Auditing without Leaks Despite Curiosity

Hagit Attiya
Technion
Haifa, Israel
hagit@cs.technion.ac.il

Antonio Fernández Anta
IMDEA Software & Networks Inst.
Madrid, Spain
antonio.fernandez@imdea.org

Alessia Milani
Aix Marseille Univ, CNRS, LIS
Marseille, France
alessia.milani@lis-lab.fr

Alexandre Rapetti
Université Paris-Saclay, CEA, List
Palaiseau, France
alexandre.rapetti@cea.fr

Corentin Travers
Aix Marseille Univ, CNRS, LIS
Marseille, France
corentin.travers@lis-lab.fr

Abstract

Auditing data accesses helps preserve privacy and ensures accountability by allowing one to determine who accessed (potentially sensitive) information. A prior formal definition of register auditability was based on the values returned by read operations, *without accounting for cases where a reader might learn a value without explicitly reading it or gain knowledge of data access without being an auditor.*

This paper introduces a refined definition of auditability that focuses on when a read operation is *effective*, rather than relying on its completion and return of a value. Furthermore, we formally specify the constraints that *prevent readers from learning values they did not explicitly read or from auditing other readers' accesses.*

Our primary algorithmic contribution is a wait-free implementation of a *multi-writer, multi-reader register* that tracks effective reads while preventing unauthorized audits. The key challenge is ensuring that a read is auditable as soon as it becomes effective, which we achieve by combining value access and access logging into a single atomic operation. Another challenge is recording accesses without exposing them to readers, which we address using a simple encryption technique (one-time pad).

We extend this implementation to an *auditable max register* that tracks the largest value ever written. The implementation deals with the additional challenge posed by the max register semantics, which allows readers to learn prior values without reading them.

The max register, in turn, serves as the foundation for implementing an *auditable snapshot* object and, more generally, *versioned types*. These extensions maintain the strengthened notion of auditability, appropriately adapted from multi-writer, multi-reader registers.

CCS Concepts

• **Theory of computation → Distributed algorithms**.

PODC '25, Huatulco, Mexico
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1885-4/25/06
https://doi.org/10.1145/3732772.3733516

Keywords

Auditability, Wait-free implementation, Synchronization power, Distributed objects, Shared memory

ACM Reference Format:
Hagit Attiya, Antonio Fernández Anta, Alessia Milani, Alexandre Rapetti, and Corentin Travers. 2025. Auditing without Leaks Despite Curiosity. In *ACM Symposium on Principles of Distributed Computing (PODC '25), June 16–20, 2025, Huatulco, Mexico.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3732772.3733516

1 Introduction

Auditing is a powerful tool for determining *who* had access to *which* (potentially sensitive) information. Auditability is crucial for preserving data privacy, as it ensures accountability for data access. This is particularly important in shared, remotely accessed storage systems, where understanding the extent of a data breach can help mitigate its impact.

1.1 Auditable Read/Write Registers

Auditability was introduced by Cogo and Bessani [9] in the context of replicated *read/write registers.* An auditable register extends traditional read and write operations with an additional *audit* operation that reports which register values have been read and by whom. The auditability definition by Cogo and Bessani 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 [6], based on collectively linearizing read, write, and audit operations. This work also analyzes the consensus number required for implementing auditable single-writer registers, showing that it scales with the number of readers and auditors. However, this definition assumes that a reader only gains access to values that are explicitly *returned* by its read operations. This assumption does not account for situations where a reader learns the register's value before it has officially returned, making the read operation *effective*. Hence, a notable limitation of this definition is that a process with an effective read can refuse to complete the operation, thereby avoiding detection by the audit mechanism.

Prior work has also overlooked the risk of non-auditors learning values without explicitly reading them or inferring accesses of other processes. Even when processes follow their prescribed algorithms without active misbehavior, existing auditable register implementations allow an "honest but curious" process to learn more than what its read operations officially return. Additionally, extending auditability beyond read/write registers remained an unexplored territory.

1.2 Our Contributions and Techniques

In this work, we propose a stronger form of auditability for read-/write registers, ensuring that all effective reads are auditable and that non-auditors cannot infer the values read by other processes. We further extend these properties to other data structures and propose new algorithms that fulfill these guarantees.

We define new properties that ensure operations do not leak information when processes are honest-but-curious [14]. Firstly, we introduce an implementation-agnostic definition of an *effective operation*, which is applicable, for instance, to read operations in an auditable register. An operation is effective if a process has determined its return value in all executions indistinguishable to it. Secondly, we define *uncompromised operations*, saying, for example, that in a register, readers do not learn which values were read by other readers or gain information about values they do not read. This definition is extended beyond registers. For arbitrary data objects, we specify that an operation is *uncompromised* if there is an indistinguishable execution where the operation does not occur.

Enforcing uncompromised operations in auditable objects poses a challenge since it is, in a sense, antithetical to securely logging data accesses. Our primary algorithmic contribution (Section 3) is a wait-free, linearizable implementation of an auditable multi-writer, multi-reader register. Our implementation ensures that all effective reads are auditable while preventing information leaks: reads are uncompromised by other readers, and cannot learn previous values — unless they actually read them. As a consequence, the implementation is immune to a honest-but-curious attacker.

