantee different properties.

■ **Table 1** Our Results.

Problem	Reference	Rounds	Resilience
BLA	Concurrent work [7]	$O(\log f)$	$f < \frac{n}{4}$
	This paper	$\min\{3h(X) + 6, 6\sqrt{f_a} + 6\}$	$f < \frac{n}{3}$
		$3 \log n + 3$	
		$4 \log f + 3$	

In a concurrent work by Di Luna et al [7], they show that the Byzantine lattice agreement problem cannot be solved with $f \geq \frac{n}{3}$ failures in a synchronous systems. This shows that our algorithms achieve optimal resilience. In their paper, they give an algorithm for this problem which takes $O(\log f)$ rounds and tolerates $f < \frac{n}{4}$ failures, whereas our algorithms tolerate $f < \frac{n}{3}$ failures. With the assumption of digital signatures, they can improve the resilience to be $f < \frac{n}{3}$, whereas our algorithms tolerate $f < \frac{n}{2}$ failures.

2 System Model and Problem Definition

System Model: We assume a distributed message system with n processes in a completely connected topology, denoted as p_1, \dots, p_n . Every process can send messages to every other process. We consider synchronous systems, which means that message delays and the duration

of the operations performed by the process have an upper bound on the time. We assume that processes can have Byzantine failures but at most $f < n/3$ processes can be Byzantine in any execution of the algorithm. We use parameter f_a to denote the actual number of Byzantine processes in a system. By our assumption, we must have $f_a \leq f$. Byzantine processes can deviate arbitrarily from the algorithm. We say a process is correct or non-faulty if it is not a Byzantine process. We use C to denote the set of correct processes in an execution. We assume that the underlying communication system is reliable.

The Byzantine Lattice Agreement (BLA) Problem: Let (X, \leq, \sqcup) be a finite join semi-lattice with the partial order \leq and the join operation \sqcup . Two values u and v in X are comparable iff $u \leq v$ or $v \leq u$. The join of u and v is denoted as $\sqcup\{u, v\}$. X is a *join semi-lattice* if the join exists for every nonempty finite subset of X . As customary in this area, we use the term *lattice* instead of *join semi-lattice* in this paper for simplicity. More background on join semi-lattices can be found in [6].

In the Byzantine lattice agreement problem, each process p_i can propose a value x_i in X and must decide on some output y_i also in X in the presence of at most f Byzantine processes in the system. Let C denote the set of correct processes. Let f_a denote the actual number of Byzantine processes in the system. An algorithm is said to solve the Byzantine lattice agreement problem if the following properties are satisfied:

Comparability: For all $i \in C$ and $j \in C$, either $y_i \leq y_j$ or $y_j \leq y_i$.

Downward-Validity: For all $i \in C$, $x_i \leq y_i$.

Upward-Validity: $\sqcup\{y_i \mid i \in C\} \leq \sqcup(\{x_i \mid i \in C\} \cup B)$, where $B \subset X$ and $|B| \leq f_a$.

Remark: The Upward-Validity given by Attiya et al [2] is not suitable in the presence of Byzantine processes, since the input value for a Byzantine process is not defined. Thus, the extra B set is used to accommodate for possible values from Byzantine processes. The above Upward-Validity is similar to the Non-Triviality defined in [8]. The only difference is that the extra B set in [8] is required to have size at most f , which is the resilience parameter. One may argue that if a Byzantine process proposes the largest element of the input lattice, then correct processes may always decide on the largest element. For applications, we can impose an additional constraint on the initial proposal of all processes. For example, in the case of a Boolean lattice, we can require that the initial proposal for any process must be a singleton. More generally, we can impose the requirement that the initial proposal of any process must have the height less than some constant.

In this paper, for a given set $V \subseteq X$, we use $\mathcal{L}(V)$ to denote the join-closed subset of X that includes all elements in V . Clearly, $\mathcal{L}(V)$ is also a join semi-lattice. The height of a value v in a lattice X is defined as the length of longest chain from any minimal value to v , denoted as $h_X(v)$ or $h(v)$ when it is clear. The height of a lattice X is the height of its largest value in this lattice, denoted as $h(X)$. For two lattices \mathcal{L}_1 and \mathcal{L}_2 , we use $\mathcal{L}_1 \subseteq \mathcal{L}_2$ to mean that \mathcal{L}_1 is a sublattice of \mathcal{L}_2 .

3 Early Stopping Algorithm for BLA

In this section, we present an early stopping algorithm for the BLA problem, which applies a slightly modified version of the Gradecast algorithm given by Feldman et al [10] as a building block. The algorithm takes $\min\{3h(X) + 6, 6\sqrt{f_a} + 6\}$ rounds. Our algorithm has two primary ingredients which are quite different from the algorithm given in [9, 21] for the crash failure model. In the Byzantine failure model, correct processes can receive arbitrary values from a Byzantine process. In order to guarantee **Upward-Validity**, we do not want correct

processes to accept arbitrarily many values sent from a Byzantine process. The idea in [8] is to construct a safe value set, which stores the values reliably broadcast by each process at the first round. Later on, each process only delivers a received message if the values included in this message are contained in its safe value set. In this way, correct processes would not deliver arbitrary values sent by Byzantine processes. However, this idea can only provide $O(f)$ rounds guarantee.

