Byzantine-Tolerant Privacy-Preserving Atomic Register

Vincent Kowalski, Achour Mostéfaoui, Matthieu Perrin, Sinchan Sengupta

Byzantine-Tolerant Privacy-Preserving Atomic Register

Vincent Kowalski, Achour Mostéfaoui, Matthieu Perrin, and Sinchan Sengupta

LS2N, Nantes Université, France.

Abstract. This paper presents the construction of a privacy-preserving single-writer multi-reader (SWMR) atomic register in a Byzantine-prone distributed model. Specifically, we consider a closed model, in which one process can write values in the register, and only a portion of the other processes are allowed to read the value. The aim is to ensure that processes that do not have the requisite reading rights are unable to read the content of the register, even if they are Byzantine. This makes the content of the register private. We ensure this privacy by encoding the value written by the writer, using secret sharing, into multiple shards and disseminating them among the participating reader processes. The technical challenge is then organize the coordination between correct reading processes to achieve Byzantine linearizability, without disseminating the content of the register. The main contribution of this paper is a linearizable read-write (R/W) privacy-preserving register for $t < \frac{n}{7}$, where t denotes the number of Byzantine processes and n denotes the total number of processes in the system.

Keywords: Byzantine · Linearizability · Privacy-protection · Secret-sharing.

1 Introduction

Context and Motivation Suppose we have a collection of medical data that is important for diagnosis. This data is personal to an individual and he would not want it to be shared amongst everyone for security and privacy reasons, except the doctor or medical personnel treating him. This medical data can only be changed by the doctor to update the history of the medical condition of the patient. Connecting this to a technical analogy, we consider a distributed system, where the data we want to share is placed in a register and participating processes try to access it. However, only a subset of processes having the requisite permission to read the data can access its value. For the processes not having the read permission, the register is opaque and we achieve the much-needed privacy. There exists a designated process, the *writer*, who can write in the register. Additionally, this register should be linearizable, which means intuitively that it should behave as if it is the only physical copy shared amongst all the processes.

A distributed system can be prone to asynchrony, crashes, and security issues. A special category of faults exhibited by processes known as Byzantine faults

was introduced by Lamport in [8] and [10]. A Byzantine process is allowed to exhibit any random behavior it wants, not conforming to the specified algorithm. A malicious trait exhibited by such processes is that it can lie about its state, and collude with other Byzantine processes to fail the system.

Diving into the field of register constructions, Attiya et al. in [2] showed a regular SWMR register construction that requires two-thirds of the processes in the system containing semi-Byzantine clients to be nonfaulty. For shared Read/Write atomic registers tolerant of Byzantine faults, we have a few notable results. The work of [9] by Mostéfaoui et al. in 2017, addresses the problem by proposing a Read/Write atomic memory system tolerant of Byzantine faults, where the tolerable limit of Byzantine processes is strictly lower than $\frac{n}{3}$. This limit was previously proven as necessary and sufficient in [6], proposed by Imbs et al. in 2014 and in [7]. Although these papers define an optimal limit of acceptable Byzantine processes in a distributed system for defining an atomic register, the proposed algorithms do not include a solution to secure the shared values. These construction techniques are based on replication and do not care about the fact that the Byzantines have unrestricted access to the content of the register. Managing access rights in such scenarios becomes impossible and so does securing shared values from a set of secured processes.

The Byzantine-tolerant privacy preservation techniques explored in this work are independent to cryptographic techniques, such as encrypting the written data. Therefore, both techniques can stack up to ensure stronger security guarantees. In asynchronous systems with high communication latency, cryptographic techniques are not always a good option and are often not scalable. The Byzantine-tolerant register construction works in the weakest trust model with very low resource constraints, whereas, cryptographic primitives and key management in cryptographic protocols often place strong assumptions on the computing model. However, it should be kept in mind that cryptographic techniques can be used as an additional layer of privacy preservation in Byzantine fault-tolerant privacy preservation. For instance, cryptographic techniques can be used within Byzantine fault-tolerant systems to secure communication channels or ensure data integrity. We do not consider the use of digital signatures or cryptographic techniques in the system model because of their high cost as well as hidden/implicit assumptions such as bounds on message latency which makes them inappropriate for truly asynchronous systems.

Problem Statement. The paper addresses the problem of constructing a privacy-preserving R/W register that tolerates Byzantine faults and restricts read access to a set of secured processes in the system.

Approach. The novelty of the proposed algorithm lies in the combination of two already existing approaches: secret sharing for privacy protection, and Byzantine-tolerant synchronization techniques to ensure linearizability.

To guarantee the privacy of shared information, we use Shamir's Secret Sharing scheme [11]. The scheme provides a mechanism to allow information to be shared between several trusted entities. This algorithm divides the secret data

into several subparts (shards) and then distributes the parts to all processes. Other information-sharing algorithms exist, but their complexity is higher, notably cryptography which is based on computing power. To do this, a polynomial P is generated randomly, with $P(0)$ being the secret information to be shared. The initiator transmits $P(x)$ (where $x \in [1, .., n]$) and n, the number of entities in the system. To reconstruct the initial information, a reader collects the parts of a minimum of entities for it to have enough parts of the information to be able to carry out Lagrange interpolation and thus reform the polynomial and find the secret $P(0)$. Since collaboration is mandatory to retrieve the secret, an entity cannot retrieve shared information without collaborating with others. If a process executes a read while not having read permissions, then it won't receive the requisite number of subparts following a request. This guarantees privacy because the reader won't be able to find the information shared initially.

To guarantee the linearizability of our R/W register, we use the replication approach first proposed by Attiya, Bar-Noy and Dolev for crash-prone systems [1]. Using this same approach, the update of the shards remains consistent. The sharing of information using Shamir's algorithm and the presence of Byzantine entities force us to implement additional protections to guarantee the linearizability of each operation carried out on our register and make it atomic and resist the Byzantines.