To achieve these properties, our algorithm carefully combines value access with access logging. Additionally, access logs are encrypted using one-time pads known only to writers and auditors. The subtle synchronization required in our implementation is achieved by using compare&swap and fetch&xor (in addition to ordinary reads and writes). Such strong synchronization primitives are necessary since even simple single-writer auditable registers can solve consensus [6]. The correctness proof of the algorithm, of basic linearizability properties as well as of advanced auditability properties, is intricate and relies on a careful linearization function.

Our second algorithmic contribution is an elegant extension of the register implementation to other commonly-used objects. We first extend our framework to a wait-free, linearizable implementation of an auditable multi-writer, multi-reader *max register* [2], which returns the largest value ever written. The semantics of a max register, together with tracking the number of operations applied to it (needed for logging accesses), may leak information to the reader about values it has not effectively read. We avoid this leakage by adding a *random nonce*, serving to introduce some noisiness, to the values written. (See Section 4.) As before, all effective reads are auditable, and no additional information is leaked.

In Section 5, we demonstrate how an auditable max register enables auditability in other data structures. Specifically, we implement auditable extension of *atomic snapshots* [1] and more generally, of *versioned types* [12]. Many useful objects, such as counters and logical clocks, are naturally versioned or can be made so with minimal modification.

Additional proofs and details can be found in the full version [3].

1.3 Related Work

Cogo and Bessani [9] present an algorithm to implement an auditable *regular* register, using $n \geq 4f + 1$ atomic read/write shared objects, f of which may fail by crashing. 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. 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 [11] study the possibility of implementing an atomic auditable register, 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 $3f + 1$.

Attiya, Del Pozzo, Milani, Pavloff and Rapetti [6] provides the first implementation-agnostic auditability definition. Using this definition they show that auditing adds power to reading and writing, as it allows processes 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).

When faulty processes are *malicious*, *accountability* [7, 8, 15, 19] 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 lets the auditor derive conclusions about the processes' behavior.

In addition to tracking access to shared data, it might be desirable to give to some designated processes the ability to grant and/or revoke access rights to the data. Frey, Gestin and Raynal [13] specify and investigate the synchronization power of shared objects called *AllowList* and *DenyList*, allowing a set of manager processes to grant or revoke access rights for a given set of resources.

2 Definitions

Basic notions. 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 reads and writes, our implementations use two additional standard synchronization primitives: compare&swap(R, old, new) atomically compares the current value of R with old and if they are equal, replaces the current value of R with new; fetch&xor(R, arg) atomically replaces the current value of R with a bitwise XOR of the current value and arg.[1]

[1] fetch&xor is part of the ISO C++ standard since C++11 [10].

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 the shared memory. The process may change its local state after a step, and it may return a *response* to the operation of the high-level object.

Implemented (high-level) operations are denoted with capital letters, e.g., READ, WRITE, AUDIT, while primitives applied to base objects, appear in normal font, e.g., read and write.

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. For an execution α and a process p, $\alpha|_p$ is the projection of α on events by p. For two executions α and β, we write $\alpha \overset{p}{\sim} \beta$ when $\alpha|_p = \beta|_p$, and say that α and β are *indistinguishable* to process p.

An operation *op completes* in an execution α if α includes both the invocation and response of *op*; if α includes the invocation of *op*, but no matching response, then *op* is *pending*. An operation *op precedes* another operation *op′* in α if the response of *op* appears before the invocation of *op′* in α.

A *history H* is a sequence of invocation and response events; no two events occur at the same time. The notions of *complete*, *pending* and *preceding* operations extend naturally to histories.

The standard correctness condition for concurrent implementations is *linearizability* [16]: 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 L (a* linearization *of the operations on T in α) such that:*

- *L 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 L, and*
- *L respects the sequential specification of the high-level object.*

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

An implementation is *lock-free* if, whenever there is a pending operation, some operation returns in a finite number of steps of all processes. Finally, 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.

Auditable objects. 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. Formally, 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.

Formally, the sequential specification of an auditable register enforces, in addition to the requirement on 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. In prior work [6], this *if and only if* property was stated as a combination of two properties of the sequential execution: *accuracy*, if a READ is in the response set of the AUDIT, then the READ is before the AUDIT (the *only if* part), and *completeness*, any READ before the AUDIT is in its response set (the *if* part).

We wish to capture in a precise, implementation-agnostic manner, the notion of an *effective operation*, which we will use to ensure that an AUDIT operation will report all *effective* operations. Assume an algorithm $\mathcal{A}$ that implements an object T. The next definition characterizes, in an execution in which a process p invokes an operation, a point at which p knows the value that the operation returns, even if the response event is not present.

