The Synchronization Power of Auditable Registers

Hagit Attiya
Technion, Haifa, Israel

Antonella Del Pozzo
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Alessia Milani
Laboratoire d'Informatique et Systèmes, Aix-Marseille Université and CNRS, Marseille, France

Ulysse Pavloff
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Alexandre Rapetti
Aix-Marseille Université, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

Auditability allows to track all the read operations performed on a register. It abstracts the need of data owners to control access to their data, tracking who read which information. This work considers possible formalizations of auditing and their ramification for the possibility of providing it.

The natural definition is to require a linearization of all write, read and audit operations together (*atomic* auditing). The paper shows that atomic auditing is a powerful tool, *as it can be used to solve consensus*. The number of processes that can solve consensus using atomic audit depends on the number of processes that can read or audit the register. If there is a single reader or a single auditor (the writer), then consensus can be solved among two processes. If multiple readers and auditors are possible, then consensus can be solved among the same number of processes. This means that strong synchronization primitives are needed to support atomic auditing.

We give implementations of atomic audit when there are either multiple readers or multiple auditors (but not both) using primitives with consensus number 2 (swap and fetch&add). When there are multiple readers *and* multiple auditors, the implementation uses compare&swap.

These findings motivate a weaker definition, in which audit operations are not linearized together with read and write operations (*regular* auditing). We prove that regular auditing can be implemented from ordinary reads and writes on atomic registers.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies

Keywords and phrases Auditability, atomic register, fault tolerance, consensus number

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.4

Related Version *Full Version*: `https://arxiv.org/abs/2308.04646`

Funding *Hagit Attiya*: Partially supported by the Israel Science Foundation (grant 22/1425).
Alessia Milani: Partially supported by CNRS-INS2I, Project PRIDE 2023.
Alexandre Rapetti: Partially supported by CNRS-INS2I, Project PRIDE 2023.

1 Introduction

Outsourcing storage capabilities to third-party distributed storage is a common practice for both private and professional users. It helps to circumvent local space constraints, dependability, and accessibility limitations. Unfortunately, this means having to trust the distributed storage provider on data integrity, retrievability, and confidentiality. Those issues are underscored by relentless attacks on data storage servers [2], which increased the awareness to data confidentiality and sovereignty and lead to a recent worldwide deployment of data protection regulations [1, 10, 19]. Even in secure storage systems where *access control* policies regulate who can access the data, an unauthorized user can access data either due to a misconfiguration of the access control system or in the occurrence of a data breach [17, 24].

As a result, data owners are increasingly concerned about who accesses their data. This makes *auditability* – the systematic tracking of who has read data in storage systems and the information it has observed – an important feature.

A *register* is an abstraction for distributed storage that provides read and write operations to the clients. An *auditable register*, introduced by Cogo and Bessani [5], is a register enriched with an *audit* operation. An audit operation indicates who performed read operations on it, and which values they have read. Auditability is defined in terms of two properties: *completeness* ensures that readers that access data are detected, while *accuracy* ensures that readers who do not access data are not incriminated.

In this work, we formalize the correctness of auditable registers in terms of their high-level operations (read, write and audit). We first formalize a natural extension of an atomic register, called *atomic register with atomic audit* where all the operations (including audit) appear to happen in a sequential order that respects their real-time order.

We show that an *atomic register with atomic audit* is a powerful abstraction, *because it has a greater consensus number than an ordinary atomic register*. Recall that the *consensus number* [12] of object type X is m if m is the largest integer such that there exists an asynchronous consensus algorithm for m processes, up to $m-1$ may crash, using only shared objects of type X and read/write registers.

We present a wait-free algorithm that solves consensus among *two processes*, using an atomic register with atomic audit where only one process (the writer) can perform audit operations. This stands in contrast to the well-known result [12] that atomic read/write registers cannot be used to solve wait-free consensus among two processes. We then show that when m processes can read and audit the register, it is possible to solve consensus among m processes.

These results indicate that base objects stronger than read/write registers are needed to implement atomic audit, motivating our implementations of an atomic register with atomic audit. When there is either a single auditor or a single reader, we use base objects with consensus number 2 (*swap* and *fetch&add*).

Specifically, we first present a simple algorithm for a single-reader atomic auditable register with atomic audit, where the writer is the only process that can execute the audit operations. The writer needs to atomically retrieve who read the previously written value while writing a new value. With a single reader, this can be easily ensured by using *swap* primitives: to read a value, the reader atomically swaps it with a special value. If the writer retrieves this special value when writing a new value, then it is aware of the value read.

Extending this idea to multi-readers is challenging, since readers might be swapping each other's values. We propose a solution that uses a single shared object accessed with *swap* and *fetch&add* primitives. The n low-order bits of the value stored in this object, where n is the number of readers, are used to indicate which readers have accessed the value stored in the high-order bits. Each reader is assigned a unique bit, which is set to 1 when the reader accesses the value. Then, when the writer writes a new value it can learn who read its previous value by checking the values of the low-order bits it retrieved from the value atomically read while writing the new value. A similar algorithm allows to support multiple auditors, but only a single reader, also using *swap* and *fetch&add*.

When there are multiple readers and auditors, we use *compare&swap*, which has consensus number ∞, in addition to *fetch&add*.

Taken together, our results mean that atomic audit cannot be implemented using only reads and writes, and stronger primitives (with consensus number > 1) should be used.

We then investigate the possibility to extend an atomic register with a useful audit operation without relying on strong synchronization primitives. This weaker abstraction is called a *regular* audit, and roughly speaking, differs from the previous one in not having the audit operations linearized together with read and write operations. In particular, a regular audit operation *aop* may not detect a read operation that is concurrent with it, even though the read has to be linearized before a write that completes before the invocation of *aop*. Our final result is a single-writer multi-reader atomic register with multi-auditor *regular* audit, using only atomic read and write operations, whose consensus number is 1.

Related Work. To the best of our knowledge, only two papers [5, 6] formally study auditability of read operations. Cogo and Bessani [5] were the first to formalize the notion of auditable register. Their definition is tailored for auditable register implementations on top of a shared memory model where some base objects can be faulty, i.e., they can omit to record readers or they can record nonexistent read operations. They present an algorithm to implement an auditable register, here read and write operations provide regular semantics using $n \geq 4f + 1$ atomic read/write shared objects, f of which may be faulty. Because of their failure model, their high-level register implementation relies on information dispersal schemes, where the input of a high-level write is split into several pieces each written in a different low-level shared object. It implies that a process can read a written value only if it collects enough pieces of information, making its read operation detectable. Their definition of completeness and accuracy for the auditable register relies on the notion of *effectively read*, which they formalize to capture the fact that the process executing the high-level read operation could have collected enough pieces of information and be able to retrieve the value, even if the read operation does not return.

In asynchronous message-passing systems where f processes can be Byzantine, Del Pozzo et al. [6] study the possibility of implementing an atomic auditable register with the accuracy and completeness properties, as defined by Cogo and Bessani, with fewer than $4f + 1$ servers. They prove that without communication between servers, auditability requires at least $4f + 1$ servers, f of which may be Byzantine. They also show that allowing servers to communicate with each other admits an auditable atomic register with optimal resilience of $3f + 1$. Their implementation also uses information dispersal scheme to deal with Byzantine processes. In contrast, we consider a classical shared memory model where processes fail by crashing. Also, our definition of auditable register is not tailored for a specific class of implementations, since it is stated in terms of high-level operations.

Most of the other works on auditing protocols for distributed storage focus on data integrity [7, 8, 13, 15, 16, 21–23], while our work focuses on auditing *who* has read *which* data.

When faulty processes are malicious, *accountability* [3, 4, 11, 20] aims to produce proofs of misbehavior in instances where processes deviate, in an observable way, from the prescribed protocol. This allows the identification and removal of malicious processes from the system as a way to clean the system after a safety violation. In contrast, auditability logs the processes actions and let the auditor derive conclusions on the processes behavior.

Frey, Gestin and Raynal [9] investigate the synchronization power of *AllowList* and *DenyList*: intricate append-only lists where AllowList contains resources that processes can access, while DenyList includes resources that processes cannot access. They prove the consensus number of AllowList is 1, while the consensus number of DenyList is equal to the number of processes that can access resources not listed in the DenyList. AllowList and DenyList control accesses, while an auditable register tracks (read) accesses; further discussion of the relation between an auditable register and DenyList appears in Section 8.