To obtain the $O(\sqrt{f_a})$ rounds guarantee, our first idea is to let each correct process in our algorithm keep track of a lattice, which we call the safe lattice, instead of just a set of values. At each round, each correct process ignores all values received which are not contained in this safe lattice. By carefully updating this safe lattice of each correct process, our algorithm ensures that the value sent from a correct process is always in the safe lattice of any other correct process and Byzantine process cannot introduce arbitrary values to break the **Upward-Validity** condition. To get the $O(\sqrt{f_a})$ bound, another crucial ingredient of our algorithm is to apply the Gradecast algorithm at each round to detect the Byzantine processes which sends different values to different correct processes and let each correct process ignore messages from these processes. This idea is used in [4] to solve the Byzantine consensus problem in synchronous systems.

3.1 The Modified Gradecast Algorithm

Gradecast [10] is a three-round distributed algorithm that ensures some properties that are similar to those of broadcast. The Gradecast procedure has two parameters. The first one specifies the leader of the Gradecast and the second one represents the value that the leader would like to send. The output of process i in the Gradecast of leader p is a triple $\langle p, v_p^i, c_p^i \rangle$ where v_p^i is the value process i thinks the leader p has sent and c_p^i is the score assigned by i for the leader. The score assigned by process i for the leader is among $\{0, 1, 2\}$. We say c_p^i is the score assigned to the value or the leader by process i .

For our purpose, we do a slight modification of the Gradecast algorithm from [10] to enable processes to filter out some invalid values received and ignore messages from known Byzantine processes. The modified Gradecast algorithm can be found in our full paper [18]. We do the following modifications: 1) Let each process store a safe lattice to filter out all received values which are not in the lattice. We call this lattice: *the safe lattice*. Specifically, each process i keeps a safe value set, denoted as SV_i . This set is updated by process i at each round of the main algorithm. From SV_i , each process i constructs $\mathcal{L}(SV_i)$, the safe lattice as the join-closed subset of X which includes all values in SV_i . 2) Let each process store a bad set, which stores the Byzantine processes known by this process. Each process ignores the messages sent by processes in this set in the modified Gradecast algorithm. This bad set is also updated at each round of the main algorithm.

We assume that a correct leader always gradecasts a value which lies in the safe lattice of each correct process and the bad set of each correct process does not contain any correct process. We will show that these assumptions hold when we invoke the modified Gradecast algorithm as a substep in our main algorithm.

► **Lemma 1.** *Assume that a correct leader always gradecasts some value v which lies in the safe lattice of each other correct process and the bad set of each correct process does not contain a correct process. Then the modified Gradecast algorithm satisfies the following properties.*

1. *If the leader p is non-faulty then $v_p^i = v$ and $c_p^i = 2$, for any non-faulty i ;*
2. *For every non-faulty i and j : if $c_p^i > 0$ and $c_p^j > 0$ then $v_p^i = v_p^j$;*
3. *$|c_p^i - c_p^j| \leq 1$ for every non-faulty i and j .*

3.2 The Main Algorithm

The main algorithm, shown in Alg. 1, runs in synchronous rounds. For ease of presentation, we use rounds to mean the iterations in the **for** loop of the main algorithm. We call the rounds taken by the Gradecast step as sub-rounds. We assume for now that there is an upper bound F on the number of rounds of the main algorithm. We will establish the accurate value of F later and show that each process must decide on an output by round F . Initially, the bad set B_i and the safe value set SV_i for each process i are empty. Process i regards any value received as valid in the gradecast of the first round.

■ **Algorithm 1** Early Stopping Algorithm for the BLA Problem.

Algorithm for Process i :

$v_i := x_i$ //value held by i during the algorithm $B_i := \emptyset$ //set of faulty processes known by process i

$SV_i := \emptyset$ //set of safe values for process i

1: **for** $r := 1$ to F // F is an upper bound on the number of rounds

2: **Gradecast**(i, v_i)

3: Let $\langle j, u_j, c_j \rangle$ denote that process j gradecast u_j with score c_j

4: Define $U_i^1 := \{u_j \mid \langle j, u_j, c_j \rangle \wedge c_j \geq 1, j \in [n]\}$,

$U_i^2 := \{u_j \mid \langle j, u_j, c_j \rangle \wedge c_j = 2, j \in [n]\}$

5: Set $B_i := B_i \cup \{j \mid \langle j, *, c_j \rangle \wedge c_j \leq 1, j \in [n]\}$

6: Set $SV_i := U_i^1$

7: **if** v_i comparable with each value in U_i^2 **then** decide on v_i , but continue execution

8: Set $v_i := \sqcup\{u \mid u \in U_i^2\}$

9: **endfor**

At each round, each process invokes the modified Gradecast algorithm with its current value and acts as the leader. So there are at least $n - f$ Gradecast instances running at each round, with each instance corresponding to one correct process. After the Gradecast phase, each process i has a set of triples, one for each process which invoked Gradecast as the leader. A triple consists of the leader id, the value sent by the leader, and the score assigned by i . From these triples, processes i updates its bad set B_i and safe value set SV_i as follows. At line 5, process i includes all processes which are assigned score at most one into its bad set B_i and ignores all messages sent from processes in B_i at future rounds. Process i also updates its safe value set SV_i to be the union of all values gradecast by processes with score at least one. By updating the safe value set in this way, we can ensure that the current value of a correct process is in the safe lattice of every other correct process. Thus, the value gradecast by a correct process in the next round is valid for every other correct process, which implies property 1 of Gradecast. On the other hand, this safe value set also prevents Byzantine processes from gradecasting an arbitrary value, i.e, Byzantine processes can only gradecast values that belong to the safe lattice.