Definition 2 (effective operation)**.** *An operation op on object T by process p is v-effective after a finite execution prefix α if, for every execution prefix β indistinguishable from α to p (i.e., such that $\alpha \overset{p}{\sim} \beta$), op returns v in every extension β' of β in which op completes.*

Observe that in this definition, α itself is also trivially an execution prefix indistinguishable to p, and hence in any extension α' in which *op* completes returns value v. Observe as well that *op* could already be completed in α or not be invoked (yet). However, the most interesting case is when *op* is pending in α.

We next define the property that an operation on T is not compromised in an execution prefix by a process. As we will see, in our register algorithm, a READ by p is linearized as soon as it becomes v-effective, in a such way that in any extension including a complete AUDIT, p is reported as a reader of v by this AUDIT. This, however, does not prevent a curious reader p from learning another value v' for which none of its READ operations is v'-effective. In such a situation, the WRITE operation with input v' is said to be *compromised* by p. The next definition states that this can happen only if a READ operation by p becomes v'-effective. The definition is general, and applies to any object.

Definition 3 (uncompromised operation)**.** *Consider a finite execution prefix α and an operation op by process q whose invocation is in α. We say that op is* uncompromised in α by process p *if there is another finite execution β such that $\alpha \overset{p}{\sim} \beta$ and op is not invoked in β.*

A value v is *uncompromised by a reader p* if all WRITE(v) operations are uncompromised by p, unless p has an effective READ returning v.

One-time pads. To avoid data leakage, we employ *one-time pads* [18, 20]. Essentially, a one-time pad is a random string—known only to the writers and auditors—with a bit for each reader. To encrypt a message m, m is bitwise XORed with the pad obtaining a ciphertext c. Our algorithm relies on an infinite sequence of one-time pads. A one-time pad is *additively malleable*, i.e., when f is an additive function, it is possible to obtain a valid encryption of $f(m)$ by applying a corresponding function f' to the ciphertext c corresponding to m.

Attacks. We consider an honest-but-curious (also called *semi-honest and passive*) [14] attacker that interacts with the implementation of T by performing operations, and adheres to its code. It may however stop prematurely and perform arbitrary local computations on the responses obtained from base objects. For instance, for an auditable register, the attacker can attempt to infer in a READ operation the current or a past value of the register, without being reported in AUDIT operations.

3 An Auditable Multi-writer, Multi-reader Register

We present a wait-free and linearizable implementation of a multi-writer, multi-reader register (Alg. 1), in which effective reads are auditable. Furthermore, the implementation does not compromise other reads, as while performing a read operation, a process is neither able to learn previous values, nor whether some other process has read the current value. We ensure that a read operation is linearized as soon as, and not before it becomes effective. Audits hence report exactly those reads that have made enough progress to infer the current value of the register. As a consequence, the implementation is immune to an honest-but-curious attacker.

3.1 Description of the Algorithm

The basic idea of the implementation is to store in a single register R, the current value and a sequence number, as well as the set of its readers, encoded as a bitset. Past values, as well as their reader set, are stored in other registers (arrays V and B in the code, indexed by sequence numbers), so auditors can retrieve them. Changing the current value from v to w consists in first copying v and its reader set to the appropriate registers $V[s]$ and $B[s]$, respectively (where s is v's sequence number), before updating R to a triple formed by w, a new sequence number, and an empty reader set. This is done with a compare&swap in order not to miss changes to the reader set occurring between the copy and the update. An auditor starts by reading R, obtaining the current value w, its set of readers, and its sequence number s. Then it goes over arrays B and V to retrieve previous values written and the processes that have read them.

In an initial design of the implementation, a READ operation obtains from R the current value v and the reader set, adding locally the ID of the reader to this set before writing it back to R, using compare&swap. This simple design is easy to linearize (each operation is linearized with a compare&swap or a read applied to R). However, besides the fact that READ and WRITE are only lock-free, this design has two drawbacks regarding information leaking:

First, a reader can read the current value without being reported by AUDIT operations, simply by not writing to the memory after reading R, when it already knows the current value v of the register. This step does not modify the state of R (nor of any other shared variables), and it thus cannot be detected by any other operation. Therefore, by following its code, but pretending to stop immediately after accessing R, a reader is able to know the current value without ever being reported by AUDIT operations.

Second, each time R is read by some process p, it learns which readers have already read the current value. Namely, while performing a READ operation, a process can compromise other reads.

Alg. 1 presents the proposed implementation of an auditable register. We deflect the "crash-simulating" attack by having each READ operation apply at most one primitive to R that atomically returns the content of R and updates the reader set. To avoid partial auditing, the reader set is encrypted, while still permitting insertion by modifying the encrypted set (i.e., a light form of homomorphic encryption.). Inserting the reader ID into the encrypted set should be kept simple, as it is part of an atomic modification of R. We apply to the reader set a simple cipher (the one-time pad [18, 20]), and benefit from its additive malleability. Specifically, the IDs of the readers of the current value are tracked by the last m bits of R, where m is the number of readers. When a new value with sequence number s is written in R, these bits are set to a random m-bit string, $rand_s$, only known by writers and auditors. This corresponds to encrypting the empty set with a random mask. Process p_i is inserted in the set by XORing the ith tracking bit with 1. Therefore, retrieving the value stored in R and updating the reader set can be done atomically by applying fetch&xor. Determining set-membership requires the mask $rand_s$, known only to auditors and writers.