Contributions. The main contribution of this paper is an algorithm to implement a linearizable R/W privacy-preserving register tolerant to Byzantine faults satisfying $t < \frac{n}{7}$. The main challenge in this work is formally showing the robustness of the algorithm in fulfilling its *security* criteria. We propose a novel specification to define the security guarantee of our algorithm, based on the notions of *knowledge and common knowledge* [5]. The algorithm is further enriched with extensive proofs for its termination and correctness.

Organization. The paper is made up of 6 sections. Section 2 presents the computing model. Section 3 specifies the problem of building a shared register respecting privacy in its two parts, the consistency part and the privacy part. Section 4 presents the proposed algorithm that uses replication, and section 5 proves its correctness. Finally, Section 6 concludes the paper.

2 Model

Computing entities. The model is composed of a set of n sequential processes denoted as $p_1, p_2, ..., p_n$. Here, each p_i is associated with an identifier i that is known to all processes, and can be used in the code. These processes are asynchronous which implies that they all go at their own pace. No process can know the state of another process.

Byzantine Processes. The algorithm tolerates the presence of Byzantine processes. A Byzantine process exhibits behaviour that does not correspond to the

underlying algorithm it is meant to execute. It may crash, fail to send or receive messages, send arbitrary messages, and arbitrarily execute code. More generally, the Byzantine processes are free to produce all kinds of actions that could harm the smooth running of the algorithm. We assume that $t < \frac{n}{7}$ is the upper bound for the number of Byzantine processes. A process that does not exhibit Byzantine behaviour is called a *correct* process. Let $Correct$ be the set of all *correct* processes in the system.

The communication model. The processes communicate by sending and receiving messages through two-way communication channels. The communication network is complete: any process p_i can send a message to any other process in the system, including itself. Furthermore, if a process p_i receives a message m from a correct process p_j, it implies that p_j actually sent m to p_i. In other words, Byzantine processes cannot impersonate correct processes in the sending of messages. In a similar way, Byzantine processes cannot know the content of messages that are sent by correct processes, to correct processes. The channels are reliable, which implies that there is no message loss or corruption. The channels are *asynchronous*, which implies that the message transmission time is finite, but not upper bounded. However, if a process p_i and a process p_k both send a message to p_j, there is no guarantee of the order in which these messages are received at p_j.

FIFO Channels. We add as an additional hypothesis that the communication channels are FIFO (First In, First Out) between correct processes. This signifies that if a process p_i sends two messages a then b to a process p_j, p_j will receive these two messages in the order of their transmission. It is well-known that FIFO channels can be easily implemented on top of non-FIFO channels by using sequence number, so this hypothesis is done without adding computability power to the computing model.

Notation. The acronym $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$ designates the Byzantine Asynchronous Message Passing model where t processes can exhibit Byzantine behaviour and communication is done by message-passing.

Distributed histories.

Definition 1 (History). *The collection of the discrete points in time that make up the invocation and response events for every operation executed by every process in the system is referred to as the History of an execution.*

Definition 2 (Configuration). *A configuration of a system consists of the states of all processes and the state of the environment (values of shared variables and contents of all message channels).*

Definition 3 (Indistinguishability). *Two configurations C and C' are indistinguishable if a process p_i is in the same state in both C and C', and is denoted as $C' \leftrightarrow_i C$. We extend this definition to a pair of indistinguishable histories H and H' wrt p_i and denote it as $H' \leftrightarrow_i H$.*

3 Problem Specification

This paper considers the implementation of a privacy-preserving atomic SWMR register. The data structure is defined by the sequential specification described in Def. 4. We consider the definition of Byzantine linearizability from Shir Cohen and Idit Keidar [4], stated in Def. 5. The liveness property is the standard termination property recalled in Def. 6. Finally, we properly specify the security property ensuring privacy protection in the following subsection.

Definition 4 (Sequential specification of the R/W register). *We consider the classical SWMR register with its sequential specification containing the following two operations. Firstly, the* WRITE *operation, which is accessible only to the* writer *process. Secondly, the* READ *operation, accessible only to some select processes that have the* reading rights. *The set of processes having reading rights is denoted as canRead.* READ *returns the last written value if such a last value exists. If no value has been written yet, then the read returns a special value $\perp$ that cannot be written.*

Definition 5 (Byzantine linearizability). *A history H is linearizable with respect to an object O if there exists a sequential history H' (called a linearization of H) such that (1) after removing certain operations from H and by completing the others by adding corresponding responses, it contains the same calls and responses as H, (2) if an operation o returns before an operation o' begins in H then o appears before o' in H', and (3) H' satisfies the sequential specification of O.*
A history H is Byzantine linearizable with respect to an object O if there exists a history H' linearizable with respect to O, such that $H'|_{correct} = H|_{correct}$ (where $H|_{correct}$ designates the history where only operations performed by correct processes are taken into account). We say that an object is Byzantine linearizable, or simply linearizable, if all its executions are Byzantine linearizable.

Definition 6 (Termination). *Let p_i be a correct process.*

- *Each call to the procedure* WRITE() *by p_i terminates.*
- *Each call to the procedure* READ() *by p_i terminates.*

Privacy protection. Processes that do not have read permissions should not be able to read the content of the register. Additionally, a Byzantine process never possesses the reading rights to successfully execute the READ() operation.

Definition 7 (Guessing). *We assume that, once in each execution, some Byzantine process p_i can invoke a special atomic event, denoted by Guess(v), to signify that it has guessed the value v written by the writer p_w. To be valid, Guess(v) has to be indeed the last value written, as defined by Linearizability. Let H_v be the history H that only contains, on the one hand, the* WRITE *invocations and responses from p_w, and on the other hand, a* READ *operation by p_i, as well as a matching return(v) event, in the place of the Guess(v) event. We say that "p_i correctly guesses in H", denoted as $\Phi(H)$, the fact that H_v is linearizable.*

Obviously, it is impossible to ensure that p_i never correctly guesses the written value, since it may guess at random and succeeds by luck. However, in this situation, p_i does not *know* that the guessed value is correct. The notion of *knowledge* was formally defined by Dwork and Moses in [5]. Following their definitions, we define $K_i(\Phi(H))$ as the fact that Φ not only holds in H, but also in all executions indistinguishable to p_i from H.