2 Model

We consider a standard shared-memory model where crash-prone asynchronous processes communicate through registers, using a given set of primitive operations. The primitive operations (sometimes called just primitives) include ordinary *read* and *write*, as well as *swap*, *fetch&add* and *compare&swap*.

A $swap(v)$ primitive atomically writes v to the register and returns its previous value. A $fetch\&add(a)$ primitive atomically writes the sum of a and the current value of the register into the register and returns its previous value. A $compare\&swap(old, new)$ primitive is an atomic conditional write: the write of *new* is executed if and only if the value of the register is *old*; a Boolean value is returned that indicates if the write was successful or not.

The auditable atomic register is an extension of an ordinary atomic read/write register [14]. It is formally defined in the next section. We only consider *single-writer* registers, where each register can be written by a single process.

An auditable register implementation specifies the state representation of the register and the algorithms processes follow when they perform the read, write and audit operations. Each operation has an invocation and a response event.

An execution is a sequence of steps performed by processes as they follow their algorithms, in each of which a process applies at most a single primitive to the shared memory (possibly in addition to some local computation).

A *history* H is a sequence of invocation and response events; no two events occur at the same time. An operation is *complete* in history H, if H contains both the invocation and the matching response for this operation. If the matching response is missing, the operation is *pending*. An operation *op precedes* another operation op' in H if the response of *op* appears before the invocation of op' in H; we also say that op' *follows op*.

A history is *sequential* if each operation invocation is immediately followed by the matching response, by the same process on the same object. For a given history H, $\mathsf{complete}(H)$ is the set of histories obtained from H by appending zero or more responses to some pending invocations and discarding the remaining pending invocations.

We consider *wait-free* implementations which ensures that a non faulty process completes an operation within a finite number of its own steps.

3 Definitions of Auditable Register

An auditable register supports three operations: $R.write(v)$ which assigns value v to the register R, $R.read()$ which returns the value of the register, and $R.audit()$ which reports the set of all values read in the register and by whom. Specifically, an audit operation returns a set of pairs (p, v), each corresponding to a read invoked by process p that returned v. In the following, we define two specifications for *audit* operations, exploring different semantics of their interaction with concurrent read and write operations.

Intuitively, *atomic audit* provides the illusion that all the read, write, and audit operations appear as if they have been executed sequentially.

▶ **Definition 1** (Atomic audit). *A history H is* atomic with atomic audit *if there is a history H' in* $\mathsf{complete}(H)$ *and a sequential history π that contains all operations in H' such that:*

1. *If operation op_1 precedes operation op_2 in H, then op_1 appears before op_2 in π.* Informally, π respects the real-time order of non-overlapping operations.
2. *Every read in π returns the value of the most recent preceding write, if there is one, and the initial value, otherwise.* Informally, the history π respects the semantics of an atomic read / write register.

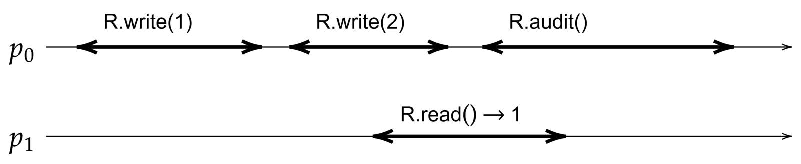

Figure 1 A scenario where a regular audit can return either $\emptyset$ or $(p_1, 1)$, while an atomic audit must return $(p_1, 1)$.

3. *Every audit op in π returns a set of pairs $\mathcal{P}$ such that*
   (π-Completeness): For each read operation op' by process p that precedes op in π, $(p, v) \in \mathcal{P}$, where v is the value returned by op'.
   (π-Accuracy): For any pair $(p, v) \in \mathcal{P}$, there is a read operation op' by process p that returned v, and op' precedes op in π.

Roughly speaking, π-*Completeness* formalizes that any read of a value from the register must be detected by the audit operation, while π-*Accuracy* ensures that a read is reported by an audit operation only if it has occurred. Note that taken together, π-Completeness and π-Accuracy say that a pair (p, v) is returned by the audit operation *if and only if* a read operation by process p, returning v, is linearized in π before the audit. That is, an *atomic audit* operation detects all the read operations linearized before it and does not detect any read operation linearized after it.

A *regular audit* operation detects all read operations that complete before the audit starts and does not detect any read operation that starts after it completes. An audit operation may detect some subset of the read operations that overlap it.

▶ **Definition 2** (Regular audit). *A history H is* atomic with regular audit *if there is a history H' in* complete(H)*, and a sequential history π that contains all read and writes operations in H' that satisfies the first two conditions of Definition 1, and in addition:*

3. *Every audit $op \in H'$ returns a set of pairs $\mathcal{P}$ such that*
   (H'-Completeness): For each read operation op' in H' by process p, that completes in H' before the invocation of op in H', $(p, v) \in \mathcal{P}$, where v is the value returned by op'.
   (H'-Accuracy): For any pair $(p, v) \in \mathcal{P}$, there is a read operation $op' \in H'$ by process p that returned v, and the invocation of op' in H' precedes the response of op in H'.

Note that while the condition on atomic audit operations (Definition 1) is stated relative to the linearization (sequential execution) π, the condition on regular audit is stated relative to the completion H' of the original history H. As we shall see, this seemingly-minor change leads to an important difference in the synchronization power of audit operations.

Figure 1 depicts a scenario where the responses of atomic audit and regular audit may differ.

In the rest of the paper, we consider that only one process can invoke write operations on the register, called the *writer*, which is also allowed to invoke audit operations. Thus, the *writer* is also an *auditor* of the register. When several processes are allowed to read the register, we call it *multi reader*; otherwise, it is *single reader*. Similarly, if several processes other than the writer can audit the register, we call it *multi auditor*; otherwise, it is *single auditor*.

For the correctness proof of the algorithms implementing the auditable registers, we assume that the values written to the register are unique.

Algorithm 1 Two-process consensus using swsr atomic registers with single-auditor atomic audit.

```
Shared Variables:
$R_i$, $i \in [0,1]$, swsr atomic register with writer / auditor $p_i$ and reader $p_{1-i}$, initially $\bot$

Local Variables:
$val$, initially $\bot$                                          ▷ value read from $R_{1-i}$
$audit\_response$, initially $\emptyset$                          ▷ response of audit on $R_i$

1: propose($v_i$)                                    ▷ Pseudo code for process $p_i$, $i \in [0,1]$
2:     $R_i.write(v_i)$
3:     $val \leftarrow R_{1-i}.read()$
4:     $audit\_response \leftarrow R_i.audit()$
5:     if $val = \bot$
6:         return $v_i$
7:     if $audit\_response = (p_{1-i}, \bot)$
8:         return $val$
9:     return $max(v_i, val)$
```

4 Using atomic audit to solve consensus

In this section, we investigate how atomic audit allows to solve consensus. An algorithm solving consensus satisfies the following properties:

Termination: A process decides within a finite number of its own steps.

Agreement: All processes decide on the same value.

Validity: The decision value has been proposed by some process.

4.1 Single-reader register with single-auditor atomic audit solves two-process consensus

Algorithm 1 solves consensus between two processes using two single-writer single-reader (swsr) atomic registers with a single-auditor atomic audit: R_i, for each $i \in \{0,1\}$, is a swsr register written and audited by process p_i and read by p_{1-i}.

Each process first writes the value it proposes in its own register. Then it reads the other process's register and audits its own register. Finally, it returns its own value or the other process's value, accordingly to the values returned from the read and audit operations. In particular, p_i returns its own value (Line 6) if it read the initial value from R_{1-i} (Line 5). In that case, p_{1-i} reads v_i from R_i (Line 3). The condition in Line 5 would not hold, and since the audit operation on R_{1-i} detects that p_i read $\bot$ from R_{1-i} (Line 4), p_{1-i} returns the value of val (Line 8), which is v_i. Finally, if p_i and p_{1-i} both read the input of the other process and they know this fact thanks to the result of the audit operation, they apply a deterministic rule to break the tie and choose the same value.