The one-time pad, as its name indicates, is secure as long as each mask is used at most once. This means we need to make sure that different sets encrypted with the same mask $rand_s$ are never observed by a particular reader, otherwise, the reader may infer some set member by XORing the two ciphered sets. To ensure that, we introduce an additional register SN, which stores only the sequence number of the current value. A READ operation by process p_i starts by reading SN, and, if it has not changed since the previous READ by the same process, immediately returns the latest value read. Otherwise, p_i obtains the current value v and records itself as one of its readers by applying a fetch&xor(2^i) operation to R. This changes the ith tracking bit, leaving the rest of R intact. Finally, p_i updates SN to the current sequence number read from R, thus ensuring that p_i will not read R again, unless its sequence number field is changed. This is done with a compare&swap to avoid writing an old sequence number in SN.

Writing a new value w requires retrieving and storing the IDs of the readers of the current value v for future AUDIT, writing w, the new sequence number $s+1$, and an empty reader set encrypted with a fresh mask $rand_{s+1}$ to R before announcing the new sequence number in SN. To that end, p_j first locally gets a new sequence number $s + 1$, where s is read from SN. It then repeatedly reads R, deciphers the tracking bits and updates shared registers $V[s]$ and $B[s]$ accordingly until it succeeds in changing it to $(s+1, w, rand_{s+1})$ or it discovers a sequence number $s' \geq s+1$ in R. In the latter case, a concurrent WRITE(w') has succeeded, and may be seen as occurring immediately after p_j's operation, which therefore can be abandoned. In the absence of a concurrent WRITE, the compare&swap applied to R may fail as the tracking bits are modified by a concurrent READ. This happens at most m times, as each reader applies at most one fetch&xor to R while its sequence number field does not change. Whether or not p_j succeeds in modifying R, we make sure that before WRITE(w) terminates, the sequence number SN is at least as large as the new sequence number $s + 1$. In this way, after that, WRITE operations overwrite the new value w and READ operations return w or a more recent value.

Because SN and R are not updated atomically, their sequence number fields may differ. In fact, an execution of Alg. 1 alternates between *normal E* phases, in which both sequence numbers are *equal*, and *transition D* phases in which they *differ*. A transition phase is triggered by a WRITE(w) with sequence number s and ends when the WRITE completes or it is helped to complete by updating SN to s. Care must be taken during a D phase, as some READ, which is *silent*, may return the old value v, while another, *direct*, READ returns the value w being written. For linearization, we push back silent READ before the compare&swap applied to R that marks the beginning of phase D, while a direct READ is linearized with its fetch&xor applied to R.

An AUDIT starts by reading R, thus obtaining the current value v, and its sequence number s; it is linearized with this step. It then returns the set of readers for v (inferred from the tracking bits read from R) as well as for each previously written value (which can be found in the registers $V[s']$ and $B[s']$, for $s' < s$.). In a D phase, a silent READ operation may start after an AUDIT reads R while being linearized before this step, so we make sure that the D phase ends before the AUDIT returns. This is done, as in direct READ and WRITE, by making sure that SN is at least as large as the sequence number s read from R. In this way, a silent READ (this also holds for a WRITE that is immediately overwritten) whose linearization point is pushed back before that of an AUDIT is concurrent with this AUDIT, ensuring that the linearization order respects the real time order between these operations.

Suppose that an AUDIT by some process p_i reports p_j as a reader of some value v. This happens because p_i directly identifies p_j as a reader of v from the tracking bits in R, or indirectly by reading the registers $V[s]$ and $B[s]$, where $V[s] = v$. In both cases, in a READ instance op, reader p_j has previously applied a fetch&xor to R while its value field is v. Since the response of this fetch&xor operation completely determines the return value of op, independently of future or past steps taken by p_j, op is effective. Therefore, only effective operations are reported by AUDIT, and if an AUDIT that starts after op is effective, it will discover that p_j read v, again either directly in the tracking bits of R, or indirectly after the reader set has been copied to $B[s]$.

3.2 Proof of Correctness

Partitioning into phases. We denote by $R.seq$, $R.val$ and $R.bits$ the sequence number, value and m-bits string, respectively, stored in R. We start by observing that the pair of values in $(R.seq, SN)$ takes on the following sequence: $(0, 0), (1, 0), (1, 1), \ldots, (x, x-1), (x, x), \ldots$ Indeed, when the state of the implemented register changes to a new value v, this value is written to R together with a sequence number $x+1$, where x is the current value of SN. SN is then updated to $x + 1$, and so on.