Definition 8 (Knowledge).

$$K_i(\Phi(H)) \equiv \forall H', (H' \leftrightarrow_i H) \Rightarrow \Phi(H') \tag{1}$$

We can finally define our security property, as the fact that a Byzantine process can only know the written value if some readers are Byzantine as well.

Definition 9 (Privacy preservation for READ). *The following holds for every process p_i in the system.*

$$\forall H : \forall i : canRead \subset Correct \Rightarrow \neg K_i(\Phi(H))$$

4 Privacy Preserving Register Construction

The implementation of the privacy preserving atomic register is broken down into two constructions: WRITE and READ operations. The privacy preservation of the register is an abstraction of the union of these two operations.

4.1 Overview

On a high level, our proposed algorithm works in the following manner. The writer process p_w begins by creating a polynomial P to hide the information it wants to write in the register R. Next, p_w distributes shards constructed using P to all the processes in the system. p_w waits until enough processes have received their shards. The shards are distributed in such a manner that it enables an arbitrary reader process p to find the initial polynomial P and thereby, find (decrypt) $P(0)$, which is the value written by p_w. This scheme of information hiding using shards is inspired by Shamir's Secret Sharing [11]. The processes in the system then communicate with each other to find out if p_w has indeed emitted enough shards to re-construct P. If p has the confirmation of the required number of shards, then each process emits an ACK to signal p_w that it is possible to find the initial information because enough processes have validated the writing. When a process commits a write, it means that it received the shard from p_w and sent the ACK message.

To read the register, p sends a request to all processes to transmit their local shards. After receiving enough responses from the processes, it performs a search to reconstruct P. We use a sequence number scheme to maintain the most recent version of the shards, wherein, only the recent shards would be used to find P. Finally, p finds P and returns the value of $P(0)$. Alternatively, it returns $\perp$ as the

initial value of the register if sufficient shards are not emitted by p_w. This read write technique is inspired from the atomic register construction of Mostefaoui et al. as shown in [9]. Both the READ and WRITE operations are inspired from the asynchronous confirmation mechanism similar to the protocol for reliable broadcast proposed by [3].

Local variables. Each process p_i takes care of local variables indicated by the index i whose scope is the entire algorithm.

- $shards_i[1..]$: This is a local array of infinite size containing the list of subparts (shards) transmitted by the writer, arranged in the order of their reception at p_i. If the writer is correct then, the values stored in this table for p_i either correspond to $P(i)$ following a write, or to $\perp$.
- $acknoledged_i$: An integer representing the number of shards validated (acknowledged) by p_i.
- $reg_i[1..n][1..]$: This is an array of size n containing arrays of infinite size. This variable records the validated sub-parts (content in $shard_j$) and is transmitted by a process p_j during the read request. More precisely, $\forall j, k$, $reg_i[j][k]$ initialized to $\perp$ contains $P(j)$ of the writing k.

In addition to the above variables, let $\mathbb{Z}_t[X]$: the set of polynomials of degree t and coefficients in $\mathbb{Z}$.

Messages. This algorithm uses 8 different message types:

- SHARE$(shard, sn)$: Pass the $shard$ from the writer to a process.
- ECHO(sn): Informs other processes in the system that a process has received a shard from the writer.
- READY(sn): Informs other processes that the current process has received enough ECHO(sn) messages to be able to send the message ACK$(sn\ )$.
- ACK(sn): Informs the writer that the value he wrote can be read.
- COLLECT(rsn): Requests the shards received by each process.
- SUPPLY$(v[], rsn)$: Sends the list of shards received to the reader process.
- CONFIRM(k): Asks the process if they have at least the value read by the reader
- RATIFY(k): Processes send this message if they have at least the value read.

4.2 The Write Operation

Algorithm 1 illustrates the steps for the WRITE operation by a correct process p_i in the model $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$.

The writer p_w distributes the SHARE message to all processes in the system, including itself. This allows p_w to communicate the shard corresponding to every process and also the sequence number (denoted sn) of this write. All the processes are associated to an identifier, and hence, a process p_k receives a shard k corresponding to the value $P(k)$ where P is the polynomial generated by the writer p_w.

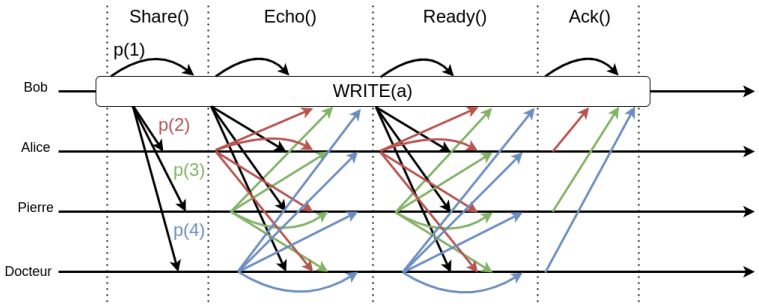

Fig. 1: Schematic explanation of the writing process illustrating the different phases that compose the WRITE operation. $p(i)$ denotes the shard being sent out from the writer Bob to the i'th process while attempting to write the value a to the register.

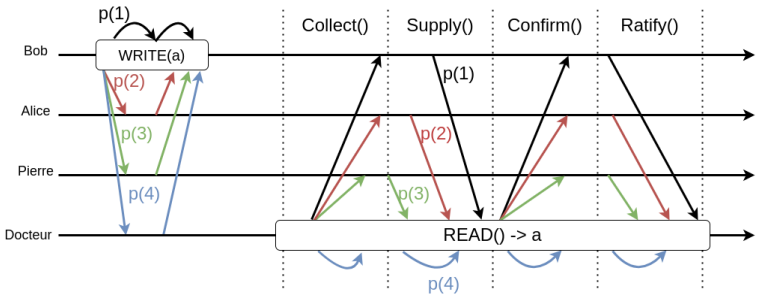

Fig. 2: The READ operation follows the WRITE and is internally composed of four phases of message collection. The specifications of the values carry the same meaning as in Figure 1.