The pseudocode appears in Algorithm 1. In in the full version, we prove:

▶ **Theorem 3.** *Algorithm 1 solves consensus for two processes.*

4.2 Multi-reader register with multi-auditor atomic audit solves n-process consensus

We now generalize Algorithm 1 to solve consensus among n processes using single-writer multi-reader (swmr) atomic registers with *multi-auditor* atomic audit. Like the algorithm for two-process consensus, processes leverage the audit to reconstruct at which point of the execution the other processes are, and base their decision on it.

Algorithm 2 uses n swmr atomic registers with multi-auditor atomic audit $R_0, \ldots, R_{n-1}$, all initially $\perp$. Process p_i is the single writer of R_i, and all processes can read and audit R_i.

Each process p_i proposes its input, by writing it in R_i. Then, p_i reads and audits all the other registers. A simple situation is when one process, say p_i, writes and reads before all other processes, the audit detects that p_i read v_i in R_i and $\perp$ in all other registers. This implies that all later processes will read p_i's value in R_i and, thanks to the audit, detect that p_i is not aware of the other processes' propositions. In this case, v_i is the only value known to all processes, so it is safe to decide on v_i. In general, we can consider the set P of processes that write before any process reads. No process reads $\perp$ from the registers of processes in P, and this can be detected by auditing these registers. This means that all processes consider the input values of processes in P as safe to decide upon, and agreement can be reached by deterministically picking one of these values, e.g., the maximal one.

Each process keeps the following local data structures: *values*[ ] is an array of length n to hold the values read from $R_0, \ldots, R_{n-1}$, initially $\perp$; *safe_values* is a set that stores the proposed values that no process missed, initially $\emptyset$; and *audit_response* holds the results of audits on $R_0, \ldots, R_{n-1}$, initially $\emptyset$.

When proposing a value v_i, each process p_i first writes v_i in R_i (Line 2). Next, p_i reads $R_0, \ldots, R_{n-1}$ and stores the responses in *values*[ ] (Line 4). Finally, p_i audits $R_0, \ldots, R_{n-1}$ and stores the returned pairs in a set *audit_response* (Line 6). For each R_j, when a value is added to *audit_response*, p_i checks if there is a process that read $\perp$ from R_j (Line 7). If this is not the case, then the value in R_j is considered safe, and is added to *safe_values* (Line 8). Finally, it returns the maximum value in *safe_values* (Line 9). (We assume, for simplicity, that the input values are from a totally-ordered set.)

▶ **Lemma 4.** *A process p_i adds a value v to safe_values, only if v was proposed by some process.*

Proof. Process p_i adds values it reads from $R_0, \ldots, R_{n-1}$, the registers of other processes, to *safe_values* in Line 8. The value read from a register R_j, in Line 4, is either $\perp$ (the initial value of the register) or the value proposed by p_j, written to R_j in Line 2.

We next argue that p_i does not add $\perp$ to *safe_values*. If p_i read $\perp$ from some R_j, then since its audit on R_j follows its read from R_j, the π-Completeness of its audit operation (Definition 1(3)) implies that $(p_i, \perp)$ is contained in the response of $R_j.audit()$. By the condition of Line 7, $\perp$ is not added to *safe_values*. ◀

▶ **Lemma 5.** *Algorithm 2 satisfies validity.*

Proof. A process decides on a value in *safe_values* (Line 9), and by Lemma 4, this set contains only values proposed by some process. We complete the proof by showing that *safe_values* is not empty.

Let p_k be the first process to apply its write of v_k to R_k. Since all processes read the registers of the other processes after applying the write, it follows that all the processes read $v_k \neq \perp$ from R_k. By π-Accuracy (Definition 1(3)), the audit of R_k does not contain a pair $(p_j, \perp)$, for any p_j. Therefore, the condition in Line 7 holds and p_i adds v_k to *safe_values* in Line 8, as needed. ◀

Algorithm 2 n-process consensus using swmr atomic registers with multi-auditor atomic audit.

```
Shared Variables:
$R_i$, $i \in [0, n-1]$, swmr atomic registers with multi-auditor atomic audit; $R_i$ is written by
process $p_i$, initially $\perp$

Local Variables:                                   ▷ Pseudo code for process $p_i$, $i \in [0, n-1]$
$values[\,]$ an array of length $n$, initially $\perp$
$safe\_values$ a set, initially $\emptyset$
$audit\_response$ a set, initially $\emptyset$
1: propose($v_i$)
2:     $R_i.write(v_i)$
3:     for $0 \le j < n$
4:         $values[j] \leftarrow R_j.read()$
5:     for $0 \le j < n$
6:         $audit\_response \leftarrow R_j.audit()$
7:         if $\nexists(*, \perp) \in audit\_response$          ▷ no process read $\perp$ from $R_j$
8:             $safe\_values.add(values[j])$
9:     return max($safe\_values$)
```

▶ **Lemma 6.** *Algorithm 2 satisfies agreement.*

Proof. We prove that all processes have the same set *safe_values* when deciding, which immediately implies agreement. Suppose that process p_i is the first to add a value v_k to its *safe_values* set. This means that p_i reads v_k from register R_k (Line 4).

Let aop_i be the audit by p_i on R_k (Line 7). Since p_i adds v_k to *safe_values*, it follows that no pair $(p_j, \perp)$, for some process p_j, is included in the response to aop_i. This implies that all read operations from R_k that are linearized before aop_i do not return $\perp$.

Consider a read operation by process p_j from R_k that is linearized after aop_i. This follows the read of processes p_i from R_k, which returns $v_k \neq \perp$, and hence, this read will also read $v_k \neq \perp$. (Since only p_k writes to R_k, once, changing its value from $\perp$ to v_k.)

Thus, no read from R_k returns $\perp$. This means that any process p'_i consider $v_k \neq \perp$ in Line 7. Moreover, by the π-Accuracy property of the audit operation (Definition 1(3)), it follows that no pair $(p_j, \perp)$ is contained in the result of the audit operation by p'_i on R_k. This implies that v_k is in *safe_values* of p'_i.

Then, the *safe_values* sets of all processes are identical, and they all decide on the same value. ◀

Therefore, the algorithm satisfies validity (Lemma 5) and agreement (Lemma 6). Furthermore, all the loops in the **propose** are iterated at most n times. Since the operations invoked in the **propose** are wait-free, we get that Algorithm 2 is wait-free. This implies:

▶ **Theorem 7.** *Algorithm 2 solves consensus for n processes.*

5 Atomic audit implementations

We now turn to present several implementations of an atomic single-writer register with atomic audit. The results of the previous section indicate which synchronization primitives must be used in the implementations. Since two-process consensus can be solved with a single auditor and single reader, we cannot avoid synchronization primitives with consensus number at least two; we use swap and fetch&add (Sections 5.1, 5.2 and 5.3). When there are multiple auditors and multiple readers, we use a universal synchronization primitive, compare&swap (Section 5.4), whose consensus number is ∞, in addition to fetch&add.

5.1 Implementing single-reader atomic register with single-auditor atomic audit using swap

We implement a swsr atomic register with single-auditor atomic audit using a swap primitive. We use a shared register R with initial value v_0, which holds the last written value, if the last operation was a write, or a special value ($\perp$) if the last operation was a read; the audit operations do not affect the value of R. In a write(v) operation, the (single) writer applies *swap* to R, atomically writing v into R and retrieving the overwritten value to check if the reader read the previously written value. In the latter case, the swap returns a special value $\perp$.

The pseudocode appears in Algorithm 3. The reader keeps the following local data structures: *val* that holds the value read from R, initially $\perp$; and *read_result* that holds the value returned by the last read operation, initially $\perp$. The writer and auditor keeps the following local data structures: *curr_val* that holds the last value written in R, initially v_0; *prev_val* that holds the previous value written into R, initially $\perp$; and a set, called *audit_result*, which stores the pairs (process,value) of the detected read operations, initially $\emptyset$.

In a read, the reader atomically reads the last value written into R and swaps it with $\perp$ to notify the writer that it read the last value written. If the response is not $\perp$, then this is the response of the read, which the reader stores in *read_result* for future read operations, before returning. Otherwise, no write has occurred since its previous read, so the read returns the value in *read_result* (without changing its value).

