Auditable Shared Objects: From Registers to Synchronization Primitives

Hagit Attiya
Technion – Israel Institute of Technology, Haifa, Israel

Antonio Fernández Anta
IMDEA Software Institute & IMDEA Networks Institute, Madrid, Spain

Alessia Milani
Aix Marseille Univ, CNRS, LIS, Marseille, France

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

Corentin Travers
Aix Marseille Univ, CNRS, LIS, Marseille, France

Abstract

Auditability allows to track operations performed on a shared object, recording who accessed which information. This gives data owners more control on their data. Initially studied in the context of single-writer registers, this work extends the notion of auditability to other shared objects, and studies their properties.

We start by moving from single-writer to *multi-writer* registers, and provide an implementation of an *auditable n-writer m-reader read / write register*, with $O(n + m)$ step complexity. This implementation uses $(m + n)$-sliding registers, which have consensus number $m + n$. We show that this consensus number is necessary. The implementation extends naturally to support an *auditable load-linked / store-conditional* (LL/SC) shared object. LL/SC is a primitive that supports efficient implementation of many shared objects. Finally, we relate auditable registers to other access control objects, by implementing an *anti-flickering deny list* from auditable registers.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Auditability, Wait-free implementation, Synchronization power, Distributed objects, Shared memory, LL/SC, Deny List

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.8

Related Version *Full Version*: `https://arxiv.org/abs/2508.14506` [5]

Funding *Hagit Attiya*: Supported by the Israel Science Foundation (22/1425 and 25/1849).
Antonio Fernández Anta: Supported by project PID2022-140560OB-I00 (DRONAC) funded by MICIU/AEI /10.13039/501100011033 and ERDF, EU.
Alessia Milani: Supported in part by ANR project TRUSTINCloudS (ANR-23-PECL-0009).
Corentin Travers: Supported in part by ANR project DUCAT (ANR-20-CE48-0006).

1 Introduction

The ability to track operations on shared objects is a fundamental feature in distributed systems. *Auditability*, in particular, enables data owners to monitor access to their data by recording who accessed which information. This mechanism provides a robust alternative to traditional access control mechanisms, shifting the emphasis from access restriction to post-hoc accountability. Auditability is helpful for preserving data privacy, as it can be used after a data breach, to enforce accountability for data access. This is particularly useful in shared, remotely accessed storage systems, where, for instance, understanding the extent of a data breach can help mitigate its impact.

Auditability has been initially studied in the context of non-atomic replicated storage [8] and single-writer atomic registers [6]. This work explores auditability for *other shared objects*, considering the following key questions: (1) What types of base objects are necessary and sufficient to implement auditable *multi-writer* atomic registers? (2) How does auditability impact the design of *common synchronization primitives*? (3) Can auditable registers be leveraged to construct more complex *access control mechanisms*?

We begin to answer these questions by presenting a wait-free, linearizable implementation of an auditable n-writer m-reader read/write register (abbreviated as (m, n)-auditable register), with linear (in $n + m$) step complexity. To achieve these properties, our algorithm uses sliding registers [19]: a *k-sliding register* keeps an ordered list of the latest k values written to it. If each reader and writer writes at most once in an $(m + n)$-sliding register, the sliding register can be used to uniquely order the operations. This implies a linearization of the read and write operations on the auditable register. The key challenge is, therefore, how to deal with multiple read and write operations on the auditable register. The backbone of our (m, n)-auditable register implementation is a sequence of $(m + n)$-sliding registers, each associated with a different value written to the auditable register. The sliding registers are used to agree which write operation will set the next value of the register, and to track which readers have read this value. An additional challenge is to be able to efficiently find the "current" sliding register when reading and writing, to maintain bounded step complexity.

The consensus number of $(m + n)$-sliding registers is exactly $m + n$. (I.e., they allow to solve consensus among exactly $m + n$ processes.) Thus, consensus number $m + n$ is sufficient for implementing an (m, n)-auditable register. We prove that it is also necessary, by showing that an (m, n)-auditable register can be used to solve consensus among $m + n$ processes.

Beyond registers, we explore auditability for more expressive synchronization primitives, like *load-linked/store-conditional* (LL/SC) objects [17]. The LL operation reads a value from a memory location, while the SC operation writes a new value to the same location only if no process has written to it since the LL operation. If another process has changed the value, the SC fails, requiring the operation to be retried. LL/SC enables the efficient construction of a wide range of non-blocking data structures [10]. We leverage the algorithmic ideas of our auditable register to construct an algorithm that implements an LL/SC object for n processes, using $2n$-sliding register.

Finally, we relate auditable registers to Deny Lists [11], showcasing how auditability can be leveraged to build security primitives. *Allow Lists* and *Deny Lists*, as defined by Frey, Gestin, and Raynal [11], are two access control objects that give designated processes (called *managers*) the ability to grant and/or revoke access rights for a given set of resources. Their work specifies and investigates the synchronization power of Allow Lists and Deny Lists; the latter in two flavors, with and without an *anti-flickering* property. (An anti-flickering Deny List ensures that transient revocations do not undermine long-term access control policies.)

We give a relatively simple specification of an *immediate* Deny List, which is stronger than the anti-flickering Deny List of Frey et al., and show that it can be efficiently implemented from auditable registers. The step complexity of the resulting implementation (polynomially) depends on the number of resources and the number of processes. In contrast, while being wait-free, the step complexity of the anti-flickering Deny List implementation in [11] grows with the number of operations, and is, essentially, unbounded.

Related Work

Auditability was introduced by Cogo and Bessani [8] in the context of replicated registers. They considered a register as an abstraction for distributed storage that provides read and write operations to clients. Cogo and Bessani define auditability in terms of two properties:

completeness ensures that all readers' data access are detected, while *accuracy* ensures that readers who do not access data are not wrongly incriminated. They present an algorithm to implement an auditable *regular* (non-atomic) register, using $n \geq 4f + 1$ atomic read/write shared objects, f of which may be faulty (writers and auditors fail only by crashing; faulty readers may be Byzantine). Their 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. Each low-level shared object keeps a trace of each access, and in order to read, a process has to collect sufficiently many pieces of information in many low-level shared objects, which allows to audit the read.

In asynchronous message-passing systems where f processes can be Byzantine, Del Pozzo, Milani, and Rapetti [9] study the possibility of implementing an atomic auditable register. They prove that without communication between servers, auditability requires at least $4f + 1$ servers. They also show that allowing servers to communicate with each other admits an auditable atomic register with optimal resilience of $3f + 1$.

The auditability definition of [8] is tightly coupled with their multi-writer, multi-reader register emulation in a replicated storage system using an information-dispersal scheme. An implementation-agnostic auditability definition was later proposed by Attiya et al. [6], based on collectively linearizing read, write, and audit operations. They show that auditing adds power to reading and writing, as it allows to solve consensus, implying that auditing requires strong synchronization primitives. They also give several implementations that use non-universal primitives (like *swap* and *fetch&add*), for a single writer and either several readers or several auditors (but not both).

Recent work [4] have extended auditability to ensure that even a curious (but honest) reader cannot effectively read data without being audited; in fact, curious readers cannot even audit other readers. The paper provides implementations of several auditable objects, including registers. However, the constructions depend on universal primitives, including *compare&swap*, in contrast to our constructions, which only employ sliding registers, whose consensus number is bounded.

Alhajaili and Jhumka [1] study an interesting variant of auditability with malicious processes, with and without a trusted party. Their definition of auditability has to do with the ability of determining if a system runs as specified, and determine the incorrect behavior in case it does not. This involves systematically tracking and recording system activities to facilitate fault detection and diagnosis. The authors propose methodologies for integrating auditability into system design, emphasizing its role in simplifying the debugging process and improving overall system robustness.

Our algorithms (like most prior work) maintain a large amount of auditing information, which grows with the number of operations performed in an execution. Hajisheykhi, Roohitavaf, and Kulkarni [13] investigate how to reduce the space used for saving auditing data (to be used to restore the state after a fault), by leveraging causal dependencies among them. They propose protocols for ensuring accountability in distributed systems when auditable events – deviations from standard protocols – occur. They introduce two self-stabilizing protocols: an unbounded state space protocol that propagates auditable events across all nodes and a more efficient bounded state space protocol that achieves the same awareness without increasing resource demands. Their approach ensures that before a system can recover from an auditable event, all processes are aware of it, preventing unauthorized restorations and reinforcing accountability.

Access control objects regulate access to resources by defining policies for granting or denying permissions. Some of the most common access control mechanisms include Access Control Lists [21], which specify allowed or denied actions for individual users or processes,

and Capability-Based Access Control [18], where access is granted through transferable tokens rather than predefined rules. Role-Based Access Control [20] simplifies management by assigning permissions to roles instead of individuals, making it widely used in enterprise environments, while Attribute-Based Access Control [15] dynamically evaluate attributes such as user location, device type, and time of access. AllowLists and DenyLists [11] further control access by explicitly specifying permitted or blocked users or processes, often used in security filters and authentication systems.