Upon receipt of the SHARE message, the processes record the shard received in their variable $shards_i[sn]$ and emit ECHO(sn) message in order to inform all processes of receiving the shard. When a process receives $n - t$ ECHO(sn) messages or $5t + 1$ READY(sn) messages, it broadcasts the READY(sn) message. When a correct process p_i receives $6t + 1$ READY(sn) messages, it records sn in $acknoledged_i$ and sends the ACK(sn) message. The writer p_w then increments its sequence number and repeats the above for every write it wishes to perform. The sequence of actions performed is illustrated in Figure 1.

4.3 The Read Operation

A reading process p_i begins by resetting the reg_i variable. p_i then sends COLLECT(rsn_i) messages, with rsn_i being the reading counter of p_i. This counter ensures version control, wherein, there is no conflict between receptions of this message from different readings. p_i now waits to receive $n-t$ SUPPLY$(shards_i, rsn_r)$ messages.

When a process p_i receives the COLLECT(rsn) message, it ensures that p_r has read rights before transmitting to it all the shards validated by p_i in the SUPPLY$(shards_i[0..acknoledged_i], rsn)$ message. For each SUPPLY$(shards_i, rsn)$ message that p_i sends, p_r checks that the parameter rsn is equal to its variable

Algorithm 1: Implementation of WRITE in the linearizable privacy-friendly register of the $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$ model.

```
   procedure write(v) invoked by p_w is
1      let P ∈ Z_t[X] : P(0) = v;
2      for j from 1 to n do
3          send SHARE(P(j), sn_w) to p_j;
4      wait until (p_w has received at least n − t messages ACK(sn_w));
5      sn_w ← sn_w + 1;
   when receive SHARE(shard, sn) from p_j :
6      shards_i[sn] ← shard;
7      send ECHO(sn) to all processes;
8  when one of these conditions holds for the first time, for each sn:
9          • p_i has received ECHO(sn) from n − t different processes
10         • p_i has received READY(sn) from 5t + 1 different processes
11 do:
12     send READY(sn) to all processes;
   when receive READY(sn) from 6t + 1 process :
13     acknoledged_i ← sn;
14     send ACK(sn) to p_w;
```

rsn_r (if this corresponds to its current reading). If so, it saves $shard_i$ in its variable $reg_r[i]$. Once p_r receives $n - t$ SUPPLY messages, it increments rsn_r. Now p_r tries to find the polynomial created by p_w by the following steps: Let k be a loop variable which counts down from the largest size among the registers received by p_r till 1. p_r tries to find the polynomial created by p_w using Lagrange interpolation with $2t + 1$ points. If p_r does not find a polynomial then it returns $\perp$. Otherwise it uses the k for which p_r found a polynomial in order to send the message CONFIRM(k) to all processes. This message allows the reader to ensure that future readings will read at least value number k. Then it waits to receive $n - 2t$ RATIFY(k) messages. When the processes p_i receive the message CONFIRM(k) they wait until their variable $acknoledged_i$ is greater than or equal to k then they send the message RATIFY(k) to p_r. When p_r has received enough RATIFY messages, it can finish and returns $P(0)$ to find the initial value written by p_w. Figure 2 diagrammatically illustrates the above notion.

5 Correctness Proofs

We now rigorously show the correctness of our proposed algorithms. All missing proofs are in the Appendix.

Definition 10 (Validation sequence number). *Let $sn \in \mathbb{N}$, and p_i be a correct process. We say that p_i* validates *sn when p_i sends a message* ACK(sn) *and $shards_i[sn] \neq \perp$.*

Algorithm 2: Implementation of READ in the linearizable privacy-friendly register for the $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$ model.

```
   procedure read() invoked by any p_i with reading rights is
15 |  reg_i[1..n][1..] ← [[], ..., []];
16 |  send COLLECT(rsn_i) to all processes;
17 |  wait until (p_i has received at least n − t messages SUPPLY(_, rsn_i));
18 |  rsn_i ← rsn_i + 1;
19 |  for k from max_j |reg_i[j]| to 1 do
20 |  |  if ∃P ∈ Z_t[X] : |{j : P(j) = reg_i[j][k]}| > 2t then
21 |  |  |  send CONFIRM(k) to all processes;
22 |  |  |  wait until (p_i has received n − 2t RATIFY(k) messages);
23 |  |  |  return P(0);
24 |  return ⊥;
   when receive COLLECT(rsn) from p_j with reading rights :
25 |  send SUPPLY(shards_i[0..acknoledged_i], rsn) to p_j
   when receive SUPPLY(v[], rsn) from p_j :
26 |  if rsn = rsn_i then reg_i[j] ← v;
   when receive CONFIRM(k) from p_j :
27 |  wait until (acknoledged_i ≥ k);
28 |  send RATIFY(k) to p_j;
```

Definition 11 (Timestamp). *For each operation o, we set a timestamp $hd(o)$ of o as follows. If o is a write by p_w, then $hd(o) = sn_w$ at the end of line 5. If o is a reading by p_i then $hd(o) = k$ at line 23 or $hd(o) = \perp$ at line 24. In other words, $hd(o)$ is equivalent to the sequence number that is read or written by o.*

Lemma 1. *If a correct process sends the message* ACK*(sn) then all correct processes send the message* ACK*(sn).*

Proof. Suppose a correct process p_i sends an ACK(sn) message at line 14. Hence, p_i received at least $6t+1$ messages READY(sn), of which at least $5t+1$ were sent by correct processes. So, all correct ones will receive these messages and emit a READY(sn) message at line 12. Like $n - t \geq 6t + 1$, at least $6t + 1$ messages READY(sn) are sent by correct ones, all correct ones send a message ACK(sn) at line 14. □