In a write, the writer stores in *prev_val* the value of the previous write, from *curr_val* (Line 7), and stores in *curr_val* the value v it is going to write (Line 8). In this way, if the next write operation detects that the reader has read the previous value written, the writer knows what this value is. Then, the writer swaps *curr_val* into R: atomically writing it into R and retrieving the overwritten value. If the writer gets $\perp$ from the swap, then the reader has read the last value it wrote (stored in *prev_val*), and it adds the pair (reader, *prev_val*) to *audit_result* (Line 10).

In an audit, the auditor (who is also the writer) returns all the (process,value) pairs collected during the previous write operations. By reading R, the auditor checks whether the reader read the value of the last write operation, in which case R is $\perp$. In this case, it adds the pair (reader, *curr_val*) to *audit_result*. Finally, the audit returns *audit_result*.

Fix a history H. It has at most two pending operations: one, either an audit or a write, by process p_w, and another by process p_r, which must be read. We never complete a pending audit. We complete a pending read in H if and only if some audit contains (p_r, v) in its response and no preceding read in H (which must be complete) returns v. We complete a pending write in H if and only if some read (including those completed) returns the corresponding value.

A pending operation that is completed has accessed R with a swap: A read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the swap in Line 2. A write is completed if some read has read its value, namely, the write has executed the swap in Line 9. We totally order all the completed operations by the order they apply their unique primitive on R. Call this total order π and note that it respects the real-time order of the high-level operations on the register, since the swaps and reads are inside the operations' intervals.

Algorithm 3 Implementing a single-reader atomic register with single-auditor atomic audit using swap.

```
    Shared Variables:
    $R$, accessed with $read$ and $swap$, initially $v_0$

    Local Variables:                                    ▷ Pseudo code for reader $p_r$
    $val$, initially $\perp$                                        ▷ result of the swap
    $read\_result$, initially $\perp$                          ▷ value returned by the read

 1: Read()
 2:     $val \leftarrow R.swap(\perp)$
 3:     if $val \neq \perp$
 4:         $read\_result \leftarrow val$
 5:     return $read\_result$

    Local Variables:                          ▷ Pseudo code for writer and auditor $p_w$
    $curr\_val$, initially $v_0$                                   ▷ last value written
    $prev\_val$, initially $\perp$                             ▷ previous value written
    $audit\_result$, initially $\emptyset$   ▷ set of tuples $(p, v)$, with $p$ the reader and $v$ a value

 6: Write($v$)
 7:     $prev\_val \leftarrow curr\_val$
 8:     $curr\_val \leftarrow v$
 9:     if $R.swap(v) = \perp$
10:         $audit\_result.add \leftarrow (p_r, prev\_val)$
11:     return

12: Audit()
13:     if $R.read() = \perp$
14:         $audit\_result.add(p_r, curr\_val)$
15:     return $audit\_result$
```

▶ **Lemma 8.** *Every read in π returns the value of the most recent preceding write, if there is one, and the initial value, otherwise.*

Proof. Consider a read op_r that returns a value v, and let $op_r{}'$ be the first read that returns this value. Since *read_result* is updated only if the value returned by the swap in Line 2 is not $\perp$, then the swap of $op_r{}'$ returns v. Thus, there is a preceding swap that sets R to v, and it must be by some write op_w of value v. Since reads and writes are linearized by the order of their swaps, op_w precedes $op_r{}'$, and therefore, also op_r in π.

We next argue that no other write is linearized between op_w and op_r in π. Assume otherwise, and let $op_w{}'$ be the last write that is linearized before op_r in π.

If the swap of op_r in Line 2 returns a value different from $\perp$, then this value was written by $op_w{}'$ because this is the last preceding swap that writes a non-$\perp$ value before the swap by op_r. This contradicts the assumption that op_r returns the value written by op_w.

If the swap of op_r in Line 2 returns $\perp$, this means that an earlier read that executed Line 2 after $op_w{}'$ executed its swap. The first such read swaps from R the value written by $op_w{}'$ with $\perp$. By Line 4, the value of *read_result* is not v when op_r returns in Line 5, which is a contradiction. ◀

▶ **Lemma 9.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Completeness property.*

Proof. Consider an audit op_a that returns a set $\mathcal{P}$, and let op_r be a read returning v that precedes op_a in π. By Lemma 8, every read returns the value of the most recent preceding write in π. Let $op_r{}'$ be the first read that returns v. Then $op_r{}'$ sets R to $\perp$, and the value in *curr_val* is v.

If there is no write between op_r and op_a, the audit reads $\perp$ from R (Line 13), while *curr_val* is still v in Line 14, implying that the audit adds (p_r, v) to $\mathcal{P}$. Otherwise, there is a write between op_r and op_a. Let op_w be the first such write, and notice that op_w completes, since there is a following audit (by the same process). Moreover, since it is the first write after op_r, the value of R is $\perp$ when p_w executes Line 9 and *curr_val* is v immediately before it executes Line 7. Thus, the pair (p_r, v) is added to *audit_result*. ◀

▶ **Lemma 10.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Accuracy property.*

Proof. Consider an audit op_a that returns a set $\mathcal{P}$, and let (p_r, v) be a pair in $\mathcal{P}$. The first operation op that adds (p_r, v) to $\mathcal{P}$ is either op_a itself or a write / audit that precedes op_a in π. This is because the variable *read_result* holding set $\mathcal{P}$ is updated immediately after the swap by p_w in the corresponding operation.

If (p_r, v) is added to $\mathcal{P}$ by an audit op, then *curr_val* is v when this happens. Since the condition in Line 13 holds, there is a reads that swaps $\perp$ into R after v was written to R. This read is between the write of v and op and by Lemma 8, it returns v, proving the lemma.

If (p_r, v) is added to $\mathcal{P}$ by a write op, then by Line 7 and Line 10, v is the value written by the write that immediately precedes op in π. Then v is the value returned by the read that swaps v with $\perp$, which allows the condition in Line 9 to hold. This read precedes op and therefore, it also precedes op_a. ◀

Lemma 8, Lemma 9 and Lemma 10 imply:

▶ **Theorem 11.** *Algorithm 3 implements a single-writer single-reader atomic register with single-auditor atomic audit.*

5.2 Implementing multi-reader atomic register with single-auditor atomic audit using swap and fetch&add

The algorithm for a *multi-reader* atomic register with single-auditor atomic audit follows a similar idea as Algorithm 3 for a *single* reader, by having each reader leave a trace of each of its reads. However, there is an additional difficulty of allowing the writer to atomically retrieve the traces of all readers when writing a new value or doing an audit.

We address this difficulty by using fetch&add, in addition to swap. A *fetch&add* allows to accurately change the value of a shared variable R so that its binary representation captures multiple pieces of information: The high-order bits hold the value written by the writer, while the n low-order bits indicate whether the readers have read the value. Specifically, the bit in position i, denoted bit_i, is associated with reader p_i, $0 \leq i < n$, and holds either 0 or 1. bit_i is set to 1 by p_i to indicate that it has read the value stored in the high-order bits of R; it is 0, otherwise. We use two functions to extract information from R. If R holds *val*, then GetValue(*val*) retrieves the value stored in the high-order bits of *val* and GetsBits(*val*) retrieves an array of n low-order bits of *val*.

In more detail (see Algorithm 4), when a reader p_i reads a value written in R, it sets bit_i to 1 by adding 2^i to the value stored in R. Since a reader can read the same value several times, p_i checks that bit_i is not already set to 1, before adding 2^i to R (Line 4). This ensures that p_i changes only its bit.