An area related to auditability is *data provenance* (also called *lineage*) in databases [7, 12], which involves tracking the origin and transformations of data, focusing on its lineage across different processes or systems. Provenance is concerned with the integrity, quality, and reproducibility of data as it moves through various stages, such as aggregation or computation. While both auditability and provenance track interactions with data, auditability is more focused on recording who performed an operation, primarily for access control and security, while provenance aims to provide transparency about the history and transformations of the data for purposes such as verification and accountability in data analysis.

Summary of Our Contributions and Organization of the Paper

- We extend auditability from single-writer to multi-writer registers and characterize the necessary and sufficient consensus number needed for their implementation.
- We introduce and implement auditable LL/SC objects, demonstrating auditability can be provided beyond read / write operations.
- We construct an anti-flickering Deny List object from auditable registers, illustrating the connection between auditability and access control mechanisms.

Section 2 presents our model and formally defines auditable shared objects. Sections 3 and 4 prove the necessary and sufficient conditions for auditable multi-writer registers, respectively. Section 5 describes how to implement auditable LL/SC objects. Section 6 presents the algorithm to implement an anti-flickering Deny List object with auditable registers. Finally, Section 7 concludes with open questions and future directions. Proofs omitted due to space constraints can be found in the full version [5].

2 Definitions

We use a standard model, in which a set of processes $p_1, \ldots, p_n$, communicate through a shared memory consisting of *base objects*. The base objects are accessed with *primitive operations*. In addition to atomic registers, our implementations use k-sliding registers, whose consensus number is exactly k [19]. Specifically, a *k-sliding register* [19] stores the sequence of the last k values written to it (or the last x values when only $x < k$ values have been written). A *write* with input v appends v at the end of the sequence and removes the first one if the sequence is already of size k. A *read* operation returns the current sequence. A standard read / write register is a 1-sliding register.

An *implementation* of a (high-level) object T specifies a program for each process and each operation of the object T; when receiving an *invocation* of an operation, the process takes *steps* according to this program. Each step by a process consists of some local computation, followed by a single primitive operation on a base object. The process may change its local state after a step, and it may return a *response* to the operation of the high-level object.

In order not to confuse operations performed on the implementation of the high-level object T and primitives applied to base objects, the former are denoted with capital letters and the later in normal font.

A *configuration* C specifies the state of every process and of every base object. An *execution* α is an alternating sequence of configurations and events, starting with an *initial configuration*; it can be finite or infinite. An operation *completes* in an execution α if α includes both the invocation and response of the operation; if α includes the invocation of an operation, but no matching response, then the operation is *pending*. An operation *op precedes* another operation op' in α if the response of *op* appears before the invocation of op'.

A *history* H is a sequence of invocation and response events. The notions of *complete*, *pending* and *preceding* operations extend naturally to histories.

The standard correctness condition for concurrent implementations is *linearizability* [14]: intuitively, it requires that each operation appears to take place instantaneously at some point between its invocation and its response. Formally:

▶ **Definition 1.** *Let $\mathcal{A}$ be an implementation of an object T. An execution α of $\mathcal{A}$ is* linearizable *if there is a sequential execution $\lambda(\alpha)$ (a* linearization *of the operations on T in α) such that:*

- *$\lambda(\alpha)$ contains all complete operations in α, and a (possibly empty) subset of the pending operations in α (completed with response events),*
- *If an operation op precedes an operation op' in α, then op appears before op' in $\lambda(\alpha)$, and*
- *$\lambda(\alpha)$ respects the sequential specification of the high-level object T.*

$\mathcal{A}$ is linearizable *if all its executions are linearizable.*

An implementation is *wait-free* if, whenever there is a pending operation by process p, this operation returns in a finite number of steps by p.

An *auditable register* supports, in addition to the standard READ and WRITE operations, also an AUDIT operation that reports which values were read by each process [6]. An AUDIT has no parameters and it returns a set of pairs, (j, v), where j is a process id, and v is a value of the register. A pair (j, v) indicates that process p_j has read the value v. The sequential specification of an auditable register enforces, in addition to the usual specification of READ and WRITE operations, that a pair appears in the set returned by an AUDIT operation if and only if it corresponds to a preceding READ operation.

We implement a *load-linked / store-conditional* (LL/SC) variable, supporting the following operations: LL(x) returns the value stored in x, and SC(x, new) writes the value *new* to x, if it was not written since the last LL(x) performed by the process; otherwise, x is not modified. SC returns *true* if it writes successfully, and *false* otherwise. An *auditable LL/SC variable adds an* AUDIT *operation, whose sequential specification* returns a set of process-value pairs, corresponding to preceding LL operations.

3 Consensus number of n-Writer, m-Reader Auditable Register $\geq m + n$

An (m, n)-auditable register can be written by n processes, read by m processes, and be audited by all processes. An audit operation returns a set of pairs (p, v) where p is a process id, and v is a value. This set holds the values returned by the read operations that precede the audit.

▶ **Theorem 2.** *For every pair of integers $m, n > 0$, there is an $(m + n)$-process consensus algorithm using (m, n)-auditable registers.*

Proof. The proof is by induction on $\ell = n + m$. The base case, $\ell = 2$, is proved in [6, Proposition 19].

For the induction step, assume the lemma holds for all pairs of values n', m', such that $n' + m' = \ell \geq 2$, and we prove it for $\ell + 1 > 2$. Pick $m > 0, n > 0$ such that $n + m = \ell + 1$; note that $m, n \leq \ell$. To solve consensus among $\ell + 1$ processes $p_1, \ldots, p_{\ell+1}$, we partition the processes into two sets: R (the *readers*) of size m, and W (the *writers*) of size n. Since $m > 0$ and $n > 0$, both sets R and W are non-empty.

Each process $p_i, 1 \leq i \leq \ell + 1$ has a standard single-writer multi-reader register S_i. By the induction hypothesis, since $n \leq \ell$, the n writers can agree on one of their proposals with an n-process consensus that uses $(1, n-1)$-auditable registers. Similarly, the m readers can agree on one of their proposal using $(m-1, 1)$-auditable registers. Each reader and writer process p_i writes the value agreed upon in its register S_i.

We use now one (m, n)-auditable register, AR, whose initial value is $\perp$. Writers and readers access AR to select one of the two consensus values. Each writer $\in W$ writes $\top$ to AR and each reader $\in R$ performs a read operation on AR. Writers and readers then audit AR. If the set returned is empty or contains no pair $(p_r, \perp)$ where $r \in R$, then the first operation performed on AR is a write and the writers win. Then, the decision for consensus can be read from at least one of the registers of the writers. Otherwise, the audit operation returns a set containing a pair $(p_r, \perp)$ with $r \in R$. In that case, the first operation performed on AR is a read operation and the readers win. As in the previous case, the decision for consensus can be found by reading the readers' registers. ◀

4 Implementing an n-Writer m-Reader Auditable Register Using $(n + m)$-Sliding Registers

We present a wait-free and linearizable implementation of an n-writer m-reader auditable register; it can support any number of auditors. The implementation uses $(m + n)$-sliding registers. Since the consensus number of (m, n)-auditable registers is $m + n$ (Theorem 2), objects with consensus number $\geq m + n$, like $(m + n)$-sliding registers, are required. (Objects with consensus number ∞, like *compare&swap*, can also be used [4], but our goal is to use objects with the minimal consensus number.) Our main result is:

▶ **Theorem 3.** *There is a wait-free linearizable implementation of an m-reader, n-writer auditable register from $(m + n)$-sliding registers. The step complexity of each operation is in $O(m + n)$.*

4.1 The Algorithm

An implementation of an auditable register has to keep track of the latest value written to the register as well as, for each written value, its set of readers. This can be easily achieved using a single sliding register SLR, provided that its window is unbounded. Such a register hence stores the complete sequence of values written to it, ordered by the oldest first but has infinite consensus number. To perform a WRITE(v), a writer simply writes v to SLR. For auditing purpose, a reader p_i first writes its identifier i to the sliding register, before reading it. The value returned by this READ is then the nearest non-identifier value in the sequence read from SLR previous to the identifier i written by p_i. The reader set of each value can easily be inferred from the sequence stored in SLR. For a value v in the sequence, its reader set is the set of processes whose identifiers follows v, and are before the first non-identifier value that succeeds v, if any.

Our implementation (Algorithms 1 and 2) is based on this simple idea, but instead of a single sliding register with an unbounded window, we use an unbounded array $SLR[-1, \ldots]$ of sliding registers, each with a bounded window of size $m + n$. In order not to confuse

identifiers and written values, WRITE operations insert into the sliding registers *w-tuples* of the form (w, j, v, h), where j the identifier of the writer, v is the input value of the WRITE operation, and h a *helping set* of readers' identifiers (whose role will be explained later).