For the deciding condition at line 7, each process decides on its current value at a certain round if all values gradecast by processes with score 2 are comparable with its current value. A process keeps executing the algorithm even if it has decided on an output. It updates its current value to be the join of all values gradecast with score 2 and starts the next round.

3.3 Correctness and Complexity

We now prove the correctness of our algorithm. Due to space limitation, we put the proof of most lemmas and theorems in our full paper [18]. The variables used for the proof are defined in Table 2. The main algorithm has the following properties.

■ **Table 2** Notations for the Proof of Correctness of the Early Stopping Algorithm.

Variable	Definition
v_i^r	Value of process i at the end of round r
v^r	Auxiliary variable. The join of values of all correct processes at the end of round r , i.e., $v^r := \sqcup\{v_i^r \mid i \in C\}$
SV_i^r	The safe value set held by process i at the end of round r
S^r	Auxiliary variable. The union of all safe value sets held by correct processes at the end of round r , i.e., $S^r = \cup\{SV_i^r \mid i \in C\}$
s^r	Auxiliary variable. The join of all values in S^r , i.e., $s^r = \sqcup\{v \mid v \in S^r\}$
c_j^i	The score process i assigned to process j in the Gradecast with j as the leader.

► **Lemma 2.** *Let i and j be any two correct processes. For any $1 \leq r \leq F$, the main algorithm satisfies the properties below.*

(p1) $v_i^r \in \mathcal{L}(SV_j^r)$ (p2) $v^r \in \mathcal{L}(SV_i^r)$ (p3) $v^r \leq s^r$ (p4) $v_i^{r-1} < v_i^r$ if process i is undecided at the end of round r

(p5) $v^r < v^{r+1}$ if at least one correct process is not decided at the end of round r

(p6) $\mathcal{L}(S^{r+1}) \subseteq \mathcal{L}(S^r)$ (p7) $s^{r+1} \leq s^r$ (p8) For each correct process i , its bad set B_i never contains a correct process (p9) $v^r \leq v_i^{r+1}$

Property (p1) and (p8) immediately justify the assumption in Lemma 1. Thus, all properties of Gradecast are satisfied. Property (p3) indicates that the join of all values of correct processes is less than the join of all values in the safe value sets of all correct processes at any round. Property (p4) and (p5) imply that the join of all values of correct processes is strictly increasing. Property (p7), implied by (p6), indicates that the join of all values in the safe value sets of all correct processes is non-increasing. Then, we must have $v^r = s^r$ at some round r . After this round, the deciding condition must be satisfied for each process.

Now we show the algorithm satisfies all the properties required by the BLA problem. For now we assume that each process decides within F rounds. We show the accurate value of F when we analyze the round complexity.

► **Lemma 3.** *The values decided by correct processes satisfy all the properties of BLA.*

Proof. (Sketch) **Comparability.** If two processes decide at the same round, their decision value must be comparable by the deciding condition. Otherwise, the process decides in a later round must receive the decision value of the other process.

Upward-Validity. The safe lattice kept by each process guarantees that each Byzantine process can introduce at most one value into the decision value of correct processes. ◀

We now analyze the round and message complexity of our algorithm. The following lemma along with property (p5) and (p7) of Lemma 2 guarantees the termination of our algorithm. It can be derived from (p1), (p7), and (p9) of Lemma 2 and the decision condition.

► **Lemma 4.** *If $v^r = s^r$ at the end of some round r , then all undecided correct processes decide in at most 2 rounds.*

► **Lemma 5.** $F \leq h(X) + 2$, where X is the input lattice and $h(X)$ is the height of X .

Proof. (Sketch) (p3), (p5) and (p7) of Lemma 2 implies that $v^r = s^r$ in at most $h(X)$ rounds. Then, Lemma 4 implies that $F \leq h(X) + 2$. ◀

We now show that the algorithm takes $O(\sqrt{f})$ rounds. We first observe that if a process is in the bad set of each correct process, then its gradecast will be graded with score 0 by each correct process. We introduce the notion of terrible processes. A process is **terrible** at round r if it is graded with score 2 by at least one correct process in each round before r and no correct process grades it with score 2 at round r . From the above definition, we observe that the terrible processes at each round are included into B_i for each correct i at line 5 of the algorithm. So, a Byzantine process can be terrible at most once.

► **Lemma 6.** *Suppose there are f_r terrible processes at round r , then each process decides within $r + f_r + 2$ rounds.*

► **Lemma 7.** *$F \leq 2\sqrt{f} + 2$, where f is the maximum number of Byzantine failures in the system such that $n \geq 3f + 1$.*

Proof. Consider the first \sqrt{f} rounds. At least one of these rounds has less than \sqrt{f} terrible processes. By Lemma 6, starting from that round, each undecided process needs at most $\sqrt{f} + 2$ more rounds to decide. Thus, the total number of rounds is at most $2\sqrt{f} + 2$. ◀

To obtain the $O(\sqrt{f_a})$ rounds guarantee, we let each correct process dynamically update its termination round, which denotes the round number such that a correct process terminates the algorithm in any case. Specifically, if a process includes t Byzantine processes into its bad set, then this process runs $\sqrt{t} + 2$ more rounds and terminate.