Initially, $(R.seq, SN) = (0, 0)$. By invariants that can be proved on the algorithm, the successive values of $R.seq$ and SN are $0, 1, 2, \ldots$, $SN \geq x - 1$ when $R.seq$ is changed to x, and when SN is changed to x, $R.seq$ has previously been updated to x. Therefore, the sequence of successive values of the pair $(R.seq, SN)$ is $(0, 0), (1, 0), (1, 1), \ldots$, $(x, x-1), (x, x), \ldots$. We can therefore partition any execution into intervals E_x and D_x (for *E*qual and *D*ifferent), so that $R.seq = x$ and $SN = x$ during E_x, and $R.seq = x$ and $SN = x - 1$ during D_x:

Lemma 1. *A finite execution α can be written, for an integer $k \geq 0$, either as $E_0\rho_1D_1\sigma_1E_1 \ldots \rho_kD_k\sigma_kE_k$ or as $E_0\rho_1D_1\sigma_1E_1 \ldots \sigma_{k-1}\,E_{k-1}\rho_kD_k$, where:*

- *ρ_ℓ and σ_ℓ are the steps that respectively change the value of $R.seq$ and SN from $\ell-1$ to ℓ (ρ_ℓ is a successful R.compare&swap, line 14, σ_ℓ is also a successful SN.compare&swap, applied within a READ, line 5, a WRITE, line 15, or an AUDIT, line 22).*
- *in any configuration in E_ℓ, $R.seq = SN = \ell$, and in any configuration in D_ℓ, $R.seq = \ell = SN + 1$.*

Algorithm 1 Multi-writer, m-reader auditable register implementation

```
shared registers:
    $R$: a register supporting read, compare&swap, and
        fetch&xor, initially $(0, v_0, rand_0)$
            ▷ store a triple (sequence number, value, $m$-bits string)
    $SN$: a register supporting read and compare&swap,
        initially 0
    $V[0..+\infty]$ registers, initially $[\bot, \ldots, \bot]$
    $B[0..+\infty][0..m-1]$ Boolean registers, initially,
        $B[s, j] = false$ for every $(s, j) : s \geq 0, 0 \leq j < m$.
local variables: reader
    $prev\_val, prev\_sn$: latest value read ($\bot$ initially)
        and its sequence number ($-1$ initially)
local variables common to writers and auditors
    $rand_0, rand_1, \ldots$: sequence of random $m$-bit strings
local variables: auditor
    $A$: audit set, initially $\emptyset$;
    $lsa$: latest “audited” seq. number, initially 0
1:  function READ()                  ▷ code for reader $p_j$, $0 \leq j < m$
2:      $sn \leftarrow SN$.read()
3:      if $sn = prev\_sn$ then return $prev\_val$
                    ▷ no new write since latest READ operation
4:      $(sn, val, \_) \leftarrow R$.fetch&xor$(2^j)$
                ▷ fetch current value and insert $j$ in reader set
5:      $SN$.compare&swap$(sn - 1, sn)$ ▷ help complete $sn$th WRITE
6:      $prev\_sn \leftarrow sn$; $prev\_val \leftarrow val$; return $val$
7:  function WRITE($v$)        ▷ code for writer $p_i$, $i \notin \{0, \ldots, m-1\}$
8:      $sn \leftarrow SN$.read() + 1
9:      repeat
10:         $(lsn, lval, bits) \leftarrow R$.read()
11:         if $lsn \geq sn$ then break
12:         $V[lsn]$.write$(lval)$;
13:         for each $j : bits[j] \neq rand_{lsn}[j]$ do
        $B[lsn][j]$.write$(true)$
14:     until $R$.compare&swap$((lsn, lval, bits), (sn, v, rand_{sn}))$
15:     $SN$.compare&swap$(sn - 1, sn)$; return
16: function AUDIT()
17:     $(rsn, rval, rbits) \leftarrow R$.read()
18:     for $s = lsa, lsa + 1, \ldots, rsn - 1$ do
19:         $val \leftarrow V[s]$.read();
20:         $A \leftarrow A \cup \{(j, val) : 0 \leq j < m, B[s][j]$.read() $= true\}$
21:     $A \leftarrow A \cup \{(j, rval) : 0 \leq j < m, bits[j] \neq rand_{rsn}[j]\}$
22:     $lsa \leftarrow rsn$; $SN$.compare&swap$(rsn - 1, rsn)$; return $A$
```

Termination. It is clear that AUDIT and READ operations are wait-free. We prove that WRITE operations are also wait-free, by showing that the repeat loop (lines 9-14) terminates after at most $m + 1$ iterations. This holds since each reader may change R at most once (by applying a R.fetch&xor, line 4) while $R.seq$ remains the same.

Lemma 2. *Every operation terminates within a finite number of its own steps.*

PROOF SKETCH. The lemma clearly holds for READ and AUDIT operations. Let wop be a WRITE operation, and assume, towards a contradiction, that it does not terminate. Let $sn = x + 1$ be the sequence number obtained at the beginning of wop at line 8, where x is the value read from SN. We denote by (sr, vr, br) the triple read from R in the first iteration of the repeat loop. It can be shown that $x \leq sr$. As $sr < sn = x + 1$ (otherwise the loop breaks in the first iteration at line 11, and the operation terminates), we have $sr = x$.