Each sliding register $SLR[x]$ contains initially the empty sequence (), except the first $SLR[-1]$ whose sequence contains the w-tuple $(\mathsf{w}, j_0, v_0, \emptyset)$ where v_0 is the initial value of the auditable register and j_0 an arbitrary writer's identifier. At any point in the execution, the current value v of the auditable register is found in the sliding register with highest index x whose sequence contains a w-tuple. Specifically, v is the value contained in the *first* w-tuple in the sequence stored in $SLR[x]$. Similarly to the basic implementation sketched above, readers of this value (if any) are the processes whose identifiers appear *before* any w-tuple in the *next* sliding register $SLR[x+1]$.

Therefore, reading or writing the auditable register involves finding the *valid* sliding register (that is, one that does not contain a w-tuple) with lowest index. If this is not done with care, progress of some operation may be lost. For example, the same writer may be always the first to write in each sliding register (i.e., each non empty sequence in any sliding register starts with a w-tuple posted by this writer), thus preventing READ operations from completing or other writers from changing the value of the auditable register. We also have to make sure that each writer or reader writes at most once to each sliding register. Otherwise, the auditing may become inaccurate (as readers identifiers may be removed from some sequence), or the current value of the auditable register may be lost.

To solve theses challenges, the implementation combines the following ideas:

First, we observe that a WRITE operation can terminate after writing in a given sliding register $SLR[x]$, even if its corresponding w-tuple is not the first in the sequence held in $SLR[x]$, provided that that it is concurrent with the WRITE operation that writes first its w-tuple into $SLR[x]$. Indeed, in that case, the WRITE operations that are late to post their w-tuple may be linearized immediately *before* the WRITE whose w-tuple is first. Accordingly, the value, denoted v_x, in the first w-tuple in the sequence stored in $SLR[x]$ is said to be *visible*. The values in the other w-tuple are never returned by any READ operation, and are thus *invisible*.

Second, we use a *max register* M to store the current smallest index $widx$ of the still valid sliding registers. Recall that a max register retains the largest value written to it; wait-free and linearizable max registers can be implemented from atomic read/write registers with linear step complexity [3]. A WRITE thus starts by retrieving this index from M, writes its w-tuple in $SLR[widx]$, and finally updates M with the new index $widx+1$. This ensures that, if several WRITE operations post w-tuples in the same sliding register $SLR[x]$, they are concurrent.

In addition, M stores an m-vector $ridx$ of indexes of sliding registers (which is part of the helping mechanism described next) and, for convenience, the $auditset$ of the values written whose set of readers is already completely determined (the set of readers of each visible value v_x, for $x < widx$). Triples $(widx, ridx, auditset)$ are ordered by their first field. An AUDIT operation therefore reads M, and for $M.widx = x$, gets the *definitive* readers of values $v_{x'}, x' \leq x-2$ from $M.auditset$, to which it adds the possibly *non-definitive* set of readers of v_{x-1} by reading $SLR[x]$.

Finally, a helping mechanism is used to make sure that READ operations are wait-free. The set of readers of the visible value v_x (that appears in the first w-tuple stored in $SLR[x]$) are the processes whose identifiers are before any w-tuple in $SLR[x+1]$ *or* are in the helping set h of the first w-tuple of $SLR[x+1]$. An additional array H indicates, for each reader, the index of the latest sliding register in which it attempted to write its identifier before

Algorithm 1 Multi-writer auditable register: READ and WRITE and AUDIT.

```
1: shared variables
2:     M: a max-register storing a triple (widx, ridx, auditset) ordered by their first field
3:         widx initially 0                                          ▷ index of sliding registers
4:         ridx array of size m, initially [−1, . . . , −1]
                  ▷ ridx[i] highest index of sliding register in which a READ by p_i is recorded
5:         auditset set of pair (process,value), initially ∅
6:     SLR[−1, 0, .. + ∞]: unbounded array of (m + n)-sliding registers,
           initially, SLR[−1] = ((w, j_0, v_0, ∅)), SLR[ℓ] = () for all ℓ ≥ 0
7:     H[1..m]: array of SWMR register, one per reader, initially [−1, . . . , −1]
8: local variables: reader
9:     lsr ← −1; lval ← ⊥                ▷ index of last sliding register read and last value read
10: function READ( )                                  ▷ code for reader process p_i, i ∈ {1, . . . , m}
11:     if lsr ≥ 0 then window ← SLR[lsr].read()          ▷ check for new WRITE since last READ
12:         if there is no w-tuple (w, _, _, _) in window then return lval          ▷ no new WRITE
            (widx, ridx, auditset) ← M.read()      ▷ new WRITE, check if it needs help to complete
13:         if widx = lsr then
14:             for each j ∈ READERS(window) do
15:                 ridx[j] ← lsr; auditset ← auditset ∪ {(j, lval)}
16:             M.writeMax(lsr + 1, ridx, auditset)
17:     repeat
18:         (widx, ridx, auditset) ← M.read()
19:         if ridx[i] > lsr then                                                  ▷ found help
20:             lsr ← ridx[i]; lval ← GETVALUE(lsr − 1) return lval
21:         lsr ← widx; H[i].write(lsr)                ▷ catch up with writers, announce attempt
22:         SLR[lsr].write(i); window ← SLR[lsr].read(); lval ←GETVALUE(lsr − 1)
23:         if ∃(w, _, _, _) ∈ window then          ▷ help corresponding WRITE to complete
24:             for each j ∈ READERS(window) do
25:                 ridx[j] ← lsr; auditset ← auditset ∪ {(j, lval)}
26:             M.writeMax(lsr + 1, ridx, auditset)
27:     until i ∈ READERS(window)
28:     return lval
29: function WRITE(v)                                 ▷ code for writer p_j, j ∈ {1, . . . , n}
30:     widx, ridx, auditset ← M.read(); to_help ← ∅
31:     for all i ∈ {1, . . . , m} do aidx_j ← H[j].read()          ▷ help ongoing read operations
32:         if ridx[j] < aidx_j then                                  ▷ reader p_j may need help
33:             window ← SLR[aidx_j].read()
34:             if j ∉ READERS(window) then to_help ← to_help ∪ {j}
35:     SLR[widx].write(w, j, v, to_help);                                  ▷ announce new write
36:     window ← SLR[widx].read(); val ← GETVALUE(widx − 1)
37:     for each j ∈ READERS(window) do ridx[j] ← widx; auditset ← auditset ∪ {(j, val)}
38:     M.writeMax(widx + 1, ridx, auditset); return
39: function AUDIT( )
40:     widx, _, auditset ← M.read()                  ▷ readers of val with seq. num ≤ widx − 2
41:     window ← SLR[widx].read(); val ← GETVALUE(widx − 1)
42:     auditset ← auditset ∪ {(j, val) : j ∈ READERS(window)}      ▷ readers of val with seq. num
    widx − 1
43:     return auditset
```

Algorithm 2 Multi-writer auditable register: auxiliary functions GETVALUE and READERS.

```
44: function GETVALUE(sn)
45:     window ← SLR[sn].read(); let (w, id, val, _) be the first w-tuple (w, _, _, _) in window
46:     return val
47: function READERS(window)
48:     readers ← {j : there is no w-tuple (w, _, _, _) preceding j in window}
49:     if ∃(w, _, _, _) ∈ window then
50:         let (w, id, _, h) be the 1st (w, _, _, _) in window; readers ← readers ∪ h
51:     return readers
```

any w-tuple. The array $ridx$ (stored, as seen above, in the max register M), records for each reader the highest index of a sliding register in which it has received help (that is, the identifier of the reader is included in the helping set of the first w-tuple of that sliding register). A writer p_j hence determines if a given reader p_i needs help by comparing $H[i]$ and $M.ridx[i]$. If $M.ridx[i] < H[i]$, p_i has an ongoing READ operation, and i is therefore added to the helping set of the w-tuple of the writer. Similarly, reader p_i discovers if it has received help by comparing $M.ridx[i]$ with the index of the latest sliding register in which it writes its identifier.

Hence, besides w-tuples, a given sliding register $SLR[x]$ may also include identifiers i of readers, with the following meaning:

1. If i appears *before* the first w-tuple in $SLR[x]$, then p_i's READ returns v_{x-1}, which is the value stored in the first w-tuple in $SLR[x-1]$.
2. If i appears in the helping set of the first w-tuple in $SLR[x]$, then p_i's READ also returns v_{x-1} as in the previous case.
3. If i appears *after* the first w-tuple, then it is too late and fails to read v_{x-1}.

The reader processes satisfying Cases (1) and (2) are said to be *recorded* in $SLR[x]$. The function READERS in Algorithm 2 returns the ids of readers recorded in a sliding register (which is used, in particular, to update the audit set). The processes satisfying Case (2) are called *helped* in $SLR[x]$.

Regarding Case (3), we prove that after at most two failed attempts, a reader p_i receives help, or succeeds in appearing before any w-tuple in a sliding register. The algorithm also ensures that a READ operation is helped at most once. Before an attempt to write its id into $SLR[x]$, reader p_i knows, by checking if $ridx[i] \geq H[i]$, if it has already received help. If this is the case, it directly returns the corresponding value without updating $H[i]$.