Lemma 2 (Write termination). *Any call by a correct process to the write(v) operation terminates.*

Proof. Suppose that p_w, a correct process calls the write(v) operation and that it does not terminate. The lines 1-3 and 5 end by definition. So if the execution of write(v) by p_w does not complete, this means that p_w has received less than $n-t$ ACK(sn) messages. At line 3, p_w will send SHARE(sn) messages to all processes since it is correct. Which implies that the correct $n-t$ processes will receive the

message SHARE(sn) and transmit the message ECHO(sn) line 7. So at least $t+1$ correct processes will receive $n-t$ ECHO(sn) messages and will therefore transmit the READY(sn) message to line 12. So all other correct processes will eventually pass READY(sn). This implies that at least 1 correct process will receive $6t+1$ READY(sn) messages and it will therefore send the message ACK(sn) line 14. However, lemma 1 tells us that if one correct process sends the ACK(sn) message, then all correct processes will eventually send it. The process p_w will therefore necessarily receive $n-t$ ACK(sn) messages and complete its execution of the WRITE(v) operation. This is in contradiction with the initial assumption, which implies that the property is true. □

Lemma 3 (End of readings). *Every invocation of the* READ() *operation by a correct process terminates.*

Proof. Suppose that a correct process p_i invokes the READ() operation and does not terminate. Note that lines 15-16, 18-21 and 23-24 must terminate by definition. So, if p_i's READ() execution does not complete, it is because it has not received $n-t$ SUPPLY(-,rsn_i) messages at line 17 or $n-2t$ RATIFY(k) messages at line 22. Let us look at the cases separately.
If p_i has not received $n-t$ SUPPLY(-,rsn_i) messages then it must be true that at least one correct process has not received the message COLLECT and has not not executed the line 25. However, at line 16, p_i sent a COLLECT to all processes and as the channels are secured, each correct process must have received the message. Subsequently, each correct process must have sent SUPPLY messages to p_i and therefore executed line 25, which is a contradiction.
If p_i has not received $n-2t$ RATIFY messages, after having sent CONFIRM(k) to all the processes then it must be true that at least one correct process is blocked at line 27. If p_i sent the message CONFIRM(k) at line 21 to all processes then a particular process must have necessarily received $2t+1$ message SUPPLY as validated by line 20. This implies at least $2t+1$ processes have validated k. Hence, at least one correct process has issued the message ACK(k). According to Lemma 1, if a correct process sends the message ACK(k) then all correct processes will send ACK(k). Hence, $\forall p_j, acknoledged_j \geq k$ for a correct process p_j and p_j would be able to send the message RATIFY(k). Process p_i will eventually receive $n-2t$ RATIFY(k) messages, so p_i will not get stuck at line 22.
Both cases are in contradiction with the initial hypothesis and hence our result follows. □

Theorem 1 (Termination of the algorithm). *Any use of the algorithm by a correct process terminates.*

Proof. Lemmas 2 and 3 show us that calls to the READ() and WRITE(v) procedures terminate. This implies that the algorithm terminates. □

Theorem 2 (Privacy preservation of READ from Byzantines). *The following holds for every process* p_i *and history* H *admitted by Algorithm 1 and 2.*

$$\forall H : \forall i : canRead \subset Correct \Rightarrow \neg K_i(\Phi(H)) \tag{2}$$

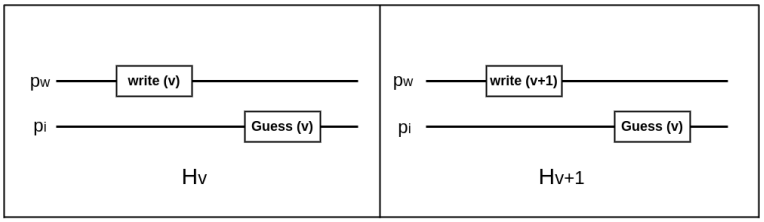

Fig. 3: The construction of two indistinguishable histories H_v and H_{v+1} is shown. These two histories are indistinguishable for p_i.

Proof. Let H be a history admitted by the algorithms, and let p_i be a process. We suppose that $canRead \subset Correct$. Let us prove that $\neg K_i(\Phi(H))$. Recalling Def. 8, it means that

$$\exists H' : H \leftrightarrow_i H' \wedge \neg\Phi(H'). \tag{3}$$

Let v be the value such that the operation $Guess(v)$ exists in H. We build H' by transforming all WRITE(v) operations in H as follows.

- The WRITE(v) invocation is replaced by a WRITE$(v+1)$ invocation in H'.
- Let P be the polynomial chosen by p_w in H. By the Lagrange Interpolation theorem, there exists a polynomial P' of degree t such that $P'(0) = v+1$ and $P'(j) = P(j)$ for all Byzantine processes p_j. We replace all SHARE$(P(j), sn_w)$ messages by messages SHARE$(P'(j), sn_w)$, and we adjust the SUPPLY messages, as well as the return values of the READ operations according to the algorithms. Figure 3 illustrates this construction diagrammatically.

Let us prove that H and H' are indistinguishable to all Byzantine processes p_j. Only SHARE and SUPPLY messages differ in H and H'; by the definition of P', Byzantine processes receive that same SHARE messages in H and H'; and since $canRead \subset Correct$, then Byzantine processes do not receive SUPPLY messages from correct processes in either H or H', since their COLLECT messages are ignored by correct processes at Line 25. Therefore, all Byzantine processes receive the same messages from correct processes in H and H', so H and H' are indistinguishable to all of them. In particular, H and H' are indistinguishable to p_i, so $K_i(\Phi(H))$ is false. Moreover, the history H' contains an operation $Guess(v)$ but no operation WRITE(v), so $\Phi(H')$ is false, which concludes the proof. □

Lemma 4. (Proof in the Appendix) *Assume the correct writer. If at least* $4t+1$ *correct processes have validated the sequence number* $hd(o_1)$ *before a read* o_2 *begins, then* $hd(o_2) \geq hd(o_1)$.