As wop does not terminate, in particular the compare&swap applied to R at the end of the first iteration fails. Let (sr', vr', br') be the value of R immediately before this step is applied. This can be used to show that if $sr' \neq sr$ or $vr' \neq vr$, then $sr' > sr$. Therefore, wop terminates in the next iteration as the sequence number read from R in that iteration is greater than or equal to sn (line 11). It thus follows that $sr = sr'$, $vr = vr'$, and $br \neq br'$: at least one reader applies a fetch&xor to R during the first iteration of repeat loop.

The same reasoning applies to the next iterations of the repeat loop. In each of them, the sequence number and the value stored in R are the same, sr and vr respectively (otherwise the loop would break at line 11), and thus a reader applies a fetch&xor to R before the compare&swap of line 14 (otherwise the compare&swap succeeds and wop terminates). But it can be shown that each reader applies at most one fetch&xor to R while it holds the same sequence number, which is a contradiction. □

Linearizability. Let α be a finite execution, and H be the history of the READ, WRITE, and AUDIT operations in α. We classify and associate a sequence number with some of READ and WRITE operations in H as explained next. Some operations that did not terminate are not classified, and they will later be discarded.

- A READ operation op is *silent* if it reads $x = prev\_sn$ at line 2. The sequence number $sn(op)$ associated with a *silent* READ operation op is the value x returned by the read from SN. Otherwise, if op applies a fetch&xor to R, it is said to be *direct*. Its sequence number $sn(op)$ is the one fetched from R (line 4).
- A WRITE operation op is *visible* if it applies a successful compare&swap to R (line 14). Otherwise, if op terminates without applying a successful compare&swap on R (by exiting the repeat loop from the break statement, line 11), it is said to be *silent*. For both cases, the sequence number $sn(op)$ associated with op is $x + 1$, where x is the value read from SN at the beginning of op (line 8).

Note that all terminated READ or WRITE operations are classified as silent, direct, or visible. An AUDIT operation op is associated with the sequence number read from R at line 17.

We define a complete history H' by removing or completing the operations that do not terminate in α, as follows: Among the operations that do not terminate, we remove every AUDIT and every unclassified READ or WRITE. For a silent READ that does not terminate in α, we add a response immediately after SN is read at line 2. The value returned is $prev\_val$, that is the value returned by the previous READ by the same process. For each direct READ operation op that does not terminate in α, we add a response with value v defined as follows. Since op is direct, it applies a fetch&xor on R that returns a triple (sr, vr, br); v is the value vr in that triple. In H', we place the response of non-terminating direct READ and visible WRITE after every response and every remaining invocation of H, in an arbitrary order.

Finally, to simplify the proof, we add at the beginning of H' an invocation immediately followed by a response of a WRITE operation with input v_0 (the initial value of the auditable register.). This fictitious operation has sequence number 0 and is visible.

Essentially, in the implemented register updating to a new value v is done in two phases. R is first modified to store v and a fresh sequence number $x + 1$, and then the new sequence number is announced in SN. Visible WRITE, direct READ, and AUDIT operations may be linearized with respect to the compare&swap, fetch&xor or read they apply to R. Special care should be taken for silent READ and WRITE operations. Indeed, a silent READ that reads x from SN, may return the previous value u stored in the implemented register or v, depending on the sequence number of the last preceding direct READ by the same process. Similarly, a silent WRITE(v') may not access R at all, or apply a compare&swap after $R.seq$ has already been changed to $x + 1$. However, WRITE(v') has to be linearized before WRITE(v), in such a way that v' is immediately overwritten.

Hence, direct READ, visible WRITE, and AUDIT are linearized first, according to the order in which they apply a primitive to R. We then place the remaining operations with respect to this partial linearization. $L(\alpha)$ is the total order on the operations in H' obtained by the following rules:

R1 For direct READ, visible WRITE, AUDIT and some silent READ operations we defined an associated step ls applied by the operation. These operations are then ordered according to the order in which their associated step takes place in α. For a direct READ, visible WRITE, or AUDIT operation op, its associated step $ls(op)$ is respectively the fetch&xor at line 4, the successful compare&swap at line 14, and the read at line 17 applied to R. For a silent READ operation op with sequence number $sn(op) = x$, if SN.read (line 2) is applied in op during E_x (that is, $R.seq = x$ when this read occurs), $ls(op)$ is this read step. The other silent READ operations do not have a linearization step, and are not ordered by this rule. They are instead linearized by Rule R2.

Recall that ρ_{x+1} is the successful compare&swap applied to R that changes $R.seq$ from x to $x + 1$ (Lemma 1). By rule R1, the visible WRITE with sequence number $x + 1$ is linearized at ρ_{x+1}.

R2 For every $x \geq 0$, every remaining silent READ op with sequence number $sn(op) = x$ is placed immediately before the unique visible WRITE operation with sequence number $x + 1$. Their relative order follows the order in which their read step of SN (line 2) is applied in α.