Algorithm 4 Implementation of multi-reader atomic register with single-auditor atomic audit using *fetch&add* and *swap*, for n readers.

```
Shared Variables:
R accessed with read, swap and fetch&add primitives, initially $v_0 * 2^n$

Local Variables:                              ▷ Pseudo code for reader $p_i$, $i \in [0, n-1]$
val, initially $\perp$                           ▷ content of the register
read_result, initially $\perp$                   ▷ last value read

1:  Read()
2:      $val \leftarrow R.read()$
3:      if $(GetBits(val)[i] = 0)$
4:          $read\_result \leftarrow GetValue(R.fetch\&add(2^i))$
5:      return read_result

Local Variables:                              ▷ Pseudo code for writer and auditor $p_w$
audit_result, initially $\emptyset$             ▷ set of tuples $(p, v)$, with $p$ the reader and $v$ a value
curr_val, initially $v_0$                        ▷ last value written
prev_val, initially $\perp$                      ▷ previous value written
val with initial value $\perp$                   ▷ content of the register

6:  Write($v$)
7:      $prev\_val \leftarrow curr\_val$
8:      $curr\_val \leftarrow v$
9:      $val \leftarrow R.swap(v, 0^n)$                ▷ write $v$ in the high order bits
10:     for $0 \le j < n$
11:         if $(GetBits(val)[j] = 1)$           ▷ check if $p_j$ read the previous value
12:             $audit\_result.add(p_j, prev\_val)$
13:     return

14: Audit()
15:     $val \leftarrow R.read()$
16:     for $0 \le j < n$
17:         if $(GetBits(val)[j] = 1)$           ▷ check if $p_j$ read the last value
18:             $audit\_result.add(p_j, curr\_val)$
19:     return audit_result
```

When writing a new value v, the writer swaps the value v and resets the n low-order bits to 0 into R and obtains the previous value of R, into a local variable *val*. Then for each reader p_i, the writer retrieves bit_i from *val* (Lines 10 and 11). If bit_i is equal to 1, the writer knows that reader p_i has read the previous value and the pair $(p_i, prev\_val)$ is added to the set to be returned by an audit, called *audit_result*. *audit_result* is a local variable, which can be accessed both by the writer and the auditor because they are the same process.

In a similar manner, an audit operation also reads R to detect high-level read operations that may have read the last value written.Since *audit_result* is a set, the pair will not be added if it was already in the set. (An efficient implementation of a sequential set can be used for this local variable.)

Fix a history H. Note that there are at most $n + 1$ pending operations in H: one (either an audit or a write) by the writer, and possibly one read operation for each reader. We never complete a pending audit. We complete a pending read invoked by process p_i in H if and only if some audit contains (p_i, v) in its response and no earlier read in H (which must be complete) returns v to p_i. We complete a pending write in H if and only if some read in H returns the corresponding value. Note that if a pending operation is completed, then it applied a primitive to R: a write is completed if some read has read its value, namely, the write has executed the swap in Line 9; a read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the *fetch&add* in Line 4.

We totally order all the completed operations by the order they apply their last primitive (*swap*, *read* or *fetch&add*) to R. A write or an audit applies only one primitive. For a read, the last primitive is the *fetch&add*, if this is the first time that the process reads a given value, and otherwise, it is the read. Let π denote this total order, and note that it respects the real-time order of the high-level operations on the auditable register because each such step is in the execution interval of the corresponding operation. In the full version, we prove:

▶ **Theorem 12.** *Algorithm 4 implements a single-writer multi-reader atomic register with single-auditor atomic audit.*

5.3 Implementing single-reader atomic register with multi-auditor atomic audit using swap and fetch&add

The algorithm for a single-reader atomic register with *multi-auditor* atomic audit follows a similar idea as the algorithm for a *multi* reader in the previous section, using a shared register accessed with the *read*, *swap* and $fetch\&add$ primitives to support the detection of read operations by the writer and the auditors.

Since an audit operation can overlap read, write and other audit operations, we need an additional mechanism to ensure that the return value of the audit is linearizable. The reader and the auditors share information in an unbounded array of read/write registers called *pairs*, where $pairs[k]$ indicates whether the reader read the k-th value written by the writer (if there was such write). If $pairs[k]$ contains the initial value $\perp$ then the reader has not read the k-th value written, otherwise $pairs[k]$ contains that value. Each value written has a unique sequence number that is incremented when the writer performs a new write. When performing a write of a value v, the writer applies a swap to R to atomically write v together with its sequence number and set the lowest order bit of the register to 0 to indicate a new write (not yet read).

Algorithm 5 presents the pseudocode. As in Algorithm 4, in a read operation, the reader reads the value of R and sets the low-order bit to 1 if it was equal to 0 (indicating that this is the first time p_r read this value). Additionally, it writes the value read in the corresponding entry of *pairs*. When a process performs an audit operation, it retrieves from R the sequence number *sn* of the last write operation, and also checks whether the reader has read the last value written v. In the latter case, it writes v into $pairs[sn]$. Then, it reads all the entries of *pairs*, from index *sn* down to the first, to obtain its return set.

Because the value v and the sequence number are unbounded, we interleave them bit by bit in R, as done in [18]. Three functions are used to extract information from R. If R holds $val = (v, sn, bit)$, then GetBit(val) retrieves its lowest-order bit, GetValue(val) retrieves the value v, and GetSn(val) retrieves *sn*.

Note that there are at most n pending operations in H: one (either an audit or a write) by the writer, one (either an audit or a read) by the reader, and possibly one (an audit) for all the other processes. We construct a history H' by completing some operations in H. We never complete a pending audit. We complete a pending read in H if and only if some audit contains (p_r, v) in its response and no preceding read in H (which must be complete) returns v. After completing the reads, we complete a pending write if and only if some

Algorithm 5 Implementation of a single-reader atomic register with multi-auditor atomic audit using *swap* and *fetch&add*.

```
    Shared Variables:
    $R$ accessed with $read$, $fetch\&add$ and $swap$. Its initial value is $(v_0, 0, 0)$.
    $pairs$ An unbounded array of read/write registers, shared by all processes. Initially all registers
    contains the special value $\perp$.

    Local Variables:                                        ▷ Pseudo code for reader $p_r$
    $val$, initially $\perp$                                     ▷ content of the register
    $sn$, initially 0                     ▷ the sequence number of the value store in the register
    $read\_result$, initially $\perp$                        ▷ value read from the register

 1: Read()
 2:     $val \leftarrow R.read()$
 3:     if $(GetBit(val) = 0)$
 4:         $val \leftarrow GetValue(R.fetch\&add(1))$
 5:         $read\_result \leftarrow GetValue(val)$
 6:         $pairs[GetSn(val)].write(read\_result)$
 7:     return $read\_result$

    Local Variables:                                        ▷ Pseudo code for writer $p_w$
    $prev\_val$, initially $\perp$                              ▷ content of the register
    $sn$, initially 0                                    ▷ the sequence number of the write

 8: Write($v$)
 9:     $sn \leftarrow sn + 1$
10:     $prev\_val \leftarrow R.swap((v, sn, 0))$
11:     if $(GetBit(prev\_val) = 1)$
12:         $pairs[GetSn(prev\_val)].write(GetValue(prev\_val))$   ▷ detect the read of the previous write
13:     return

    Local Variables:                                        ▷ Pseudo code for auditor $p_i$
    $audit\_result$, initially $\emptyset$                   ▷ set of couples (process, value)
    $audit\_index$, initially 0                  ▷ index of the last updated value in pairs[ ]
    $val$, initially $\perp$                                    ▷ content of the register

14: Audit()
15:     $val \leftarrow R.read()$
16:     $audit\_index \leftarrow GetSn(val)$
17:     if $(GetBit(val) = 1)$
18:         $pairs[audit\_index].write(GetValue(val))$
19:     for $j$ from $audit\_index$ to 0
20:         if$(pairs[j].read() \neq \perp)$
21:             $audit\_result.add(p_r, pairs[j].read())$
22:     return $audit\_result$
```

(completed) read returns the corresponding value. We remove from H' all other pending operations in H. Note that if a pending operation is completed, then it applied a primitive to R: a write is completed if some read has read its value, namely, the write has executed the swap in Line 10; a read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed the *fetch&add* in Line 4. Thus, all operations in H' applied a primitive to R, and we can associate a sequence number sn to each operation, which corresponds to the sequence number they read (for a read or audit operation) or write (for a write operation) from the shared register R during this primitive.

We construct a total order π of the operations in H'. First, we put in π all the write operations according to the order they occur in H'; because write operations are executed sequentially by the unique writer, this sequence is well-defined and the order is consistent with the sequence number associated with the values written.

Next, we add the read operations in π. Since there is a unique reader the read operations are executed sequentially. The sub-sequence of read operations that returns a value with sequence number sn is placed immediately after the write operation that generates the sequence number sn, while preserving their order in H'.

The construction of π immediately implies that a read operation returns the value written by the write preceding it in π.

▶ **Lemma 13.** *Every read operation in π returns the value of the most recent preceding write in π, if there is one, and the initial value otherwise.*

Finally, we consider the audit operations one by one, in reverse order of their response in H. Consider an audit operation op_a and let sn be the sequence number it read at Line 15. There are three cases.