Lemma 5. (Proof in the Appendix) *If the writer is correct and at least* $n - 2t$ *correct processes sent the message* ACK*(sn), then at least* $4t+1$ *correct processes committed* sn.

Lemma 6 (Write after read). *Given 2 correct processes* p_i *and* p_j*, if* p_j *finishes reading* o_1 *before* p_i *begins writing* o_2 *then* $hd(o_1) < hd(o_2)$.

Lemma 7 (Read after write). *Let a correct process p_i complete a write o_1 before a correct process p_j begins a read o_2 then $hd(o_2) \geq hd(o_1)$.*

Lemma 8 (Read after read). *Let p_i be a correct process that completes a read operation o_1 before a correct process p_j begins a read operation o_2. Then $hd(o_1) \leq hd(o_2)$.*

Lemma 9. (Proof in the Appendix) *Given a read o_1 and a write o_2, if $hd(o_1) = hd(o_2)$ then o_1 will return the argument of o_2.*

Lemma 10 (Linearizability for the correct writer). *Assume the correct writer. There is a total order $<$ on all writes and reads made by the correct ones, such that:*

- *execution of operations in order $<$ respects the sequential specification*
- *if e ends before e' begins, then $e < e'$*

Lemma 11 (Linearizability for the Byzantine writer). *Let us assume the Byzantine writer. Any history accepted by the algorithm 1 is Byzantine linearizable for a Read/Write register*

Proof. We construct a history H' made up of all the readings of H made by corrects in which we add a writing of the value read before each reading. Hence, H' is linearizable and coincides with H on all readings. □

Theorem 3 (Linearizability). *Algorithm 1 implements a register respecting Byzantine linearizability.*

Proof. Lemmas 10 and 11 detail the two cases following the behavior of the writer. □

6 Conclusion

In this article, we implemented a linearizable SWMR register tolerant of Byzantine faults and preserving privacy. To do this, we used Shamir's Secret Sharing algorithm which allowed us not to share the information in its entirety and to guarantee that a process that does not have reading rights would be unable to read the register. Our register algorithm can support a maximum of no more than one-seventh of the total number of Byzantines processes in the system. As an important contribution to this work, we rigorously prove how our register construction achieves privacy preservation even when Byzantines mimic READ operations without having access rights.

A few open questions remain for this work. It is pertinent to ask if the bound of $t < \frac{n}{7}$ for our algorithm is optimal. Additionally, it would be nice to explore the possibility and benefits of increasing security by combining several approaches, such as adding cryptographic techniques when passing shards.

References

1. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.
2. Hagit Attiya and Amir Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pages 371–378. IEEE, 2003.
3. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.
4. Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. *arXiv preprint arXiv:2102.10597*, 2021.
5. Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.
6. Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Reliable shared memory abstraction on top of asynchronous byzantine message-passing systems. In *Structural Information and Communication Complexity: 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings 21*, pages 37–53. Springer, 2014.
7. Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *Journal of Parallel and Distributed Computing*, 93:1–9, 2016.
8. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.
9. Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60:677–694, 2017.
10. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
11. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

7 Appendix

The missing proofs from Section 5 are detailed here.

Lemma 4 *Assume the correct writer. If at least $4t+1$ correct processes have validated the sequence number $hd(o_1)$ before a read o_2 begins, then $hd(o_2) \geq hd(o_1)$.*

Proof. Assume the correct writer, that at least $n-3t$ correct processes have committed the sequence number $hd(o_1)$ by the time a process p_i begins a read o_2. When p_i executes the line 17, it receives $n-4t$ messages from correct processes p_j of the form SUPPLY(v_j, rsn_i) which have already committed $hd(o_1)$. Let Π_S denote the set of these $n-4t$ processes.

Let us prove that, for all $p_j \in \Pi_S$, we have $|v_j| \geq a$. Let $p_j \in \Pi_S$. As p_j has already validated $hd(o_1)$ (line 14), it executed line 13 previously. To be able to execute these lines, p_j must have received $6t+1$ times the message READY$(hd(o_1))$ coming from different processes, including at least $5t+1$ correct processes. This implies that at least $5t+1$ correct processes received the message ECHO$(hd(o_1))$ from $n-t$ different processes where they received the message READY$(hd(o_1))$ of $5t+1$ different process to be able to emit READY$(hd(o_1))$ at line 12. This implies that at least $4t+1$ correct processes emit the message ECHO$(hd(o_1))$ line 7. To do this, they had to receive the message ECHO$(hd(o_1))$ coming from $n-t$ different processes, including at least $n-2t$ correct processes. If these $n-2t$ correct processes sent ECHO$(hd(o_1))$ line 7, they necessarily received a shard from the writer process and therefore they recorded this shard at line 6. So $|v_j| \geq hd(o_1)$ according to the line 25.

When p_i executes the line 20, it will be able to find the value of the entry having the sequence number $hd(o_1)$. Indeed, p_i has $n-5t$ messages from correct processes p_j of the form SUPPLY(v_j, rsn_i) which have already validated $hd(o_1)$. p_i must manage to find a polynomial with $2t+1$ given shards. Now, if p_i has $n-5t$ messages from correct processes, it therefore has at least $2t+1$ because $t < \frac{n}{7}$. The writer being correct, he therefore created a line polynomial 1 then he sent a value associated with each process using this polynomial in the message SHARE$(P(j), sn_w)$. As shown previously, all p_j stored this value and then transmitted it to p_i. When p_i finishes its read operation (see lemma 3), it will return line 23 the value written by the writer process. □

Lemma 5 *If the writer is correct and at least $n-2t$ correct processes sent the message* ACK*(sn), then at least $4t+1$ correct processes committed sn.*