► **Theorem 8.** *There is a $\min\{3h(X) + 6, 6\sqrt{f_a} + 6\}$ rounds algorithm for the Byzantine lattice agreement problem in synchronous systems, which can tolerate $f < \frac{n}{3}$ Byzantine failures. f_a is the actual number of Byzantine failures. The term $h(X)$ is the height of the input lattice X . The algorithm takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages.*

► **Corollary 9.** *There is a $\min\{4h(X) + 8, 8\sqrt{f_a} + 8\}$ rounds algorithm for the authenticated (allow digital signatures) BLA problem in synchronous systems, which can tolerate $f < \frac{n}{2}$ Byzantine failures. The algorithm takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages.*

Proof. Using the 4-round Gradecast algorithm [12] in the authenticated setting that tolerates $f < \frac{n}{2}$ Byzantine failures immediately gives the result. ◀

4 $O(\log n)$ Rounds Algorithm for the BLA problem

The $3 \log n + 3$ round algorithm shown in this section is inspired by algorithms proposed for the crash failure model in [2, 21]. The basic idea is to divide a group of processes into the slave subgroup and the master subgroup based on process ids, and ensure the property that the value of any correct process in the slave group is at most the value of any correct process in the master group. With the above property, if we recurse within each subgroup, then all correct processes can obtain comparable values in $O(\log n)$ rounds.

In the Byzantine failure model, however, simply ensuring the above property is not enough. For example, suppose we divided a group of processes G into the slave group $S(G)$ and the master group $M(G)$ such that the above property is satisfied. Suppose there is a Byzantine process in $S(G)$, it might send a value to some correct process in $S(G)$ in a later round such that the value is not known by correct processes in $M(G)$. Then, a correct slave process might have a value which is greater than some master process.

In order to prevent such cases, our algorithm introduces two novel ideas. First, when we divide a group into the slave subgroup and the master subgroup, we apply a modified Gradecast algorithm to guarantee that the value of a slave process is at most the value

of a master process. The Gradecast algorithm serves the same purpose as the *Classifier* procedure as given in [21, 20]. A nice property of the modified Gradecast algorithm is that if some correct process assigns score 2 for the value gradecast by the leader, then each other correct process assigns score at least 1 for this value. Suppose we let each process in a group gradecast its value. Let U^2 denote the set of values assigned score of 2 by some correct process. Let U^1 denote the set of values assigned score of at least 1 by some correct process. Then, we must have $U^2 \subseteq U^1$. If each process in the master group updates their value to U^1 and each process in the slave group updates their value to U^2 , then it is guaranteed that the set of all values of the slave group is a subset of the values of each master process.

However, the above property is only guaranteed at the current recursion level. Suppose there is a Byzantine process in the slave group, then it can gradecast a new value, which is not contained in the value set of some master process, to correct processes in the slave group. Then, the above property that the value of any correct slave process is at most the value of any correct master process does not hold any more. We need to ensure that the values of all slave processes are always a subset of the values of each master process when the recursion within each subgroup continues. To achieve that, we introduce a second novel idea. We let each process keep track of a safe value set for each other process and regard any value received from that process but not in the safe value set as invalid. This is also different from the algorithm in previous section, where each process just keeps track of one single safe lattice for all. These safe values sets are used to restrict what values a process in a slave group can send. If we can guarantee that the union of all safe value sets for processes in the slave group is a subset of the value set of each master process, then we can ensure that the above property continues to hold.

4.1 The SetGradecast Algorithm

In the $3 \log n + 3$ rounds algorithm, a process needs to gradecast a set of values instead of just one single value. Thus, we propose the **SetGradecast** algorithm (presented in the full paper [18]) which is similar to the **Gradecast** algorithm, except that it is used to gradecast a set of values. In the $3 \log n + 3$ rounds algorithm, each process i keeps track of a safe array S_i of size n with $S_i[j]$ being a safe value set for process j . Process i considers a value v received from process j as valid if $v \in S_i[j]$, otherwise invalid. Process i uses S_i to filter out invalid values received from any process in the **SetGradecast** algorithm. We show how to construct and update the safe value array for each process in the main algorithm.

In the **SetGradecast** algorithm, we assume that the leader needs to gradecast a set of distinct values, which can be guaranteed by introducing some unique tags for each value. If a process receives a message which contains duplicate values from some leader, the leader must be Byzantine. It just ignores the message. Each process i returns a triple $\langle j, R_j, C_j \rangle$ when process j invokes the **SetGradecast** as the leader. The set R_j is the set of values gradecast by process j with score at least 1 and the map C_j stores the score assigned by process i for each value in R_j . Let v be an arbitrary value gradecast by the leader. Let c_v^i denote the score of v assigned by process i . Then, we have the following lemma.

► **Lemma 10.** *Algorithm **SetGradecast** has the following properties.*

1. *If a value v gradecast by a correct leader i is in the safe value set of each correct process j for i , i.e., $v \in S_j[i]$ for each correct $j \in [n]$, then the score of v assigned by each correct j must be 2, i.e., $c_v^j = 2$.*
2. *Let v be an arbitrary value gradecast by the leader. Then $|c_v^i - c_v^j| \leq 1$ for any two correct process i and j .*
3. *If a value v gradecast by a leader i is not in the safe value set of any correct process for i , i.e., $v \notin S_j[i]$ for any $j \in C$, then $c_v^j = 0$ for each $j \in C$.*

■ **Algorithm 2** The $3 \log n + 3$ Rounds Algorithm for the BLA Problem.