- Case 1: If op_a reads a value v in $pairs[sn]$, we place op_a in π immediately after the last read with sequence number sn that starts before op_a terminates.
- Case 2: If op_a read a value v in $pairs[sn-1]$ and the initial value $\perp$ in $pairs[sn]$, we place op_a in π immediately after the write operation with sequence number sn (at the start if no such write exists).
- Case 3: If op_a read the initial value $\perp$ both in $pairs[sn]$ and in $pairs[sn-1]$, then we place op_a in π immediately after the write operation with sequence number sn if it terminates before op_a is invoked in H'. Otherwise, op_a is placed immediately after the write operation with sequence number $sn-1$ (at the start if there is no such operation).

Case 3 handles the situation where an audit operation op_a reads a sequence number sn but misses a read operation op_r that returns the value with sequence number $sn-1$. This happens only if op_a is concurrent with op_r and with the write op_w that generates the sequence number sn; in particular, op_r and op_w write into $pairs[sn-1]$ after op_a read it.

In the full version, we prove that this linearization preserves the real-time order of non-overlapping operations. We next argue completeness and accuracy.

▶ **Lemma 14.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Completeness property.*

Proof. Consider an audit operation op_a that returns a set $\mathcal{P}$, and let op_r be a read operation by process p_r that returns a value v and precedes op_a in π. We prove that $(p_r, v) \in \mathcal{P}$.

Let sn be the sequence number read by op_a, and let sn' be the sequence number of the value read by op_r. Since op_a follows op_r in π, according to our linearization rules $sn \geq sn'$. Thus, $audit\_index \geq sn'$ and op_a reads $pairs[sn']$ (Lines 19). If the read returns v, then (p_r, v) is added to $\mathcal{P}$ (Line 21), and the lemma follows. It remains to prove, by way of contradiction, that op_a does not read $\perp$ from $pairs[sn']$. We consider the possible cases:

1. $sn = sn'$, since op_a read $\perp$ from $pairs[sn']$, Case 2 or Case 3 apply. Thus, op_a is placed in π before the write that generates sequence number sn' (at the latest). Since op_r follows this write, op_a is placed before op_r in π.
2. $sn = sn' + 1$: Case 2 does not hold because op_a read $\perp$ from $pairs[sn']$ with $sn' = sn - 1$. Since op_r precedes op_a in π, by Case 3, op_a follows the write with sequence number sn in π. We are left with two cases.
   If op_a read a value $\neq \perp$ from $pairs[sn]$ (Case 1), then we reach a contradiction by showing op_a cannot read $\perp$ from $pairs[sn-1]$. Since $pairs[sn] = v'$, there is a read operation $op_{r'}$ that reads v', and since there is a single reader, $op_{r'}$ follows op_r in H. Thus, the value of $pairs[sn-1]$ is set to v before $pairs[sn]$ is set to v'. By Line 19, op_a first reads $pairs[sn]$ and then reads $pairs[sn-1]$, and therefore, it does not read $\perp$ from $pairs[sn-1]$.

Otherwise, op_a read $\bot$ from $pairs[sn]$ but the write op_w that generates sn precedes op_a (Case 3). Since op_r returns v, there is a read operation (possibly op_r) that set the corresponding low-order bit to 1 (Line 4). Then, when op_w writes the new value with sequence number sn, it reads 1 from this bit and sets $pairs[sn-1] = v$, so op_a does not read $\bot$ from $pairs[sn-1]$.

3. $sn \geq sn' + 2$. Then the write operation with sequence number $sn' + 1$ completes before op_a reads R, and we can apply the same reasoning as in case b(Case 3). ◀

▶ **Lemma 15.** *The set of pairs $\mathcal{P}$ returned by an audit in π satisfies the π-Accuracy property.*

Proof. Consider an audit operation op_a that returns a set $\mathcal{P}$, and let (p_r, v) be a pair in $\mathcal{P}$. We prove that a read operation op_r by process p_r that returns v is placed before op_a in π.

Since (p_r, v) is in $\mathcal{P}$, op_a adds (p_r, v) to *audit_result* because it read $pairs[sn] = v$ for some sn. If v was written by the reader (Line 6), then the reader returned the value v associated with sn, in Line 4. Otherwise, v was written to $pairs[sn]$ either by the writer (Line 12) or by the auditor (Line 18). This means that the writer or the auditor read the bit set to 1 when, respectively, checking the condition in Line 11 or in Line 17. This bit is set to 1 only by the reader when reading the corresponding value in Line 4.

Thus, there is a read operation op_r that read the value v with sequence number sn that does not follow op_a in H. We now show that it precedes op_a in π.

Let sn' be the sequence number of op_a. Since op_a reads $pairs[sn]$, $sn' \geq sn$. If $sn' = sn$, then since op_a read v in $pairs[sn]$, op_a is placed after op_r in π and the claim holds. If $sn' = sn + 1$, then since op_a reads v from $pairs[sn]$, by Cases 1 and 2, it is placed after the write that generates sequence number $sn+1$, and therefore, after op_r. Finally, if $sn' > sn+1$, then op_a is placed in π after a write with a sequence number greater than sn, and hence, after op_r. ◀

▶ **Theorem 16.** *Algorithm 5 implements a single-writer single-reader atomic register with multi-auditor atomic audit.*

5.4 Implementing multi-reader atomic register with multi-auditor atomic audit using compare&swap

To deal with multiple readers, as in Algorithm 4, each reader sets a dedicated bit in the n lower-order bits of a shared register R and the writer writes the value together with a sequence number in the higher-order bits of R. To deal with multiple auditors, we use an array *pairs*, as in Algorithm 5. To accommodate multiple readers, the array is bi-dimensional, with an unbounded number of columns (corresponding to each written value) and n rows, one for each reader. Specifically, $pairs[i][sn] = \bot$ indicates that process p_i has not read the value v written by the sn-th write operation (if any); otherwise, $pairs[i][sn] = v$.

We do not need readers to write into pairs because the writer applies a *compare&swap* to write the new value in R, using the previously-read state of R. So, if in the meanwhile, some reader read the current value stored in R and set its bit to 1, the *compare&swap* fails. Thus, the writer detects and write into *pairs* all the read operations of the last value written before succeeding the next write. Thus, either the auditor can detect the read operations by reading R because the bits were not reset by the new write, or this information is in *pairs*.

Because the sequence numbers and the corresponding values are unbounded, we cannot a priori divide the high-order bits between them. Instead, we interleave them bit-by-bit, as done in the previous algorithm (following [18]).

Algorithm 6 shows the pseudocode. Each process stores the value read from R in a local variable val, in order to select the information it needs. A read operation by a process p_i checks if low order bit associated with p_i is equal to 1 or 0, meaning p_i has already read the current value or not, respectively. We use the following functions on val: $\mathsf{GetValue}(val)$ returns the value stored in the high-order bits of val, $\mathsf{GetSn}(val)$ returns the sequence number stored in the high-order bits of val, and $\mathsf{GetBits}(val)$ returns the n low-order bits of val.

A reader has a local variable called *read_result* to store the value that has to be returned, initially $\perp$. This is used to ensure that if a new value was not written, consecutive read operations by the same process can return the correct value without setting the corresponding bit to 1 more than once.

An auditor has a local variable called *audit_result* that holds a set of pairs (process,value), one for each detected read operation; it is initially $\emptyset$. The local variable *audit_index* holds the sequence number read from R, indicating the last column of the matrix *pairs* written by the writer that the auditor has to read; it is initially 0.

In a read, the reader first reads R to check whether it has already read its last value. If this is the case, it simply returns the value. Otherwise, it applies a *fetch&add* to set its bit to 1 (indicating that it read the value) and returns the value represented by the high-order bits of the value returned by the *fetch&add*.

In an audit, the auditor first reads R to get the sequence of bits indicating the read operations performed since the last write operation and to atomically get the sequence number of the last write operation. It then reads all the pairs stored in entries of *pairs* until this index, and adds them to the set that the audit will return; this set is persistent. Finally, it adds to the set additional pairs corresponding to the low-order bits of R that are set. For simplicity, all the audit reads all entries of *pairs* starting from 0 up to the read sequence number. It is simple to ensure that the same auditor reads each column of *pairs* at most once, by using a persistent local variable to store the last column read.