Proof. Let p_i be a correct process which completes a write with sequence number sn. The process p_i therefore received $n-t$ messages ACK(sn) line 4. This implies that all correct processes p_j sent the message ACK(sn) line 14 and so they stored sn in the variable $acknoledged_j$ at line refline:incrAcknoledged. In order to be able to execute these lines, the p_j processes had to receive READY(sn) messages from $6t+1$ different processes, of which at least $5t+1$ were correct. For a correct

process to emit the message READY(sn) line 12, it must have received $n-t$ message ECHO(sn) or $5t+1$ messages READY(sn). At a minimum, there are therefore $4t+1$ correct processes which emit the message READY(sn) after having received the messages ECHO(sn). For a correct process p_j to send the message ECHO(sn) line 7, it must have received the message SHARE($shard, sn$). The process p_i being correct, it will transmit a message SHARE to all the processes in the system at line 3. So $n-t$ processes sent the message ECHO(sn) of which at least $n-2t$ are correct. All correct processes will be able to send the READY(sn) message. Correct processes will therefore receive at least $6t+1$ message READY(sn). They will therefore be able to transmit the message ACK(sn). Let $Q1$ be the set of processes that sent the message ECHO(sn), and let $Q2$ be the set of processes that sent the message ACK(sn). We have $|Q1| \geq n-t$ and $|Q2| \geq n-t$. We have:

$$\begin{aligned} |Q1 \cap Q2| &= |Q1| + |Q2| - |Q1 \cup Q2| \\ &\geq n - t + n - t - n && (\because |Q1 \cup Q2| \leq n) \\ &\geq n - 2t \\ &> 5t && (\because n > 7t) \\ |Q1 \cap Q2| &\geq 5t + 1. \end{aligned}$$

In particular, $Q1 \cap Q2$ contains at least $|Q1 \cap Q2| - t \geq 4t+1$ correct processes p_j which have $shard_j[sn] \neq \perp$ and which sent the message ACK(sn), i.e. the definition of validated. □

Lemma 6 (Write after read) *Given 2 correct processes p_i and p_j, if p_j finishes reading o_1 before p_i begins writing o_2 then $hd(o_1) < hd(o_2)$.*

Proof. Consider a correct process p_j which performs a read operation o_1 and a correct process p_i which performs a write operation o_2 such that $hd(o_1) < hd(o_2)$ and suppose that p_j returns the value associated with the writing o_2.

For p_j to be able to return this value, at line 20 it must have succeeded in finding a polynomial with $2t+1$ shards of size greater than or equal to $hd(o_2)$ emitted by correct processes p_k. The information contained in $reg_j[k]$ being different from $\perp$, we have $reg_j[k][hd(o_2)] = shards_k\,[hd(o_2)]$, which was edited at the line 6 when p_k received the message SHARE($shard, hd(o_2)$) sent by the writer process at line 3. This therefore implies that the write $hd(o_2)$ has started if o_1 returns the value of the write o_2 which implies that $hd(o_1) \geq hd(o_2)$ by definition of hd. □

Lemma 7 (Read after write) *Let a correct process p_i complete a write o_1 before a correct process p_j begins a read o_2 then $hd(o_2) \geq hd(o_1)$.*

Proof. Let there be 2 correct processes p_i and p_j, if p_i completes an operation o_1 before p_j begins reading o_2. For p_i to finish writing, it had to wait to receive $n-t$ messages ACK(sn) (line 4) from different processes, including at least $n-2t$ correct processes. So according to lemma 5, there are at least $4t+1$ correct processes which validated the sequence number $hd(a)$. So according to lemma 4, $hd(o_2) \geq hd(o_1)$. □

Lemma 8 (Read after read) *Let p_i be a correct process that completes a read operation o_1 before a correct process p_j begins a read operation o_2. Then $hd(o_1) \leq hd(o_2)$.*

Proof. Let there be 2 correct processes p_i and p_j, if p_l completes its read operation o_1 which implies that at least $4t + 1$ correct processes have validated the write with $hd(o_1)$ according to lemma 5. If now p_j begins a read o_2, then it has received shards from at least $n - t$ processes and has determined line 19 that $hd(o_2)$ was the most recent validated by the greatest number of people. According to Lemma 4, $n - t$ processes guarantee at least $3t + 1$ processes have validated $hd(o_1)$. So, by removing t byzantines, we have $3t + 1 - t = 2t + 1$. We can still find $P(0)$ because at least $2t + 1$ shards are necessary. Knowing that at least $4t + 1$ processes have validated the writing o_1, we can conclude that $hd(o_2)$ is either equal to $hd(o_1)$ or greater and that corresponds to a new written value. □

Lemma 9 *Given a read o_1 and a write o_2, if $hd(o_1) = hd(o_2)$ then o_1 will return the argument of o_2.*

Proof. Let two correct processes p_i and p_j such that p_i performs the write operation o_2 and p_j performs the read operation o_1. Suppose $hd(o_1) = hd(o_2)$. Let P_1, the polynomial such that $P_1(0)$ is returned line 23 by p_i, and P_2, the polynomial chosen by p_j at line 1.

There exists at least $t + 1$ correct processes p_k such that $P_1(k)$ is equal to $reg_j[k][hd(o_1)]$ at line 20. The information contained in $reg_j[k]$ being different from $\perp$, we have $reg_j[k][hd(o_1)] = shards_k\,[hd(o_1)]$, which was edited at line 6 when p_k received the message SHARE$(shard, hd(o_1))$ sent by the writer process at line 3. We therefore have $P_1(k) = reg_j[k][hd(o_1)] = P_2(k)$.