Let \mathcal{G}_r denote the collection of groups at round r , initially $\mathcal{G}_1 = \{G_1\}$
 x_i : input value for process i V_i : value set of process i with $V_i := \{x_i\}$ initially.
 S_i : an array of size n with $S_i[j]$ being the safe value set for process j .

- 1: **for** each process i , **in parallel do** /* Build the Initial Safe Array */
- 2: Process i invokes **Gradecast**(i, x_i)
- 3: Let $\langle j, v_j, c_j \rangle$ denote the tuple obtained from the gradecast of p_j
- 4: Set $S_i[k] := \{v_j \mid c_j \geq 1, j \in [n]\}$ for each process $k \in [n]$
- 5: **endfor**
- 6: **for** $r := 1$ to $\log n$
- 7: Divide each group $G \in \mathcal{G}_r$ into the slave subgroup $S(G)$ and the master subgroup $M(G)$ based on process ids
- 8: Let \mathcal{G}_{r+1} denote the collection of new groups
- 9: Each slave process p executes **SetGradecast**(p, V_p)
- 10: **for** each process i , **in parallel do**
- 11: Let $\langle j, Val_j^i, C_j^i \rangle$ denote the leader-value-score triple that p_i obtained for p_j
- 12: Let R_j^i denote the set of values assigned score 2 by p_i in the gradecast of p_j ,
 i.e., $R_j^i := \{v \in Val_j^i \mid C_j^i[v] = 2\}$
- 13: **for** each group $G \in \mathcal{G}_r$
- 14: Let U_1 denote the set of values sent by processes in $S(G)$ with score ≥ 1 ,
 i.e., $U_1 := \bigcup_{j \in S(G)} Val_j^i$
- 15: Let U_2 denote the set of values sent by processes in $S(G)$ with score 2,
 i.e., $U_2 := \bigcup_{j \in S(G)} R_j^i$
- 16: $S_i[j] := U_2$ for each process $j \in S(G)$
- 17: $S_i[j] := S_i[j] \cup U_1$ for each process $j \in M(G)$,
- 18: **if** $i \in S(G)$ **then** $V_i := U_2$
- 19: **elif** $i \in M(G)$ **then** $V_i := U_1$
- 20: **endfor**
- 21: **endfor**
- 22: **endfor**
- 23: Output $y_i := \sqcap \{v \in V_i\}$

4.2 The Main Algorithm

The $3 \log n + 3$ rounds algorithm is shown in Alg. 2. In the algorithm, each process i stores a value set V_i which is updated at each round. Initially, $V_i = \{x_i\}$. Each process i keeps track of a safe array S_i of size n with $S_i[j]$ being the safe value set for j . Process i regards any value received from j which is not in $S_i[j]$ as invalid and thus ignores it. Different processes may have different safe value sets for a process j . Initially, all processes are in the same group, denoted as G_1 . During the algorithm, processes might be divided into different groups. The algorithm proceeds as follows.

The initial round at lines 1-5 is used to build the initial safe array of each process i . At this round, each process i invokes the **Gradecast** algorithm to send its input value x_i to all. Each process i constructs the same initial safe value set for each process j , which includes all values gradecast by some process and assigned score of at least 1 by process i . We will show

later that this initial round of gradecast guarantees **Upward-Validity** of the BLA problem. Intuitively, this is because each Byzantine process can only introduce one value into the safe value sets by properties of Gradecast.

In lines 6-22, at each round r from 1 to $\log n$, each group G is divided into two subgroups: the slave group $S(G)$ and the master group $M(G)$, where $S(G)$ contains all processes in G with ids in the lower half and $M(G)$ contains all processes in G with ids in the upper half. Each process in $S(G)$ invokes **SetGradecast** to gradecast its current value set to all processes. Processes in $M(G)$ do not gradecast their value sets. After this step, for each group G , process i obtains a set of values gradecast by processes in $S(G)$. From line 13 to line 20, process i updates its safe value set S_i and its value set V_i based the values obtained in all **SetGradecast** instances. At line 16, process i updates its safe value set for each process $j \in S(G)$ to be the values gradecast with score 2 by some process in $S(G)$. At line 17, it updates the safe value set for each process in $M(G)$ to be the values gradecast with score at least 1 by some process in $S(G)$. If process i is a slave process in $S(G)$, it updates its value set to be the set of values gradecast by processes in $S(G)$ and assigned score of 2. If it is a master process in $M(G)$, it updates its value set to be the set of values gradecast by processes in $S(G)$ with score at least 1.

■ **Table 3** Notations for the Proof of Correctness of the $O(\log n)$ Algorithm.

Variable	Definition
V_i^r	The value set held by process i at the end of round r .
S_i^r	The safe value array of i at the end of round r .
SF_j^r	Auxiliary variable. The union of the safe value sets of all correct process for j at the end of round r , i.e., $SF_j^r := \{S_i^r[j] \mid i \in C\}$
SF_G^r	Auxiliary variable. The union of the safe value sets of all correct process for processes in group G at the end of round r , i.e., $SF_G^r := \bigcup_{j \in G} SF_j^r$

4.3 Correctness and Complexity

Now we analyze the correctness and complexity of our algorithm. For any group G , let $S(G)$ and $M(G)$ denote the slave group and the master group obtained when dividing G . The variables we use for analysis are given in Table 3. Detailed proof of lemmas can be found in our full paper [18].