To write a new value v, the writer increments the sequence number sn, and applies *compare&swap* to R to store sn together with v and reset the n low-order bits to 0. If this is successful, the operation completes; otherwise, the writer reads R, and for each of the n low-order bits that is set to 1, it writes v into the corresponding entry of $pairs[\,][sn-1]$ to announce the read operation that set the bit. The writer then retries the *compare&swap*.

We first show that all operations (by a correct process) complete within a finite number of steps. This is immediate for read operations. For an audit operation, the number of the iterations of the first *for* loop is bounded by the value of n and of *audit_index* read from R, while the second *for* loop has n iterations. The next lemma (see the full version) bounds the number of iterations in a write operation.

▶ **Lemma 17.** *The compare&swap (Line 19) of a write operation fails at most n times.*

To prove the linearizability of the algorithm, fix a history H. Note that there are at most n pending operations in H, one for each process. We construct a history H' by completing some pending read and write operations in H; we never complete a pending audit. We first complete a pending read invoked by process p_i in H if and only if some audit contains (p_i, v) in its response and no read in H (which must be complete) returns v to p_i. After completing the reads, we complete a pending write if and only if some (completed) read returns the corresponding value. We remove from H' all other pending operations in H.

Note that if a pending operation is completed, then it applied a primitive to R: A read is completed if it is the only read that returns a value detected by an audit, thus, the read has executed *fetch&add* in Line 4. A write is completed if some read has read its value, namely, the write has executed the *compare&swap* in Line 19.

Algorithm 6 Implementation of a multi-reader atomic register with multi-auditor atomic audit using *compare&swap* and *fetch&add* for n processes.

```
Shared Variables:
$R$: a register shared by all processes, accessed with $read$, $compare\&swap$, and $fetch\&add$.
It contains a sequence number, the corresponding value, and $n$ bits. Initially $(0, v_0, 0^n)$.
$pairs[n, \ldots]$: a matrix of read/write registers, where $pairs[j][k]$ indicates if process $p_j$ has read
the k-th written value. Initially, $\perp$.

Local Variables:                          ▷ Pseudo code for reader and auditors $p_i$ $(i \in [0, n-1])$
$temp$ initially $\perp$                                          ▷ the content of the register $R$
$read\_result$ initially $\perp$                                          ▷ the last value read
$audit\_result$ initially $\emptyset$                                    ▷ set of (process, value) pairs
$audit\_index$ initially 0                           ▷ index of the last updated value in pairs[ ]

1: Read()
2:     $temp \leftarrow R.read()$
3:     if (GetBits($temp$)[$i$] = 0)
4:         $read\_result \leftarrow$ GetValue($R.fetch\&add(2^i)$)
5:     return $read\_result$

6: Audit()
7:     $temp \leftarrow R.read()$
8:     $audit\_index \leftarrow$ GetSn($temp$)
9:     for $0 \leq j < n$
10:        for $0 \leq k < audit\_index$
11:            if($pairs[j][k].read() \neq \perp$)
12:                $audit\_result.add(p_j, pairs[j][k].read())$
13:    for $0 \leq j < n$
14:        if (GetBits($temp$)[$j$] = 1)              ▷ checks if $p_j$ read the last value written in R
15:            $audit\_result.add(p_j$, GetValue($temp$))
16:    return $audit\_result$

Local Variables:                                                ▷ Pseudo code for writer $p_0$
$temp$ initially $\perp$                                                ▷ the value read from $R$
$sn$ initially 0                                    ▷ sequence number of the high-level writes
$val$ initially $v_0$                                  ▷ input value of the last high-level write
$bits$[ ] initially $0^n$                  ▷ $n$ lowest-order bits of $R$ to detect high-level reads

17: Write($v$)
18:    $sn \leftarrow sn + 1$
19:    while($R.compare\&swap((sn-1, val, bits), (sn, v, 0^n)) \neq True$)
20:        $temp \leftarrow R.read()$
21:        $bits \leftarrow$ GetBits($temp$)
22:        for $0 \leq j < n$
23:            if ($bits[j] = 1$)                            ▷ check if $p_j$ read the last value
24:                $pairs[sn-1][j].write(val)$
25:    $bits \leftarrow 0^n$
26:    $val \leftarrow v$
27:    return
```

We construct a sequential history π that contains all the operations in H', while preserving their real-time order. We (totally) order all the read, audit and write operations in H' according to the order they apply their last primitive on R: this is either a *read* or a *fetch&add* for read operations, it is a *read* for an audit, and the *compare&swap* for write operations. Note that these are atomic primitives and their order is well-defined. Clearly, operations are linearized inside their execution intervals, implying that π preserves the real-time order of all the operations in H. In the full version, we prove that this order satisfies completeness and accuracy.

▶ **Theorem 18.** *Algorithm 6 implements a single-writer multi-reader atomic register with multi-auditor atomic audit.*

Note that all (process,value) pairs must be stored somewhere in order to allow the audit to return all the read values. When there is a single auditor, as in Section 5.1 and Section 5.2, the pairs are stored locally at the auditor. This space can be reduced if the π-Completeness property is weakened to require that the audit operation returns only the last $k \geq 1$ values read by each reader. This weaker form of completeness does not affect the consensus number.

6 The consensus number of atomic audit

The *consensus number* [12] of a concurrent object type X is the largest positive integer m such that consensus can be wait-free implemented from any number of read/write registers, and any number of objects of type X, in an asynchronous system with m processes. If there is no largest m, the consensus number is infinite.

Algorithm 1 solves consensus among two processes, using only swsr atomic registers with single-auditor atomic audit. This implies that the consensus number of a swsr atomic register with single-auditor atomic audit is at least 2. Clearly, the same holds if the register is multi-reader or multi-auditor. To prove that the consensus number of this object type is 2, it remains to prove that it is not possible to solve consensus for more than 2 processes. To this aim, we provide algorithms that implement it with a single auditor (with single or multiple readers), using a single register that supports a combination of read, swap and $fetch\&add$. In particular, Algorithm 3 implements a single-reader atomic register with a single-auditor atomic audit by applying swap and read primitive operations on a single register. Algorithm 4 implements a multi-reader atomic register with a single-auditor atomic audit by applying swap, fetch&add, and read primitives on a single register.

Herlihy [12] proves that there is no wait-free consensus algorithm for three processes using registers that support any combination of *read*, *write*, *swap* and $fetch\&add$. It follows that an atomic register with a single-auditor atomic audit cannot be used to solve consensus among three or more processes, which implies:

▶ **Proposition 19.** *A single-reader or multi-reader atomic register with a single-auditor atomic audit has consensus number two.*

Similarly, Algorithm 5 implements a single-reader atomic register with multi-auditor atomic audit. It only uses a register accessed by read, swap and fetch& add primitives, in addition to read / write registers. Herlihy's impossibility together with Theorem 3, imply:

▶ **Proposition 20.** *A single-writer single-reader atomic register with multi-auditor atomic audit has consensus number two.*

Table 1 Consensus number of atomic register with atomic audit.

Number of writers	Number of readers	Number of auditors	Consensus number
1	1	1	2
1	n	1	2
1	1	n	2
1	n	n	$\geq n$

We also show that a multi-reader atomic register with multi-auditor atomic audit has consensus number larger than 2 if each reader is also an auditor of the register. In particular, according to Algorithm 2, we can solve consensus among n processes using n-reader atomic registers with n-auditors atomic audit. Thus, the consensus number of this object type is at least n. See Table 1.

We finally provide an implementation of the swmr atomic register with multi-auditor atomic audit using read/write registers and a register accessed via read, fetch&add and compare&swap primitives. Even though an object type that supports these three primitives is not traditionally used, current architectures support this combination of primitives. Since compare&swap has infinite consensus number, it is an open question whether the consensus number of an n-reader n-auditor atomic register with atomic audit is n or more.

7 Regular Audit

A multi-reader atomic register with multi-auditor regular audit can be implemented using only single-writer multi-reader atomic registers, with a straightforward approach: during a read operation each reader leaves a trace in a register of all the values they read.