R3 Finally, we place for each $x \geq 0$ every silent WRITE operation op with sequence number $sn(op) = x + 1$. They are placed after the silent READ operations with sequence number x ordered according to rule R2, and before the unique visible WRITE operation with sequence number $x+1$. As above, their respective order is determined by the order in which their read step of SN (line 8) is applied in α.

Rules R2 and R3 are well-defined, in [3] we prove the existence and uniqueness of a visible WRITE with sequence number x, if there is an operation op with $sn(op) = x$.

We can show that the linearization $L(\alpha)$ extends the real-time order between operations, and that the READ and WRITE operations satisfy the sequential specification of a register.

Audit Properties. For the rest of the proof, fix a finite execution α. The next lemma helps to show that effective operations are audited; it demonstrates how indistinguishability is used in our proofs.

Lemma 3. *A* READ *operation rop that is invoked in α is in $L(\alpha)$ if and only if rop is effective in α.*

Proof. If *rop* completes in α, then it is effective and it is in $L(\alpha)$. Otherwise, *rop* is pending after α. Let p_j be the process that invokes *rop*. We can show:

Claim 4. *rop is effective after α if and only if either*
(1) p_j has read x from SN and $x = prev\_sn$ (line 2) or
(2) p_j has applied fetch&xor *to R (line 4).*

Proof. First, let α' be an arbitrary extension of α in which *rop* returns some value a, β a finite execution indistinguishable from α to p_j, and β' one of its extensions in which *rop* returns some value b. We show that if α satisfies (1) or (2), then $a = b$. (1) If in α after invoking *rop*, p_j reads $x = prev\_sn$ from SN at line 2, then *rop* returns $a = prev\_val$ in α'. Since $\alpha \overset{p_j}{\sim} \beta$, $prev\_val = a$ and $prev\_sn = x$ when *rop* starts in β, and p_j reads also x from SN. Therefore, *rop* returns $b = a$ in β'. (2) If p_j applies a fetch&xor to R (line 4) while performing *rop* in α, then *rop* returns $a = v$ (line 6), where v is the value fetched from $R.val$ in α'. Since $\alpha \overset{p_j}{\sim} \beta$, p_j also applies a fetch&xor to R while performing *rop* in β, and fetches v from $R.val$. Therefore *rop* also returns v in β'.

Conversely, suppose that neither (1) nor (2) hold for α. That is, p_j has not applied a fetch&xor to R and, if x has been read from SN, $x \neq prev\_sn$. We construct two extensions α' and α'' in which *rop* returns $v' \neq v''$, respectively. Let X be the value of SN at the end of α, and p_i be a writer. In α', p_i first completes its pending WRITE if it has one, before repeatedly writing the same value v' until performing a visible WRITE(v'). Finally, p_j completes *rop*. Since p_i is the only writer that takes steps in α, it eventually has a visible WRITE(v'), that is in which $R.val$ is changed to v'. Note also that when this happens, $SN > X$. The extension α'' is similar, except that v' is replaced by v''.

Since conditions (1) and (2) do not hold, p_i's next step in *rop* is reading SN or issuing R.fetch&xor. If p_j reads SN after resuming *rop*, it gets a value $x > prev\_val$. Thus, in both cases, p_j accesses R in which it reads $R.val = v'$ (or $R.val = v''$). Therefore, *rop* returns v' in α' and v'' in α''. □

Now, if (1) holds (p_j reads $x = prev\_val$ from SN at line 2), then *rop* is classified as a silent READ, and it appears in $L(\alpha)$, by rule $R1$ if $R.seq = x$ when SN is read or rule $R2$, otherwise. If (2) holds (p_j applies a fetch&xor to R), then *op* is a direct READ, and linearized in $L(\alpha)$ by rule $R1$.

If neither (1) nor (2) hold, then p_j has either not read SN, or read a value $\neq prev\_val$ from SN but without yet accessing R. In both cases, *op* is unclassified and hence not linearized. □

We can prove that an audit *aop* includes a pair (j, v) in its response set *if and only if* a READ operation by process p_j with output v is linearized before it. Since a READ is linearized if and only it is effective (Lemma 3), any AUDIT operation that is linearized after the READ is effective, must report it. This implies:

Lemma 5. *If an* AUDIT *operation aop is invoked and returns in an extension α' of α, and α contains a v-effective* READ *operation by process p_j, then (j, v) is contained in the response set of aop.*

Lemma 6 shows that writes are uncompromised by readers, namely, a read cannot learn of a value written, unless it has an effective READ that returned this value. Lemma 7 shows that reads are uncompromised by other readers, namely, they do not learn of each other.