► **Lemma 11.** *Let G be a group that is divided into $S(G)$ and $M(G)$ at round r . Then*
 (p1) $SF_{S(G)}^r \subseteq SF_G^{r-1}$ (p2) $SF_{M(G)}^r \subseteq SF_G^{r-1}$ (p3) For each $i \in G \cap C$, $V_i^r \subseteq SF_G^{r-1}$

The following lemma shows that if a value of correct process i is contained in the safe value set of each correct process for process i , then this value remains in the value set of process i . It also implies **Downward-Validity**.

► **Lemma 12.** *Consider an arbitrary value $v \in V_j^r$ of correct process j . If it is contained in $S_i^r[j]$ for each correct process i , then we have (1) $v \in S_i^t[j]$ for each correct process i and $t \geq r$. (2) $v \in V_j^t$ for any $t \geq r$.*

The following lemma shows that the union of all safe value sets of correct processes for slave processes will always be a subset of the values of each master process.

► **Lemma 13.** *Let G be a group which is divided into $S(G)$ and $M(G)$ at round r . Then $SF_{S(G)}^r \subseteq V_j^t$ for each correct $j \in M(G)$ and any round number $t \geq r$.*

► **Theorem 14.** *There is a $3 \log n + 3$ rounds algorithm for the Byzantine lattice agreement problem in synchronous systems, which can tolerate $f < \frac{n}{3}$ Byzantine failures. The algorithm takes $O(n^2 \log n)$ messages.*

Proof. (Sketch) **Comparability.** Let G denote the last group both i and j belong to at round r . W.l.o.g, suppose $i \in S(G)$ and $j \in M(G)$. By (p1), (p2) and (p3) of Lemma 11, we have $V_i^{\log n} \subseteq SF_{S(G)}^r$. Lemma 13 implies that $SF_{S(G)}^r \subseteq V_j^{\log n}$. Thus, $V_i^{\log n} \subseteq V_j^{\log n}$.

Upward-Validity. The initial round ensures that each Byzantine process can introduce at most one value into the initial safe value sets of correct processes. ◀

► **Corollary 15.** *There is a $4 \log n + 4$ rounds algorithm for the authenticated BLA problem in synchronous systems which can tolerate $f < \frac{n}{2}$ Byzantine failures, where n is the number of processes. The algorithm takes $O(n^2 \log n)$ messages.*

5 $O(\log f)$ Algorithm for the BLA Problem

In this section, we present an algorithm for the BLA problem which takes $4 \log f + 3$ synchronous rounds. The $O(\log f)$ algorithm by Xiong et al. in [21] for the crash failure model uses a crash-tolerant classifier procedure. The idea of using a classifier procedure to obtain comparable views is initially applied to implement atomic snapshot objects in shared memory systems by Attiya et al. [3]. Their crash-tolerant classifier procedure divides a group of processes into two subgroups: the master group and the slave group based on a threshold parameter k and guarantees the following properties. (C1) The value of any slave process \leq the value of any master process. (C2) The value of any master process has height in the input lattice $> k$. (C3) The join of all slave values has height $\leq k$.

With the above properties, the classifier procedure can be recursively applied within each subgroup and all processes have comparable values after $O(\log f)$ rounds by setting the knowledge threshold k in a binary way as follows. Initially, all processes are in the same group with initial knowledge threshold $n - \frac{f}{2}$. Consider a group G at level r with knowledge threshold k , then the slave group of G has knowledge threshold $k - \frac{f}{2^{r+1}}$ and the master group of G has knowledge threshold $k + \frac{f}{2^{r+1}}$. If all processes exchange their values before recursively invoking the classifier procedure, each correct process must have at least $n - f$ values and have at most n values. Then, after $\log f$ levels of recursion, by applying property (C1) and (C2) recursively, all processes in different groups must have comparable values and all processes in the same group must have the same value.

The crash-tolerant classifier procedure in [21] is quite simple. All processes within the same group exchange their current values. If a process obtains a value set with size greater than k , it is classified as a master. Otherwise, it is classified as a slave. A slave process keeps its value unchanged. A master process updates its value to be the join of all values received.

In presence of Byzantine processes, however, properties (C1)-(C3) are not sufficient for subgroups to invoke the classifier procedure recursively due to the following reason. Even if we can guarantee (C1) and (C3) at the current classifier, Byzantine processes can introduce additional values into the slave group when processes in the slave subgroup invokes the classifier within themselves. Then, properties (C1) and (C3) can be violated.

In our algorithm, we apply a *Byzantine-tolerant classifier procedure* to divide a group of processes into two subgroups: the slave group and the master group. A group of processes apply the Byzantine-tolerant classifier procedure to update their value sets and decide which subgroup they are classified into. In our algorithm, each process i keeps track of a value set V_i : a set of values, which is updated in the classifier procedure. Similar to the $O(\log n)$

algorithm, we let each process store a safe value set for each group which restricts the values that processes in this group can send. If SF is the union of the safe value sets of all correct processes for the slave group, then, our algorithm guarantees that the value set of any slave process must be always a subset of SF . The following properties are guaranteed after a group of processes invoke the Byzantine-tolerant classifier procedure within themselves. (B1) *The union of the safe value sets of all correct processes* for the slave group is a subset of the value set of a master process. (B2) The value set of any master process has size $> k$. (B3) *The union of the safe value sets of all correct processes* for the slave group has size $\leq k$.

Property (B1) guarantees that the value set of a slave process is always a subset of the value set of a master process starting from the point when they are classified into different subgroups. For property (B3), our algorithm maintains the following variant: a value is considered as valid by a correct process if it is in the safe value set of some correct process. Thus, property (B3) restricts what values a group of processes can ever send, which are also the values processes in the group can ever have (will be more clear in our algorithm).