4.2 Proof of Correctness

In this section, we prove Theorem 3, starting with basic properties. We show that each operation returns within $O(m+n)$ of its own steps, and then explain how to construct a sequential execution λ that contains all completed operations in an execution α, and some pending operations. Lemma 13 shows that λ preserves the real-time order between operations in α. By Lemma 14, Lemma 15, and Lemma 16, λ is a sequential execution of a m-reader, n-writer auditable register.

To proceed with the detailed proofs, fix a finite execution α of the algorithm. We start with some basic properties. Lemma 4, whose proof is in the full version, shows that that each sliding register retains the complete history of the write applied to it.

▶ **Lemma 4.** *Each process applies at most one* write *to $SLR[k]$, for any $k \geq 0$,*

The max register M stores a triple $(widx, ridx, auditset)$, where $ridx$ is a m-vector and $audiset$ a set of (process,value) pairs. Triples are ordered in increasing order of their first element. At any point in the execution, the triple stored in M is thus a triple with the largest first member written to M. We also partially order m-vectors as follows: $ridx \leq ridx'$ if and only $\forall i \in \{1, \ldots, m\}, ridx[i] \leq ridx'[i]$.

▶ **Proposition 5.** *The successive values of $M.widx$ are $0, 1, 2, \ldots$.*

In the full version, we prove the following lemma by induction on k.

▶ **Lemma 6.** *Suppose* writeMax$(k, iv, _)$ *and* writeMax$(k', iv', _)$ *are applied to M. (1) if $k = k'$ then $iv = iv'$ and, (2) if $k < k'$ then $iv \leq iv'$.*

By Proposition 5, the successive values of $M.widx$ are $0, 1, \ldots$ By Lemma 6(1), $M.ridx$ remains the same while $M.widx$ does not change. Lemma 6(2) implies that the successive vectors in $M.ridx$ are ordered and form an increasing sequence.

Let $WIDX$ denote the highest index of a sliding register to which a w-tuple has been written. That is, at the end of a finite execution α, $WIDX = k$ if and only a w-tuple was written in $SLR[k]$, and no w-tuple was written in $SLR[k']$, for any $k' > k$. When $WIDX$ is changed to $k + 1$ as a result of some process p applying write$(\mathsf{w}, _, _, _)$ to $SLR[k + 1]$, $M.widx \geq k + 1$. Indeed, before writing a w-tuple to $SLR[k + 1]$, p has read k from $M.widx$ in line 30. Observe also that when $M.widx$ is changed from k to $k + 1$ by some process p, $M.widx \geq k$. Indeed, before applying M.writeMax$(k + 1, _, _)$, p reads a sequence from $SLR[k]$ that contains a w-tuple (line 11 or line 22, p is performing a READ), or has written a w-tuple to $SLR[k]$ (line 35, p is performing a WRITE). This implies that $M.widx$ is always in $\{WIDX, WIDX - 1\}$, which shows:

▶ **Proposition 7.** *For every $x \geq 0$, if $SLR[x]$ is accessed then $SLR[x - 1]$ contains a w-tuple.*

Combining these observations with Lemma 6(2), we have:

▶ **Lemma 8.** *For some $k \geq 0$, the finite execution α can be written as either $D_0\rho_0E_0\mu_1D_1\rho_1E_1 \ldots \mu_kD_k$ or $D_0\rho_0E_0\mu_1D_1\rho_1E_1 \ldots \mu_kD_k\rho_kE_k$, where:*

- *ρ_ℓ is the first step that writes a w-tuple to $SLR[\ell]$ (applied by a writer, line 35), μ_ℓ is the step that changes $M.widx$ from $\ell - 1$ to ℓ (applied within a READ, line 26 or line 16, or a WRITE, line 38).*
- *in any configuration in D_ℓ, $M.widx = \ell = WIDX + 1$, and in any configuration in E_ℓ, $M.widx = \ell = WIDX$*

Therefore, the first step that writes a w-tuple to $SLR[x]$ is preceded by steps $\rho_0, \ldots, \rho_{x-1}$ that write w-tuples to $SLR[0], \ldots, SLR[x - 1]$, and they are never deleted (by Lemma 4).

Let x_0 be the value of the local variable lsr when READ operation op by process p_i starts.

▶ **Proposition 9.** *If op enters the repeat loop (line 12), then $x_0 < x_1 < x_2 < \ldots$ where x_k, $k > 0$, is the value read from $M.widx$ (line 18) at the beginning of the kth iteration.*

If op does not terminate in the kth iteration of the loop (line 20), after reading $(x_k, \_, \_)$ from M, then x_k is written to $H[i]$ (line 21) before an attempt is made to place i ahead of any w-tuple in $SLR[x_k]$. In the full version, we further prove:

▶ **Lemma 10.** *If x_k is written to $H[i]$ then $i \notin$ READERS(win), for any sequence win in any sliding register $SLR[x], x_0 < x < x_k$.*

Finally, we prove, in the full version, that *op* does not find help in $SLR[x]$, for any $x, x_0 < x < x_1$.

▶ **Lemma 11.** *$i \notin$ READERS(win), for any sequence win in a sliding register $SLR[x]$, $x_0 < x < x_1$.*

An AUDIT operation reads M once and a single sliding register. A WRITE operation applies a single writeMax and a single read to M, reads $O(m)$ registers and writes $O(1)$ registers. Since there is a linearizable implementation of max registers for $(m+n)$ processes using registers with $O(m+n)$ step complexity per operation [3], this implies that the step complexity of a WRITE or a AUDIT operation is in $O(m+n)$.

Lemma 20 (Appendix A) shows that READ operations are also wait-free, by proving that the repeat loop has at most 3 iterations. This is because a writer may place a w-tuple in at most one sliding register without detecting a concurrent READ operation and helping it.

To prove linearizability, let H be the history of READ, WRITE and AUDIT operations in the execution α. For simplicity, we assume that the values written to the register in α are unique. We start by classifying the operations in H. Each classified operation *op* is also associated with an integer $idx(op)$, which is the index of a sliding register.

For READ, we distinguish *silent*, *direct* and *helped* operations. Let *op* be a READ operation by some process p_i. We denote by x_0 the value the local variable *lsr* when *op* starts.

- *rop* is *silent* if it is not the first READ operation by p_i and it immediately returns after reading $SLR[x_0]$ (line 12). This corresponds to the case in which no new WRITE operation has occurred since the last READ by p_i. We set $idx(op) = x_0$.

If there is no $x > x_0$ for which $i \in$ READERS$(SLR[x])$, *op* is *unclassified*. In that case, note that *op* has no response in H. Otherwise, let $x_1 > x_0$ be the smallest index such that $i \in$ READERS$(SLR[x_1])$. We set $idx(op) = x_1$ and say that

- *op* is *direct* if i precedes any w-tuple in $SLR[x_1]$, and *helped* otherwise, as in that case, i appears in the helping set of the first w-tuple in $SLR[x_1]$.

A WRITE(v) operation *op* by some process p_j applies at most one write to a sliding register (line 35). *op* is *unclassified* if does not write to a sliding register. Otherwise, let x be the index of the sliding register *op* writes to. $x = idx(op)$ and we say that

- *op* is *visible* if $(\mathsf{w}, j, v, \_)$ is the first w-tuple written to $SLR[x_1]$, and *hidden* otherwise.

For AUDIT, only operations that have a response in H are classified. Let *op* be an AUDIT operation that terminates. We define $idx(op) = x$, where x is the value read from $M.widx$ in the first step of *op* (line 40). *op* is *non-definitive* if there is no w-tuple in the sequence it reads from $SLR[x]$ (in line 41), and *definitive* otherwise. Indeed, once a w-tuple has been written to $SLR[x]$, the set of readers of v_{x-1} no longer changes, while READ operations may still return v_{x-1} after *op* terminates otherwise (v_{x-1} the value in the first w-tuple in $SLR[x-1]$.). In Appendix A, we prove:

▶ **Lemma 12.** *If an operation op terminates before an operation op′ starts in H', then $idx(op) \leq idx(op')$.*

We define H' that contains every completed operation of H as well as some incomplete operations, to which we add a matching response. We first discard from H every AUDIT and every silent READ invocation without a matching response, as well as every invocation of an unclassified READ and WRITE operation. We then add at the end a response for each remaining READ and WRITE operation that has no response in H. The return value of a READ operation *op* with $idx(op) = x$ is the value *val* in the first w-tuple in $(\mathsf{w}, \_, val, \_)$ in $SLR[x-1]$. Responses are added in arbitrary order.

A linearization $\lambda(\alpha)$ of α is defined in two steps. We first order operations in H' according to their associated index, in ascending order (rule $R0$). We then order operations with the same index. Let $B(x)$ be the set of operations $op \in H'$ such that $idx(op) = x$. These operations are ordered according to the following rules.

R1 We place first silent READ, direct READ and non-definitive AUDIT operations. They are ordered according to the order in which they apply a read (for silent READ and non-definitive AUDIT) or a write (for direct READ) to $SLR[x]$ in α.