Finally, there exist at least $t+1$ different values of k such that $P_1(k) = P_2(k)$. By uniqueness of Lagrange interpolation for polynomials of degree at most t, we have $P_1 = P_2$, therefore the value $P_1(0)$ returned by reading o_1 is equal to the argument $P_2(0)$ of writing o_2. □

Lemma 10 *Assume the correct writer. There is a total order $<$ on all writes and reads made by the correct ones, such that:*

- *execution of operations in order $<$ respects the sequential specification*
- *if e ends before e' begins, then $e < e'$*

Proof. Consider an execution of the algorithm 1. We define the binary relation $\rightarrow$ between operations o_1 and o_2 by: $o_1 \rightarrow o_2$ if, either 1) o_1 is finished before o_2 begins (denoted: $o_1 \rightarrow_1 o_2$), or 2) $hd(o_1) < hd(o_2)$ (noted: $o_1 \rightarrow_2 o_2$), or 3) o_1 is a write, o_2 is a read and $hd(o_1) \leq hd(o_2)$ (noted: $o_1 \rightarrow_3 o_2$), or 4) $o_1 = o_2$ (noted: $o_1 \rightarrow_4 o_2$).

Note that if $o_1 \rightarrow o_2$ then $hd(o_1) \leq hd(o_2)$ and if moreover o_2 is a different writing from o_1, then $hd(o_1) < hd(o_2)$. This is true by definition for $\rightarrow_2$ and $\rightarrow_3$. For $\rightarrow_1$, if it is a read followed by a read, it is true by lemma 8. If it is a write followed by a read, it is true by lemma 7. If it is a read followed by a write, it

is true by lemma 6. If it is a write followed by a write then the line 5 and the definition of $hd()$ are true.

The relation $\rightarrow$ is reflexive by definition of $\rightarrow_4$.

Let us prove the transitivity of $\rightarrow$. Consider three operations o_1, o_2 and o_3 such that $o_1 \rightarrow o_2 \rightarrow o_3$. Let us prove that $o_1 \rightarrow o_3$. The following cases must be distinguished.

1. Case $o_1 \rightarrow_4 o_2$ or $o_2 \rightarrow_4 o_3$: we have $o_1 = o_2 \rightarrow o_3$ or $o_1 \rightarrow o_2 = o_3$, therefore $o_1 \rightarrow o_3$.
2. Case $o_1 \rightarrow_2 o_2$ or $o_2 \rightarrow_2 o_3$: we have $hd(o_1) < hd(o_2) \leq hd(o_3)$ or $hd(o_1) \leq hd(o_2) < hd(o_3)$, so $o_1 \rightarrow_2 o_3$.
3. Case $o_1 \rightarrow_1 o_2 \rightarrow_1 o_3$: o_1 ends before o_2 begins and o_2 ends before o_3 begins. So o_1 is finished before o_3 begins. So, $o_1 \rightarrow_1 o_3$.
4. Case $o_1 \rightarrow_1 o_2 \rightarrow_3 o_3$: o_2 is a write. If o_1 is a read, we have $hd(o_1) < hd(o_2)$ by lemma 6. If o_1 is a write then $hd(o_1) < hd(o_2)$ by line 5. In both cases, $hd(o_1) < hd(o_3)$, so $o_1 \rightarrow_2 o_3$.
5. Case $o_1 \rightarrow_3 o_2 \rightarrow_1 o_3$: We have $hd(o_1) \leq hd(o_2) \leq hd(o_3)$. If $hd(o_1) < hd(o_3)$, then $o_1 \rightarrow_2 o_3$. Otherwise $hd(o_1) = hd(o_3)$. Since o_1 is a write (by definition of $\rightarrow_3$) o_3 cannot be a write because of line 5. So o_3 is a reading, so $o_1 \rightarrow_3 o_3$.
6. Case $o_1 \rightarrow_3 o_2 \rightarrow_3 o_3$: Impossible because o_2 would be both read and write.

Let us prove the antisymmetry of $\rightarrow$. Let two operations o_1 and o_2 be such that $o_1 \rightarrow o_2 \rightarrow o_1$. Let us prove that $o_1 = o_2$. The following cases must be distinguished.

1. Case $o_1 \rightarrow_4 o_2$ or $o_2 \rightarrow_4 o_1$: we have $o_1 = o_2$.
2. Case $o_1 \rightarrow_2 o_2$ or $o_2 \rightarrow_2 o_1$: we have $hd(o_1) < hd(o_1)$, which is absurd.
3. Case $o_1 \rightarrow_1 o_2 \rightarrow_1 o_1$: Paradoxical because o_1 would have finished before having started.
4. Case $o_1 \rightarrow_1 o_2 \rightarrow_3 o_1$: By definition of $\rightarrow_3$, o_2 is a write and o_1 is a read, and by definition of $\rightarrow_1$, o_1 ends before until o_2 begins. According to lemma 6, we have $hd(o_1) < hd(o2)$. This contradicts the definition of $\rightarrow_3$ that $hd(o_1) = hd(o_2)$.
5. Case $o_1 \rightarrow_3 o_2 \rightarrow_1 o_1$: We have $o_2 \rightarrow_1 o_1 \rightarrow_3 o_2$, which brings us back to the previous case.
6. Case $o_1 \rightarrow_3 o_2 \rightarrow_3 o_1$: Impossible because o_2 would be both read and write.

The relation $\rightarrow$ is therefore a partial order relation, which contains real-time, by definition of $\rightarrow_1$.

Assume the correct writer process named p_w, all reads done by process p_k such that $hd(o_{k0})$ is equal to $\perp$ are placed before the first write o_1 such that $\forall o_{k0}, o_{k0} \rightarrow_2 o_1$. Then, lemma 9 tells us that if a read o_k takes place between the write o_n and the write o_{n+1} then $hd(o_n) = hd(o_k)$. We therefore obtain the order $o_n \rightarrow_1 o_k$ and $o_k \rightarrow_2 o_{n+1}$. This creates a partial order for us that we can extend to a total order.

The binary relation $\rightarrow$ can be applied to a total order which respects real time thanks to $\rightarrow_1$ and each read returns the initial value if the timestamp is 0 or the value written by the previous write since sn is updated line 5 jointly with k line 20 thanks to $\rightarrow_2$ and $\rightarrow_3$. The execution considered is therefore linearizable. □