To guarantee (B1) and (B2), similar to the $O(\log n)$ algorithm, we use the SetGradecast algorithm as a communication primitive and update the safe value sets of correct processes carefully. To guarantee (B3), dividing into the slave subgroup and the master subgroup in our algorithm is based on the safe value sets of processes.

From property (B1) and (B3), we can observe that the Byzantine-tolerant classifier procedure for a group of processes depends on not only processes in the group but also other processes in the system (via the safe value sets). Thus, in our presentation, we do not present the Byzantine-tolerant classifier procedure as a separate procedure, instead we present it inside the main algorithm and call it as a *classification step*.

In the $O(\log n)$ algorithm, we divide a group into slave subgroup and master subgroup based on process ids. A Byzantine process cannot lie about its group identity, i.e., whether it is in the slave group or the master group. In the $O(\log f)$ algorithm, the classification is based on the values received at each round. So, process i does not know whether process j is classified as a slave or a master at any given round. Thus, a Byzantine process can lie about its group identity and the $O(\log f)$ algorithm needs a mechanism to prevent such lies. In the $O(\log f)$ algorithm, each process $i \in [n]$ has a label l_i , which serves as the threshold when it performs the classification step and also indicates its group identity. We require that when a process sends a value, it needs to attach its label.

We formally define a *group* as a set of processes which have the same label. The *label of a group* is the label of the processes in this group. The label of a group is also the threshold value processes in this group use to do classification. We also use label to indicate a group. We say a process is in group k if its value is associated with label k . Initially all processes are within the same group with label $k_0 = n - \frac{f}{2}$.

Consider the classification step for group G with label k . There are two main differences between the $O(\log n)$ algorithm and the $O(\log f)$ algorithm. First, in the $O(\log n)$ algorithm, only processes in the slave group of G invoke the gradecast primitive to send its current value set. In the $O(\log f)$ algorithm, all processes in group G invoke the gradecast primitive. Second, in the $O(\log n)$ algorithm, the classification is based on process ids. In the $O(\log f)$ algorithm, we let each process send its safe value set for group G to all processes in G and each process in G performs classification based on the safe sets received. If the union of the received safe sets has size $> k$, the process is classified as a master, otherwise as a slave. Each slave process updates its value to be the set of values with score 2. Each master process updates its value to be the union of its current value and the set of values with score at least 1. The safe value set is updated in a similar manner to the $O(\log n)$ algorithm.

■ **Algorithm 3** The $4 \log f + 3$ Rounds Algorithm for the BLA Problem.

x_i : input value for process i V_i : value set of process i . $V_i := \{x_i\}$ initially.
Initially all process are within the same group with label $k_0 = n - \frac{f}{2}$
 l_i : label of p_i . $l_i := k_0 = n - \frac{f}{2}$ initially and is updated at each round
Map F_i , with $F_i[k]$ being the safe value set for processes with label k .

- 1: **for** each process i , **in parallel do** /* Build the Initial Safe Map and Value Set */
- 2: Execute **Gradecast**(i, x_i)
- 3: Let $\langle j, v_j, c_j \rangle$ denote that process j gradecasts v_j with score c_j
- 4: Set $F_i[k_0] := \{v_j \mid c_j \geq 1 \wedge j \in [n]\}$ // safe set for the initial group with label k_0
- 5: Set $V_i := \{v_j \mid c_j = 2 \wedge j \in [n]\}$
- 6: **endfor**
- 7: **for** $r := 1$ to $\log f$
- 8: **for** each process i , **in parallel do**
- 9: Execute **SetGradecast**(i, V_i, l_i)
- 10: Let L denote the set of labels received in all Gradecasts
- 11: **for** each label $k \in L$ /* Each label represents a group */
/* Lines 12-21 is the classification step for group k */
- 12: Let $U_{i,1}^k$ denote the set of values with label k and assigned score ≥ 1 by p_i .
- 13: Let $U_{i,2}^k$ denote the set of values with label k and assigned score 2 by p_i
- 14: Set $F_i[m(k, r)] := F_i[k] \cup U_{i,1}^k$ and $F_i[s(k, r)] := U_{i,2}^k$
- 15: Send $U_{i,2}^k$ to processes who gradecast with label k .
- 16: **if** $l_i = k$ //if my label is k
- 17: Let R_j denote the set of values received from p_j at line 15
- 18: Set $T := \cup\{R_j \mid R_j \subseteq U_{i,1}^k, j \in [n]\}$
- 19: /* Classification */
- 20: **if** $|T| > k$ **then** $V_i := U_{i,1}^k$, $l_i := l_i + \frac{f}{2^{r+1}}$ //master process
- 21: **else** $V_i := U_{i,2}^k$, $l_i := l_i - \frac{f}{2^{r+1}}$ //slave process
- 22: **endifor**
- 23: **endifor**
- 24: **endifor**
- 25: $y_i := \sqcup\{v \in V_i^{\log f + 1}\}$

In the $O(\log f)$ algorithm, each process $i \in [n]$ keeps track of a safe value map F_i with $F_i[k]$ being the safe value set of process i for group k , i.e., $F_i[k]$ is an upper bound on the values with label k that process i considers valid. Define $s(k, r) = k - \frac{f}{2^{r+1}}$ and $m(k, r) = k + \frac{f}{2^{r+1}}$. The algorithm is shown in Alg. 3.