R2 We then place helped READ in arbitrary order, followed by the definitive AUDIT operations. The definitive AUDIT are ordered according to the order in which they apply a read to $SLR[x]$ in α.

R3 We next put every hidden WRITE. They are ordered according to the order in which their write to $SLR[x]$ is applied in α.

R4 The (unique) visible WRITE is placed last.

Let λ be the linearization obtained by applying linearization rules $R0$-$R4$ to the operations in H'. In Appendix A, we prove:

▶ **Lemma 13.** *If an operation op terminates before an operation op′ starts in H', then op precedes op′ in λ.*

▶ **Lemma 14.** *If READ operation op in H' returns v, then v is the value written by the last WRITE that precedes op in λ, or the initial value v_0 if there is no such WRITE.*

Finally, we prove (in Appendix A) that a pair (p, v) is in the set returned by an AUDIT operation *op* if and only if there is a READ by p returning v precedes *op*.

▶ **Lemma 15.** *Let aop be an AUDIT operation in H' that returns A. If there is a READ operation rop by process p_j returning v and that precedes aop in λ, $(j, v) \in A$.*

▶ **Lemma 16.** *Let aop be an AUDIT operation in H' whose response contains (j, v). There exists a READ operation by p_j returning v that precedes aop in λ.*

5 Auditable LL/SC from $2n$-Sliding Register

We show how to adapt our auditable register algorithm to implement an auditable LL/SC object for n processes, and an arbitrary number of auditors. An AUDIT operation returns a set of pairs (p, v), representing the values returned by the processes by the LL operations preceding the AUDIT. The implementation uses $2n$-sliding registers, showing:

▶ **Theorem 17.** *There is a wait-free, linearizable implementation of an n-process auditable LL/SC object from $2n$-sliding registers and standard registers with $O(n)$ step complexity per operation.*

Algorithm 3 is essentially the same as Algorithm 1. LL operations are identified with READ operations, and SC with WRITE. At the end of a finite execution α of Algorithm 1, the sequences stored in the sliding registers in the array SLR indicate, for each x, which is the xth value v_x held in the auditable register, and which processes write or read this value. Specifically, given the sequence win_x stored in $SLR[x]$, the first w-tuple (w, j, v, h) in win_x indicates that $v_x = v$, and its writer is p_j. The readers of v_{x-1} are the processes p_i, where $i \in h$, or precedes (w, j, v, h) in win_x. Each other w-tuple $(\mathsf{w}, j', v', \_)$ in win_x

Algorithm 3 n-process LL/SC with auditable LL.

```
1: shared variables
2:     M, H[1..n]: max register and helping array of n SWMR register, initialized as in Algorithm 1
3:     SLR[−1, 0, ...]: unbounded array of 2n-sliding registers, initialized as in Algorithm 1
4: local variables
5:     lsr, val, as in Algorithm 1
6: function LL( )                                   ▷ identical to READ in Algorithm 1
7: function SC(v)                                   ▷ code for process p_i, i ∈ {1, ..., n}
8:     widx, ridx, auditset ← M.read(); to_help ← ∅
9:     if widx > lsr then return false              ▷ a successful SC happened since p_i's last LL
10:    for all i ∈ {1, ..., m} do aidx_j ← H[j].read()   ▷ help ongoing LL operations
11:        if ridx[j] < aidx_j then                 ▷ an LL by p_j may need help
12:            window ← SLR[aidx_j].read()
13:            if j ∉ READERS(window) then to_help ← to_help ∪ {j}
14:    SLR[widx].write(w, j, v, to_help);           ▷ announce new SC
15:    window ← SLR[widx].read(); val ← GETVALUE(widx − 1)
16:    for each j ∈ READERS(window) do ridx[j] ← widx; auditset ← auditset ∪ {(j, val)}
17:    M.writeMax(widx + 1, ridx, auditset);
18:    if (w, j, v, to_help) is the 1st w-tuple in window then return true else return false
19: function AUDIT,GETVALUE, READERS                ▷ as in Algorithm 1
```

corresponds to an *invisible* write operation, whose input v' is never read. We may think as these operations as *unsuccessful*, in the sense that they fail to change the value of the auditable register, being immediately overwritten by another write.

Alternatively, we may think of the sequences in SLR as a trace of an execution β of an LL/SC object implementation by identifying reads with LL and writes with SC. Again, the xth value held by the object in β is v_x, the value in the first w-tuple (w, j, v, h) in win_x. Each process p_i, with $p_i \in h$ or preceding this tuple has an LL that returns v_{x-1}. The first w-tuple (w, j, v, h) indicates a successful SC(v) by process p_j, that changes the object from v_{x-1} to $v_x = v$. And every following tuple $(\mathsf{w}, j', v', \_)$ marks an unsuccessful SC(v') by process $p_{j'}$. Recall that, per its specification, an SC is successful if and only if it is preceded by an LL by the same process, without any successful SC operation between them. In particular, β is a valid sequential execution if (1) in each sequence win_x, the first w-tuple (w, j, v, h) is after p_j's LL and (2) each SC(v') operation corresponding to a following $(\mathsf{w}, j', v', \_)$ tuple is after SC(v) in β.

The code of an LL operation is the same as for a READ operation in Algorithm 1, while AUDIT and the auxiliary functions GETVALUES and READERS are identical. SC operations follow the code of WRITE, with two additions (line 9 and line 18) highlighted in gray.

To maintain property (1), an SC operation *op* by process p_i should be prevented from writing a w-tuple to a sliding register $SLR[x]$ in which p_i 's last LL is not recorded. In Algorithm 1, the local variable lsr of a process p_i is the highest index of a sliding register that keeps track of p_i's last READ, and hence here, $SLR[lsr]$ records p_i's last LL. Therefore, before writing to $SLR[widx]$ (in line 14), p_i checks that $lsr = widx$. If this is not the case, a successful SC has occurred since p_i's last LL, and *op* may immediately return *false* (line 9). The other addition is the return statement at line 18: *true* is returned if p_i's w-tuple is the first in $SLR[widx]$, and *false* otherwise.

For property (2), the linearization rules are slightly modified. In Algorithm 1, WRITE operations recorded in the same sliding register $SLR[x]$ are linearized in the reverse order of their corresponding w-tuple appearance in the sequence stored in $SLR[x]$. Here, we do the

opposite, linearizing SC operations in the same order their corresponding w-tuple appear in $SLR[x]$. Only the first w-tuple represents a successful SC, which is aligned with the return statement of line 18.

Finally, as each process may write twice to a sliding register, once in an LL operation, and once in a SC operation, SLR is an array of $2n$-sliding registers. The code of AUDIT may easily be adapted to report which process has successfully stored which value instead of or in complement to values returned by LL operations to the processes.

The code of AUDIT is the same in Algorithm 1 and Algorithm 3, and the same code is shared by READ and LL operations. For SC, the additional statements (line 9 and line 18) in the code do not affect termination or step-complexity. Therefore, the step complexity of LL, SC, and AUDIT is $O(n)$.

Fix a finite *well-formed* execution β, in which each process alternates LL and SC operations. Thanks to the similarities in the code, a linearization $\mu(\beta)$ of β can be obtained from a linearization $\lambda(\alpha)$ of an execution α of the auditable register implementation induced by β. α is constructed as follows. We introduce a new class for SC operations that terminate immediately after reading $M.widx$ in line 9. Those operations are said to be *silent* (similarly to silent READ operations). We next extract from β an execution α of Algorithm 1 by removing all steps applied by silent SC, and replacing each invocation of SC and LL by and invocation of WRITE and READ, respectively, with the same input. α is a valid execution of Algorithm 1, as besides early termination for SC (line 9), which we dispose of by removing steps of silent SC operations, the code of SC and LL is the same as the code of WRITE and READ, respectively, and the code for AUDIT is identical. We explain in Appendix B how a linearization $\mu(\beta)$ can be inferred from the linearization $\lambda(\alpha)$ of α.

6 Immediate Deny List from Auditable Registers

A *Deny List* [11] is used to control resources, by having a set of managers maintain a list of which users are unauthorized to access which resources. To access a resource, a user must prove that the corresponding user-resource pair has not been added to the Deny List. The managers have to agree on the set of process-resource pairs in the Deny List. A Deny List has an *anti-flickering* property that ensures that there are no transient periods: once access is disallowed, it is never allowed again.

More formally, a *deny list* object over a set of resources S supports three operations, for $x \in S$: APPEND(x), PROVE(x) which returns a Boolean value, and READ(), which returns a set of process-resource (p, x) pairs. A PROVE(x) that returns *false* is *invalid*; otherwise, it is *valid*. The intuition is that an APPEND(x) revokes the authorization to access a resource x to all processes. A valid PROVE(x) by process p indicates that p is authorized to access x. The set of processes that can invoke APPEND is called the *managers*, and those that can invoke PROVE are called the *provers*. These sets of processes are predefined and static. The property *termination* requires that the operations PROVE, APPEND, and READ return within a finite number of steps.