The algorithm uses several atomic registers. A swmr atomic register R_v is shared between the writer and the readers. This register is used by the writer to write a new value and by the readers to access it. In addition to R_v, each reader p_i shares a swmr atomic register $R_a[i]$ with the auditors. This register is used by p_i to communicate to the auditor all the values it read from the register.

In a read, a reader p_i reads a value from R_v and stores it in *read_result*. Then, it adds v together with its identifier in *read_log*, and writes *read_log* in the register $R_a[i]$ it shares with the auditors. Finally, it returns *read_result*. In a write, the writer simply writes v in R_v. In an audit, the auditor simply reads from all the swmr register R_a of each reader, combines it with the result in *audit_result*, and returns.

The pseudocode appears in Algorithm 7. In the full version, we prove:

▶ **Theorem 21.** *Algorithm 7 implements a single-writer multi-reader atomic register with multi-auditor regular audit.*

We remark that this algorithm can be specialized to get *single*-writer *single*-reader registers, and extended to get *multi*-writer multi-reader atomic registers. In both cases, the algorithm can provide regular audit for one or many auditors.

8 Discussion

This paper studies the synchronization power of auditing an atomic read / write register, in a shared memory system. We consider two alternative definitions of the audit operation, one that is atomic relative to the read and write operations, and another that is regular. The first

Algorithm 7 Implementation of a single-writer multi-reader atomic register with *multi-auditor regular audit* using only read and write.

```
Shared Variables:
$R_v$, swmr atomic register, initially $v_0$
$\forall i \in [0, n-1]$ $R_a$[i] swmr atomic register with writer $p_i$. Initially $\perp$

Local Variables:                                        ▷ Pseudo code for reader $p_i$
$read\_log$, initially $\emptyset$          ▷ tuples $(p_j, v)$, for each value $v$ read by $p_j$
$read\_result$, initially $\perp$                           ▷ value the reader read
Read()
    $read\_result \leftarrow R_v.read()$
    $read\_log.add(p_i, read\_result)$
    $R_a[i].write(read\_log)$
    return $read\_result$

Write($v$)                                           ▷ Pseudo code for the writer $p_0$
    $R_v.write(v)$

Local Variables:                                       ▷ Pseudo code for auditor $p_k$
$audit\_result$, initially $\emptyset$    ▷ tuples $(p, v)$, with $p$ the reader and $v$ the value.
Audit()
    for $1 \leq j < n$
        $audit\_result.add(R_a[j].read())$
    return $audit\_result$
```

definition is shown to have a strong synchronization power, allowing to solve consensus; the number of processes that can solve consensus corresponds to the number of processes that can read and audit the register. We also implement an atomic audit operation, using swap and fetch&add for a single auditor (and multiple readers) or a single reader (and multiple auditors), and compare&swap when there are multiple readers and multiple auditors. On the other hand, the weaker, regular audit can be implemented from ordinary reads and writes.

We studied single-writer registers and leave the interesting question of registers with multiple writers to future work.

It is also interesting to investigate the precise relationship between auditable registers and DenyList objects [9], which record which processes accessed a resource and how many times. We conjecture that there are reductions between DenyList and and registers with atomic audit. The precise reductions would also explain why some variants of an auditable register with atomic audit has consensus number 2, a phenomenon that does not happen with DenyList. A reduction to DenyList, and more specifically, to the Proof-List object of [9], might also yield an implementation of an auditable register with atomic audit, and n readers and auditors, from n-consensus objects.

From a practical point of view, our results indicate that determining the precise requirements from auditable registers in real systems can be subtle, since a too-strong definition would incur high synchronization cost.

References

1 *California Consumer Privacy Act.* State of California Department of Justice `https://oag.ca.gov/privacy/ccpa`.

2 Identity Theft Resource Center. At mid-year, U.S. data breaches increase at record pace. In *ITRC*, 2018.

3 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. As easy as ABC: optimal (a)ccountable (b)yzantine (c)onsensus is easy! In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 – June 3, 2022*, pages 560–570. IEEE, 2022. `doi:10.1109/IPDPS53621.2022.00061`.

4 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Seredinschi. Crime and punishment in distributed byzantine decision tasks. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*, pages 34–44. IEEE, 2022. `doi:10.1109/ICDCS54860.2022.00013`.

5 Vinicius Vielmo Cogo and Alysson Bessani. Brief Announcement: Auditable Register Emulations. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 53:1–53:4, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPICS.DISC.2021.53`.

6 Antonella Del Pozzo, Alessia Milani, and Alexandre Rapetti. Byzantine auditable atomic register with optimal resilience. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 121–132. IEEE Computer Society, 2022. `doi:10.1109/SRDS55811.2022.00020`.

7 Denise Demirel, Stephan Krenn, Thomas Lorünser, and Giulia Traverso. Efficient and privacy preserving third party auditing for a distributed storage system. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 88–97. IEEE, 2016. `doi:10.1109/ARES.2016.88`.

8 Dipa Dharamadhikari and Sharvaree Tamne. Public auditing schemes (pas) for dynamic data in cloud: A review. In *International Conference on Smart Trends for Information Technology and Computer Communications*, pages 186–191. Springer, 2017. `doi:10.1007/978-981-13-1423-0_21`.

9 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist. In *to appear in 37th International Symposium on Distributed Computing, DISC 2023*, 2023. `doi:10.4230/LIPICS.DISC.2023.21`.

10 *General Data Protection Regulation*. Regulation (EU) 2016/679 `https://gdpr-info.eu/`.

11 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007. `doi:10.1145/1294261.1294279`.

12 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. `doi:10.1145/114005.102808`.

13 V Kavya, R Sumathi, and AN Shwetha. A survey on data auditing approaches to preserve privacy and data integrity in cloud computing. In *International conference on sustainable communication networks and application*, pages 108–118. Springer, 2019. `doi:10.1007/978-3-030-34515-0_12`.

14 Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986. `doi:10.1007/BF01786228`.

15 Anh Le, Athina Markopoulou, and Alexandros G Dimakis. Auditing for distributed storage systems. *IEEE/ACM Transactions on Networking*, 24(4):2182–2195, 2015. `doi:10.1109/TNET.2015.2450761`.

16 Bo Li, Qiang He, Feifei Chen, Hai Jin, Yang Xiang, and Yun Yang. Auditing cache data integrity in the edge computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1210–1223, 2020. `doi:10.1109/TPDS.2020.3043755`.

17 Jin Li, Kui Ren, and Kwangjo Kim. A2be: Accountable attribute-based encryption for abuse free access control. *Cryptology ePrint Archive*, 2009. URL: `http://eprint.iacr.org/2009/118`.

18 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable Fetch&Add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.OPODIS.2021.29`.

19 *Personal Information Protection Law of the People's Republic of China*. 30th meeting of the Standing Committee of the 13th National People's Congress of the People's Republic of China on August 20.

20 Antonella Del Pozzo and Thibault Rieutord. Fork accountability in tenderbake. In Sara Tucci Piergiovanni and Natacha Crooks, editors, *5th International Symposium on Foundations and Applications of Blockchain 2022, FAB 2022, June 3, 2022, Berkeley, CA, USA*, volume 101 of *OASIcs*, pages 5:1–5:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/OASICS.FAB.2022.5`.

21 Hui Tian, Yuxiang Chen, Chin-Chen Chang, Hong Jiang, Yongfeng Huang, Yonghong Chen, and Jin Liu. Dynamic-hash-table based public auditing for secure cloud storage. *IEEE Transactions on Services Computing*, 10(05):701–714, 2017. `doi:10.1109/TSC.2015.2512589`.

22 Boyang Wang, Baochun Li, and Hui Li. Oruta: Privacy-preserving public auditing for shared data in the cloud. *IEEE transactions on cloud computing*, 2(1):43–56, 2014. `doi:10.1109/TCC.2014.2299807`.

23 Jiaojiao Wu, Yanping Li, Fang Ren, and Bo Yang. Robust and auditable distributed data storage with scalability in edge computing. *Ad Hoc Networks*, 117:102494, 2021. `doi:10.1016/J.ADHOC.2021.102494`.

24 Yinghui Zhang, Robert H Deng, Shengmin Xu, Jianfei Sun, Qi Li, and Dong Zheng. Attribute-based encryption for cloud computing access control: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–41, 2020. `doi:10.1145/3398036`.