In lines 1-6 of the algorithm, each process first invokes **Gradecast** to send its input value to all. Then, it constructs its value set as the set of values gradecast with score 2 and its initial safe value set for the initial group as the set of values gradecast with score at least 1. Lines 1-6 serve two purposes: 1) To construct the safe value set for the initial group with label $k_0 = n - \frac{f}{2}$ and ensure that each Byzantine process can introduce at most one value in the safe value sets 2) Ensure that there are at most f values unknown to each correct processes. Then, $\log f$ recursion levels suffice for all processes to obtain comparable values.

In lines 7-25, at each round, each process invokes **SetGradecast** to send its current value to all and performs classification. When a process invokes the **SetGradecast** to send its current value set, its current label is attached. After the gradecast step, for each process

$j \in [n]$, process i obtains a value-score-label triple from the gradecast of process j . Then, process i executes line 11-15 for each group at the current round. Specifically, for the group with label k , process i obtains the set of values with score at least 1, $U_{i,1}^k$, and the set of values with score 2, $U_{i,2}^k$, from the gradecast of processes with label k . Then, process i updates its safe value set for group $m(k, r)$, i.e., the master group of group k , to be the union of its safe value set for group k and $U_{i,1}^k$. It also updates its safe value set for group $s(k, r)$, i.e., the slave group, to be $U_{i,2}^k$. Due to property 2 of SetGradecast, we must have $U_{i,2}^k \subseteq U_{j,1}^k$ for any process i and j . This step is to ensure that if a master process j obtains a value in $U_{i,2}^k$, this value will be in the safe value set of each correct process for j . Then, when the master process tries to send this value using SetGradecast, it must be assigned score of 2 by each correct process, due to property 3 of SetGradecast. At line 15, process i sends its safe value set for the slave group to the processes in group k .

Lines 16-21 are only executed by processes in group k . They obtain the set of values sent by all processes at line 15 and do classification based the size of this set. To prevent a Byzantine process from sending arbitrary values at line 15, each process in group k only accepts the set R_j from process j if R_j is a subset of $U_{i,1}^k$. If process j is correct, this condition must hold by property 2 of SetGradecast. So the sets sent from correct processes will always be accepted. Then, the set T at line 18 must contain all the sets sent from correct processes. Since each such set is the safe value set of a process for the slave group, each process in group k is actually doing classification based on the safe sets of processes for the slave group. Lines 20-21 is the classification step. If the size of T is greater than k , then the process is classified as a master and updates its value to be the set of values gradecast by processes in group k with score at least 1. Otherwise, its value is updated to be the set of values gradecast by processes in group k with score 2. Its label is updated based on whether the process is a master or a slave.

Due to space limitation, we put the correctness proof and complexity analysis in our full paper [18]. Our main result is summarized in the following Theorem.

► **Theorem 16.** *There is a $4 \log f + 3$ rounds algorithm for the BLA problem in synchronous systems which can tolerate $f < \frac{n}{3}$ Byzantine failures. It takes $O(n^2 \log f)$ messages.*

► **Corollary 17.** *There is a $5 \log f + 4$ rounds algorithm for the authenticated BLA problem in synchronous systems which can tolerate $f < \frac{n}{2}$ Byzantine failures. It takes $O(n^2 \log f)$ messages.*

6 Conclusion

We have presented three algorithms for the Byzantine lattice agreement problem in synchronous systems. The first algorithm has early stopping property. The second algorithms take logarithmic rounds. The $O(\log f)$ rounds upper bound matches the bound for the crash failure setting. For future work, the following questions are interesting: 1) Can we improve the upper bound or prove some lower bound on the round complexity? 2) Can we solve the BLA problem in asynchronous systems in logarithmic rounds? We have some partial results in [19], but the algorithm proposed can only tolerate $f < \frac{n}{5}$ Byzantine failures.

References

- 1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
- 2 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
- 3 Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
- 4 Michael Ben-Or, Danny Dolev, and Ezra N Hoch. Simple gradecast based algorithms. *arXiv preprint arXiv:1007.1049*, 2010.
- 5 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- 6 B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.
- 7 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, Silvia Bonomi, and Leonardo Querzoni. Synchronous byzantine lattice agreement in $O(\log f)$ rounds. *arXiv preprint arXiv:2001.02670*, 2020.
- 8 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. *arXiv preprint arXiv:1910.05768*, 2019.
- 9 Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
- 10 Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 148–161. ACM, 1988.
- 11 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- 12 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.
- 13 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
- 14 Marios Mavronicolasa. A bound on the rounds to reach lattice agreement. <http://www.cs.ucy.ac.cy/mavronic/pdf/lattice.pdf>, 2018.
- 15 Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 16 Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable state machine replication of state-based crdts without logs. *arXiv preprint arXiv:1905.08733*, 2019.
- 17 TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.
- 18 Xiong Zheng and Vijay Garg. Byzantine lattice agreement in synchronous systems. *arXiv preprint arXiv:1910.14141*, 2019.
- 19 Xiong Zheng and Vijay Garg. Byzantine lattice agreement in asynchronous systems. *arXiv preprint arXiv:2002.06779*, 2020.
- 20 Xiong Zheng, Vijay K. Garg, and John Kaippallimalil. Linearizable Replicated State Machines With Lattice Agreement. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 21 Xiong Zheng, Changyong Hu, and Vijay K Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.