In the original definition [11], an APPEND can be *successful* or *unsuccessful*. Their sequential specification in the *anti-flickering flavor* of a deny list is as follows:

Append progress: Only a finite number of APPEND(x) operations are unsuccessful; that is, APPEND(x) is eventually successful.

Prove progress: After a successful APPEND(x) operation, only a finite number of PROVE(x) operations can be valid; that is, PROVE(x) is eventually invalid after a successful APPEND(x).

Prove validity: A PROVE(x) operation is invalid only if a successful APPEND(x) operation appears before it.

Prove anti-flickering: If a PROVE(x) operation *op* is invalid, then all following PROVE(x) operations are invalid.

Read validity: The set of object-process pairs returned by a READ() operation includes exactly all preceding valid PROVE(x) operations and the processes that invoked them.

We consider a stronger version, called *immediate* deny list, where all APPEND(x) are successful, and where all PROVE(x) operations that follow an APPEND(x) are invalid. The sequential specification for the *immediate* deny list is therefore as follows:

Strong prove validity: A PROVE(x) operation is invalid if and only if an APPEND(x) operation appears before it.

Read validity: The set of object-process pairs returned by a READ() operation includes exactly all preceding valid PROVE(x) operations and the processes that invoked them.

An *Immediate Deny List* guarantees all the properties of the *Anti-flickering Deny List*: Since all APPEND(x) are successful, we trivially guarantee *Append progress*. Our *Strong prove validity* includes the *Prove validity* and *Prove progress* where the number of PROVE(x) that can be valid after an APPEND(x) is 0. *Prove anti-flickering* is trivially implied by our *Strong prove validity* property.

It is already known that the consensus number of an *Anti-flickering Deny List* object where n processes can both do APPEND(x) and PROVE(x), denoted anti-flickering n-deny list, is n [11]. In particular, they show how to solve consensus among n processes using this object with a single resource. Thus, an Anti-flickering n-deny list has at least consensus number n, and therefore, the consensus number of an immediate n-deny list is also at least n.

They also present an algorithm to show that the consensus number of the anti-flickering n-deny list over a set of resources S is at most n. In the following, we show how to implement (Algorithm 4) an immediate-n-deny list object using $(1, n-1)$-auditable registers, which have consensus number n as proved in Sections 3 and 4. This proves that the immediate n-deny list has consensus number at most n. Our algorithm provides a somewhat simpler proof for their upper bound. We present the algorithm for a single resource. The generalized version can be easily built thanks to the locality property of linearizability, and by implementing the READ() on the set of resource $\mathcal{S}$ using classical techniques to implement a snapshot by applying the READ() on each $x \in \mathcal{S}$.

Algorithm 4 implements an immediate n-deny list for a resource x. It uses a vector $AR_x[1 \dots n]$ of binary auditable registers, initially *true*. $AR_x[i]$ is written by process p_i and read by all other processes. When a process p_i wants to perform APPEND(x), it writes false in its auditable register $AR_x[i]$ and locally stores the information that it did an APPEND.

To perform PROVE(x), a process simply checks if it previously did an APPEND(x), and if not, it reads all the auditable registers (but its own) to check if some process wrote into one such register. If none of the previous conditions happen, then it can return *true*.

The READ() repeatedly audits all the registers to collect the processes that have executed a valid PROVE(x). These are the ones that read *true* in all registers. To obtain a snapshot, the READ() terminates when two consecutive collects returns the same set. Appendix C presents proof sketch of the correctness of the algorithm.

7 Conclusions and Future Work

In this work, we extended the concept of auditability from single-writer registers to more general shared objects. We start by providing a rigorous characterization of the synchronization power required to support auditable multi-writer registers. Our results establish a

Algorithm 4 Immediate n deny-list object from auditable registers, code for process p_i.

Shared objects
$AR_x[1 \dots n]$ is a vector of $(1, n-1)$-auditable registers, initially *true*
local variables
$append_x$ is a boolean initially *false*, that is set to *true* when p_i does APPEND(x)
function APPEND(x)
$AR_x[i]$.write(*false*); $append_x \leftarrow true$ **return**
function PROVE(x)
if $append_x$ **then return** *false*;
for all $j \in \{1, \dots, n\}$ with $j \neq i$ **do**
$b \leftarrow AR_x$.read(); **if** $\neg b$ **then return** *false*
return *true*
function READ( ) ▷ audit all registers until you get a successful double collect
$c2 \leftarrow \emptyset$
repeat
$c1 \leftarrow c2$; $c2 \leftarrow \emptyset$
for all $j \in \{1, \dots, n\}$ **do** $a_j \leftarrow AR_x[j]$.audit()
$c2 \leftarrow \{(q, x) : \forall j \neq q, (q, true) \in a_j\}$
until $c1 = c2$
return $c2$

tight bound on the consensus number necessary for achieving auditability, demonstrating the feasibility of implementing auditable storage mechanisms.

Looking ahead, there are several promising directions for future research. First, extending auditability to a broader range of shared objects beyond registers and LL/SC remains an open challenge. Second, investigating the impact of adversarial behavior on auditable implementations could lead to more robust security guarantees. Finally, exploring efficient, scalable implementations of auditable objects in real-world distributed storage systems could bridge the gap between theoretical feasibility and practical deployment. In particular, while sliding registers are not supported in hardware (to the best of our knowledge), they bear resemblance to *shift registers*. This hardware object [16] holds the last w bits "shifted in" the register and has a functionality similar to a sliding register; the consensus number of shift registers is w [2], Although they cannot directly replace sliding registers in our implementations, our algorithmic insights might be leveraged to develop other algorithms that employ shift registers.

References

1 Sara Alhajaili and Arshad Jhumka. Auditability: An approach to ease debugging of reliable distributed systems. In *IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 227–2278, 2019. doi:10.1109/PRDC47002.2019.00053.

2 James Aspnes. The consensus number of a shift register equals its width, 2025. doi:10.48550/arXiv.2505.01691.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1), March 2012. doi:10.1145/2108242.2108244.

4 Hagit Attiya, Antonio Fernández Anta, Alessia Milani, Alexandre Rapetti, and Corentin Travers. Auditing without leaks despite curiosity. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 455–465. ACM, 2025. doi:10.1145/3732772.3733516.

5 Hagit Attiya, Antonio Fernández Anta, Alessia Milani, Alexandre Rapetti, and Corentin Travers. Auditable shared objects: From registers to synchronization primitives, 2025. arXiv:2508.14506.

6 Hagit Attiya, Antonella Del Pozzo, Alessia Milani, Ulysse Pavloff, and Alexandre Rapetti. The synchronization power of auditable registers. In *27th International Conference on Principles of Distributed Systems OPODIS*, volume 286 of *LIPIcs*, pages 4:1–4:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.OPODIS.2023.4.

7 Richard A Becker and John M Chambers. Auditing of data analyses. *SIAM Journal on Scientific and Statistical Computing*, 9(4):747–760, 1988.

8 Vinicius Vielmo Cogo and Alysson Bessani. Brief announcement: Auditable register emulations. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 53:1–53:4. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.53.

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

10 Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 115–124, 2012. doi:10.1145/2332432.2332457.

11 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: the case of allowlist and denylist. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*, volume 281 of *LIPIcs*, pages 21:1–21:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.DISC.2023.21.

12 Boris Glavic et al. Data provenance. *Foundations and Trends® in Databases*, 9(3-4):209–441, 2021. doi:10.1561/1900000068.

13 Reza Hajisheykhi, Mohammad Roohitavaf, and Sandeep S. Kulkarni. Bounded auditable restoration of distributed systems. *IEEE Transactions on Computers*, 66(2):240–255, 2017. doi:10.1109/TC.2016.2595578.

14 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

15 Vincent C Hu, D Richard Kuhn, David F Ferraiolo, and Jeffrey Voas. Attribute-based access control. *Computer*, 48(2):85–88, 2015. doi:10.1109/MC.2015.33.

16 Intel. x86 assembly/shift and rotate. URL: https://en.wikibooks.org/wiki/X86_Assembly/Shift_and_Rotate.

17 Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

18 Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

19 Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. A simple object that spans the whole consensus hierarchy. *Parallel Process. Lett.*, 28(2):1850006:1–1850006:9, 2018. doi:10.1142/S0129626418500068.

20 Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998. doi:10.1016/S0065-2458(08)60206-5.

21 Ravi S. Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994. doi:10.1109/35.312842.

A Additional Lemmas and Proofs for Section 4

We present in this appendix missing proof and additional lemmas for Section 4. Omitted proofs can be found in the full version.

The function READERS extracts from a sequence read from some sliding register $SLR[x]$ a set of processes. This set does not change once a w-tuple has been written to $SLR[x]$.

▶ **Proposition 18.** *If win and win′ are two sequences read from $SLR[x]$ after $SLR[x]$.write(w, _, _, _) has been applied, then READERS(win) = READERS(win′).*

▶ **Lemma 19.** *$M.ridx[i] \geq x$ if and only if $SLR[x]$ holds a sequence win such that $i \in$READERS(win).*

▶ **Lemma 20.** *The step complexity of a READ operation is in $O(m+n)$.*

▶ **Lemma 12.** *If an operation op terminates before an operation op′ starts in H', then $idx(op) \leq idx(op')$.*

Proof. Let $x = idx(op)$ and $x' = idx(op')$, and let p_i and $p_{i'}$ be the processes that perform op and op' respectively.

We first prove that when op terminates, $M.widx \geq x$. As $M.widx$ is increasing (Proposition 5), it is enough to show that $M.widx \geq x$ in a configuration in the execution interval of op. If op is a WRITE or an AUDIT, op writes to $SLR[x]$ (line 35) or reads from $SLR[x]$ (line 41) after reading x from $M.widx$ (line 30 or line 40).

If op is a direct READ, p_i writes i to $SLR[x]$ (line 22) after having read x from $M.widx$ (line 18). If op is helped, $M.ridx[i] = x$ when M is read at the beginning of some iteration of the loop (line 18). Therefore $M.widx > x$, as the writeMax that changes $M.ridx[i]$ to x also set $M.widx$ to $x+1$ (lines 15-16, lines 25-26, or lines 37-38). If op is silent, it is preceded by direct READ op'' with the same index x, and hence $M.widx \geq x$ already when op'' terminates.

We next examine several cases according to the type of op'.

- op' is a silent READ. In op', $SLR[x']$ is read by $p_{i'}$ and does not contain a w-tuple. Hence, before op starts, $WIDX = x' - 1$ (the largest index of a sliding register to which a w-tuple has been written), and therefore, by Lemma 8, $M.widx \leq x' - 1$, from which we have $x \leq x' - 1$, as $x \leq M.widx$ when op terminates.
- op' is a direct or a helped READ. Let x_0 be the value of the local variable lsr when op' starts. op' does not terminate before reading some value x_1 from $M.widx$ (line 18) in the first iteration of the repeat loop. By Lemma 11, for every y, $x_0 < y < x_1$, the helping set of the first w-tuple in $SLR[y]$ does not contains i'. Therefore, $x_1 \leq x' = idx(op')$ and hence $x \leq x'$ as $x \leq M.widx$ before op' starts.
- op' is a WRITE or an AUDIT. $p_{i'}$ reads x' from $M.widx$ (line 30 and line 40, respectively). As $x \leq M.widx$ when op terminates, $x \leq x'$. ◀

To show real-time order is preserved, we first prove the following facts about the precedence of operations with the same sequence number.

▶ **Lemma 21.** *Let op, op′ be two operations in H' with $idx(op) = idx(op') = x$. Let $x \geq 0$.*

1. *If op is a silent READ, a direct READ, or a non-definitive AUDIT, and op′ is a helped READ, a definitive AUDIT, or a WRITE operation, then op′ does not precede op.*
2. *If op is any operation and op′ is a WRITE, then op′ does not precede op.*

▶ **Lemma 22.** *Let op be a helped* READ *operation in H' with $idx(op) = x$. The first write of a w-tuple to $SLR[x]$ happens during the execution interval of op.*

▶ **Lemma 13.** *If an operation op terminates before an operation op' starts in H', then op precedes op' in λ.*

Proof. By Lemma 12, $idx(op) \leq idx(op')$. If $idx(op) < idx(op')$, op is before op' in λ by rule $R0$. We assume in the following that $idx(op) = idx(op') = x$.

- If op' is a silent READ, a direct READ, or non-definitive AUDIT, it follows from Lemma 21(1) that op also falls into this category. Therefore, op and op' are both ordered in λ using rule $R1$. They are ordered according to the order in which a step in their execution interval occurs in α. Hence, op precedes op' in α implies that op precedes op' in λ.
- If op' is a helped READ or a definitive AUDIT, it follows from Lemma 21(1) and Lemma 21(2) that op also falls into this category, or is a silent or direct READ, or a non-definitive AUDIT. In the latter case, op is ordered in λ according to rule $R1$, and op', rule $R2$, from which we have that op precedes op' in λ.
  In the former case, as op and op' are both helped READ, their execution interval intersect (Lemma 22) and thus op cannot precedes op'. If op and op' are both definitive AUDIT, they are placed according to the order in which they apply a read to $SLR[x]$ in α. Hence op is before op' in λ. If op is an helped READ and op' a definitive AUDIT, op is placed before op' in λ by rule $R2$. The last case remaining is op being a definitive AUDIT and op', a helped READ. As op is a definitive AUDIT, it sees a w-tuple mark in $SLR[x]$, but the step in which the first such tuple is written to $SLR[x]$ is in the execution interval of op' (Lemma 22). Therefore op cannot terminate before op' starts.
- If op' is a WRITE visible or hidden, Lemma 21(2) implies that op cannot be a WRITE. Therefore, op is placed according to rule $R1$ or $R2$, and sthus precedes op' which is placed after, according to $R3$ or $R4$. ◀

▶ **Lemma 14.** *If* READ *operation op in H' returns v, then v is the value written by the last* WRITE *that precedes op in λ, or the initial value v_0 if there is no such* WRITE.

Proof. Let $x = idx(op)$, and let p_j be the process that performs op.

We first consider the case $x > 0$. If op is helped or direct, the value returned by op is the value v in the first w-tuple $(\mathsf{w}, i, v, \_)$ stored in $SLR[x-1]$ (lines 45). Note that i and v are well defined, as when p_j reads from or writes to $SLR[x]$, a w-tuple has already be written to $SLR[s-1]$ (Proposition 7). Let wop be the operation p_i is performing when it writes $(\mathsf{w}, i, v, \_)$ to $SLR[x-1]$. By definition, $idx(op) = x - 1$, and wop is a visible WRITE. Therefore, among the operations with index $x - 1$, wop is placed last in λ (rule $R4$), and there is no other WRITE operation between wop and op (every WRITE operation with index $\geq x$ is after op in λ.)

If rop is silent, it is preceded by a direct READ op' by the same process with the same index x. By the linearization rule $R1$, there is no WRITE operation between op' and op in λ. Also, op and op' return the same value. By the same reasoning as above, it follows that op returns the input value of the last WRITE operation that precedes it in λ.

If $x = 0$, op returns the initial value v_0. Indeed, $SLR[-1]$ is initialized with a sequence that contains a single tuple $(\mathsf{w}, i_0, v_0, \emptyset)$. There is no WRITE that precedes op in λ, as for every WRITE operation wop, $idx(wop) \geq 0$. ◀

▶ **Lemma 23.** *If $i \in$ READERS$(SLR[x])$ then there exists a* READ *operation op in H' by process p_i with $idx(op) = x$.*

Proof. Suppose that i precedes any w-tuple in $SLR[x]$. p_i writes i to $SLR[x]$ (line 22) while performing the kth iteration of the repeat loop (lines 17-27) in some READ operation op. Let x_0 be the value of lsr when op starts, and let $x_1 < \ldots < x_k$ be the value read from $M.widx$ (line 18) in the first k iterations of the loop. Note that $x = x_k$. As before writing i to $SLR[x_k = x]$, $H[i]$ is changed to x_k (line 21), it follows from Lemma 10 that for every $x', x_0 < x' < x_k$, $i \notin$ READERS$(SLR[x'])$. As $i \in$ READERS$(SLR[x_k])$, $x_k = \min\{x' > x_0 : i \in$READERS$(SLR[x'])\}$ and therefore by definition $x = x_k = sn(op)$.

Suppose now that i is in the helping set of the first w-tuple $(\mathsf{w}, j, \_, h)$ written to $SLR[x]$. Before writing $(\mathsf{w}, j, \_, h)$ (line 35), p_j reads x from $M.widx$, y from $M.ridx[i]$ (line 30) and x_ℓ from $H[i]$ (line 31). As i is placed into to_help, $y < x_\ell$.

$H[i]$ is changed by p_i (line 21) in an iteration of the repeat loop while performing some READ operation op. Let x_0 be the value of lsr when op starts, and p_i is performing iteration ℓ when it writes x_ℓ to $H[i]$. By the code, x_ℓ is the value read from $M.widx$ in that iteration, and by Lemma 10, $i \notin$ READERS$(SLR[x'])$, for every $x', x_0 < x' < x_\ell$.

To summarize, starting from $i \in$READERS$(SLR[x])$, we have shown that there exists a READ operation op by p_i that starts with $lsr = x_0$, and that for every $x', x_0 < x' < x(= x_\ell)$, $i \notin$READERS$(SLR[x'])$. By definition, $idx(op) = x$. ◀

▶ **Lemma 15.** *Let aop be an* AUDIT *operation in H' that returns A. If there is a* READ *operation rop by process p_j returning v and that precedes aop in λ, $(j, v) \in A$.*

Proof. By linearization rules $R0$-$R2$, $idx(rop) = x \leq idx(aop)$. Let p_i be the process that performs the AUDIT operation aop.

If rop is silent, it is preceded by a direct READ operation by the same process, with the same output value v and the same index x. In the following, we thus assume that rop is direct or helped. In $SLR[x]$, j is written before any w-tuple, or the first w-tuple written to $SLR[x]$ has an helping set h containing i. Note that v is the value of the first w-tuple in $SLR[x-1]$.

Let us assume that $x < idx(aop) = x'$. In aop, p_i reads x' from $M.widx$ and a set as' from $M.auditset$ (line 40). Before this step, M is changed from x to $x+1$ (in line 16, line 26 or line 38), after a win containing a w-tuple is read from $SLR[x]$. Hence, $j \in$ READERS(win), and therefore (j, v) is added to the audit set as written together with $x+1$ to M. By the code (line 15, line 25 or line 37 and line 42), we have that $as \subseteq as' \subseteq A$.

We now assume that $x = idx(aop)$. If aop is a non-definitive AUDIT, no w-tuple has been written to $SLR[x]$ when p_i reads $SLR[x]$. As rop precedes aop in λ, there are both placed in λ following rule $R1$ and therefore p_j writes j to $SLR[x]$ before the sliding register is read in aop, from which it follows that $(j, v) \in A$ (line 42). Otherwise, aop is a definitive AUDIT, which means that the value win reads from $SLR[x]$ (line 42) contains the first w-tuple $(\mathsf{w}, \_, \_, h)$ written to $SLR[x]$. As j precedes this tuple or $i \in h$, $j \in$ READERS(win) and therefore $(j, v) \in A$. ◀

▶ **Lemma 16.** *Let aop be an* AUDIT *operation in H' whose response contains (j, v). There exists a* READ *operation by p_j returning v that precedes aop in λ.*

Proof. Let p_i be the process that performs aop, and let A be the set of pairs (process,value) returned by this operation. As (j, v) is in A, there exists x such that $i \in$READERS$(SLR[x])$ and v is the value in the first w-tuple $(\mathsf{w}, \_, v\_)$ written to $SLR[x-1]$. Indeed, pair (j, v) is inserted to A by p_i after reading $SLR[x]$, if $x = idx(aop)$ (line 42), or when $M.widx$ is changed to $x+1$ (lines 15-16, lines 25-26 or lines 37-38) if $x < idx(aop)$.

By Lemma 23 shows there is a READ operation op by process p_i with $idx(op) = x$. This READ returns v, which is the value of the first w-tuple written to $SLR[x-1]$. It remains to prove that op precedes aop in λ. If $x < idx(aop)$, op precedes aop in λ (rule $R0$). Otherwise, $x = idx(aop)$. If aop is definitive, it is linearized after rop by rules $R1$ and $R2$. Else, aop is non-definitive, and j is before any w-tuple in $SLR[x]$. j is also written to $SLR[x]$ before before $SLR[x]$ is read by p_i. op is hence direct, and linearized before aop by rule $R1$. ◀

B Proof Sketch for Section 5

Recall that β is an execution of the auditable LL/SC implementation (Algorithm 3) and α an execution of the auditable register implementation induced from β. We build a linearization $\mu(\beta)$ of β from the linearization $\lambda(\alpha)$ of α as follows:

1. WRITE operations with the same index x are reordered, by applying first rule $R4$ and then rule $R3$. Hence, the WRITE operation corresponding to the first w-tuple in $SLR[x]$ is placed first, and then the WRITEs corresponding to the other w-tuples follow, in arbitrary order. The resulting sequence $\lambda'(\alpha)$ is no longer a valid linearization of an auditable register, but still extends the real-time order among operations, as WRITEs with the same index are concurrent (Lemma 21(2)). Observe that in $\lambda'(\alpha)$, as in $\lambda(\alpha)$, WRITEs with the same index form a contiguous block, denoted W_x. A READ operation no longer returns the input of the last preceding WRITE, but rather the input v_x of the first WRITE in the last block W_x that precedes it.
2. We revert to LL/SC operations, by replacing in $\lambda'(\alpha)$ each READ and WRITE operation by the LL or SC operation it originated from. This results in a partial linearization $\mu'(\beta)$ of β, missing silent SC operations. In $\mu'(\beta)$, an LL operation op returns the input v_x of the first SC operation sop in the last block W_x that precedes it. Every SC operation in W_x writes a w-tuple to $SLR[x]$, sop being the first to do so. Hence, except sop, each of them returns false (line 18). Consequently, op returns the input of the last preceding successful SC.
   If p_i has a SC operation in W_x, its matching preceding LL operation op is recorded in $SLR[x]$ (line 9) and therefore, has index $idx(op) = x$. By rules $R1$-$R2$, op is before W_x, and after any operation op' with index $idx(op') < x$ (rule $R0$). Only the first SC operation sop in W_x returns true, and indeed, there is no SC between sop and its last preceding LL. Each other SC operation in W_x is unsuccessful, in agreement with the fact that it preceded by the successful SC sop, with no LL operation in between.
   LL and AUDIT are ordered in $\mu'(\beta)$ as their corresponding READ and AUDIT in $\lambda(\alpha)$, and each LL returns the same value as its corresponding READ. Hence, a pair (p, v) is included in the response of an AUDIT operation op if and only if there is an LL by p that returns v before op in $\mu'(\beta)$.
3. Finally, we bring back the silent SC operations that were removed from β to form a full linearization $\mu(\beta)$. Each silent SC operation op that returns is unsuccessful and therefore can be inserted in $\mu'(\beta)$ without affecting the outcome of other SC or LL operations, as long as there is a successful SC between the matching LL op' by the same process and op. Let x be the value read by op from $M.widx$. We insert op in $\mu'(\beta)$ within the operations whose index is x and after every operation that terminates before op starts. As $x > x'$, op is placed after the operations with index x'. In particular, op follows the successful SC operation sop that writes the first w-tuple to $SLR[x]$, which in turn follows op'. This can be done while preserving the real-time order, as when $x > x'$ is read from $M.widx$ in op, a w-tuple has already been written to $SRL[x']$ (Lemma 8).

C Proof Sketch for Section 6

To prove the correctness of Algorithm 4, we start by noting that all operations terminate within a finite number of their own steps. This is obvious for APPEND and PROVE, relying on wait-freedom of the encapsulated auditable register operations. For a READ operation, note that the sets it returns are monotonically contained in each other; the maximal set is the one containing all process-object pairs. Furthermore, if two sets are equal, the operation terminates. Thus, the loop can be repeated at most n times.

We next prove that Algorithm 4 implements a linearizable immediate n-deny list. First, for all $i \in \{1, \ldots, n\}$, we linearize all the APPEND(x) in the step corresponding to the write to the auditable register. APPEND(x) that have not reached this step are discarded.

A PROVE(x) operation by p_i that returns *false* and that does not return at line 6 is linearized at the last read primitive p_i applies to an auditable register $AR_x[j]$. A PROVE(x) that returns at line 6 is linearized at its invocation. A READ is linearized at the last audit of the second last loop (i.e., the last audit before c_1 is set for the last time). We remove incomplete READ.

PROVE(x) operations that return *true* are inserted at the beginning before the first APPEND(x), according to their real-time order. In particular, we pick one by one in the order of their last step (i.e., the last read from an auditable register), the ones that happen earlier first), denoted s. We put PROVE(x) immediately before the first READ() whose linearization point follows s or immediately the first APPEND(x) if such READ() does not exist.

Strong prove validity: A PROVE(x) by a process p returns *false*, either if p previously performed an APPEND(x) or if it read *false* in one auditable register. In both cases, there is an APPEND(x) in the set of operations we linearize and it is linearized before the PROVE(x) according to our rules. The only if part is immediate since we linearize all PROVE(x) that return *true* before the first APPEND(x).

▶ **Lemma 24.** *The set of object-process pairs returned by a* READ() *operation includes exactly all preceding valid* PROVE(x) *operations and the processes that invoked them.*

Proof. A $op =$ PROVE(x) that returns *true* is linearized before the first audit, namely op', whose linearization point follows the last read primitive applied by op to an auditable register. After the step where it is linearized op' executes a second loop where it audits all the auditable registers. At that point op is added in c_2. By the definition of the linearization point of op', op is in the set returned by op'.

On the other hand, suppose that a pair $(q, true)$ is in the set c_2 returned by a READ() by p. Then, there is a PROVE(x) by q that read *true* in all the auditable registers (but $AR_x[q]$) before p start the execution of the last loop. By a simple inspection of the pseudo code it is easy to see that op returns *true*. Our linearization rule completes the proof. ◀

Finally, we show that our linearization respects the *real-time order* of operations. All operations but the PROVE(x) that return *true* are linearized in a point of their execution. A PROVE(x) operations that returns *true* is linearized before the first APPEND(x) in the linearization. As it reads *true* in all low-level auditable registers, there is no APPEND(x) that completes before this PROVE(x) is invoked, otherwise one of the low-level read would have returned *false*. If PROVE(x) op completes before a READ() op' starts, then the last step of op precedes the linearization point of op'. Thus, op is linearized before op'. Similarly if op is invoked after op' completes, then the linearization point of op' precedes the last step of op which is then linearized after op'. This concludes the proof since PROVE(x) are linearized in the order of their last step.