Lemma 6. *Assume p_j only performs* READ *operations. Then for every value v either there is a* READ *operation by p_j in α that is v-effective, or there is α', $\alpha' \overset{p_j}{\sim} \alpha$ in which no* WRITE *has input v.*

Proof. If v is not an input of some WRITE operation in α, the lemma follows by taking $\alpha' = \alpha$. If there is no visible WRITE(v) operation in α, then, since a silent WRITE(v) does not change $R.val$ to v, the lemma follows by changing its input to some value $v' \neq v$ to obtain an execution $\alpha' \overset{p_j}{\sim} \alpha$

Let *wop* be a visible WRITE(v) operation in α. Since it is visible, *wop* applies a compare&swap to R that changes $(R.seq, R.val)$ to (x, v) where x is some sequence number. If p_j applies a fetch&xor to R while $R.val = v$, then the corresponding READ operation *rop* it is performing is direct and v-effective. Otherwise, p_j never applies a fetch&xor to R while $R.val = v$. R is the only shared variable in which inputs of WRITE are written and that is read by p_j. Hence, the input of *wop* can be replaced by another value $v' \neq v$, creating an indistinguishable execution α' without a WRITE with input v. □

Lemma 7. *Assume p_j only performs* READ *operations, then for any reader p_k, $k \neq j$, there is an execution $\alpha' \overset{p_j}{\sim} \alpha$ in which no* READ *by p_k is v-effective, for any value v.*

Proof. The lemma clearly holdes if there is no v-effective READ by process p_k. So, assume there is a v-effective READ operation *rop* by p_k. Let α' be the execution in which we remove all v-effective READ operations performed by p_k that are silent. Such operations do not change any shared variables, and therefore, $\alpha' \overset{p_j}{\sim} \alpha$.

So, let *rop* be a direct, v-effective READ by p_k. When performing *rop*, p_k applies fetch&xor to R (line 4), when $(R.seq, R.val) = (x, v)$, for some sequence number x. This step only changes the kth tracking bit of R unchanged to, say, b. Recall that R is accessed (by applying a fetch&xor) at most once by p_j while $R.seq = x$. If no fetch&xor by p_j is applied to R while $R.seq = x$, or one is applied before p_k's, *rop* can be removed without being noticed by p_j. Suppose that both p_k and p_j apply a fetch&xor to R while $R.seq = x$, and that p_j's fetch&xor is after p_k's. Let $\alpha'_{x,b}$ be the execution identical to α', except that (1) the kth bit of $rand_x$ is b and, (2) *rop* is removed. Therefore, $\alpha'_{x,b} \overset{p_j}{\sim} \alpha'$, and since $\alpha' \overset{p_j}{\sim} \alpha$, we have that $\alpha'_{x,b} \overset{p_j}{\sim} \alpha$. □

Theorem 8. *Alg. 1 is a linearizable and wait-free implementation of an auditable multi-writer, multi-reader register. Moreover,*

- *An* AUDIT *reports (j, v) if and only if p_j has an v-effective* READ *operation in α.*
- *a* WRITE *is uncompromised by a reader p_j, unless p_j has a v-effective* READ*.*
- *a* READ *by p_k is uncompromised by a reader $p_j \neq p_k$.*

4 An Auditable Max Register

This section shows how to extend the register implementation of the previous section into an implementation of a max register with the same properties. A *max register* provides two operations: WRITEMAX(v) which writes a value v and READ which returns a value. Its sequential specification is that a READ returns the largest value previously written. An auditable max register also provides an AUDIT operation, which returns a set of pairs (j, v). As in the previous section, reads are audited if and only if they are effective, and readers cannot compromise other WRITEMAX operations, unless they read them, or other READ operations.

Alg. 2 uses essentially the same READ and AUDIT as in Alg. 1. The WRITEMAX operation is also quite similar, with the following differences (lines in blue in the pseudo-code). In Alg. 1, a WRITE(w) obtains a new sequence number $s + 1$ and then attempts to change R to $(s + 1, w, rand_{s+1})$. The operation terminates after it succeeds in doing so, or if it sees in R a sequence number $s' \geq s + 1$. In the latter case, a concurrent WRITE(w') has succeeded and may be seen as overwriting w, so WRITE(w) can terminate, even if w is never written to R. The implementation of WRITEMAX uses a similar idea, except that (1) we make sure that the successive values in R are non-decreasing and (2) a WRITEMAX(w) with sequence number $s+1$ is no longer abandoned when a sequence number $s' \geq s + 1$ is read from R, but instead when R stores a value $w' \geq w$.

There is however, a subtlety that must be taken care of. A reader may obtain a value v with sequence number s, and later read a value $v + 2$ with sequence number $s' > s + 1$. This leaks to the reader that some WRITEMAX operations occur in between its READ operations, and in particular, that a WRITEMAX($v + 1$) occurred, without ever effectively reading $v + 1$.

To deal with this problem, we append a *random nonce* N to the argument of a WRITEMAX operation, where N is a random number. The pair (w, N) is used as the value written v was used in Alg. 1. The pairs (w, N) are ordered lexicographically, that is, first by their value w and then by their nonce N. Thus, the reader cannot guess intermediate values. The code for READ and AUDIT is slightly adjusted in Alg. 2 versus Alg. 1, to ignore the random nonce N from the pairs when values are returned.

In the algorithm, a (non-auditable) max-register M is shared among the writers. A WRITEMAX(w) by p starts by writing the pair $v = (w, N)$ of the value w and the nonce N to M, before entering a repeat loop. Each iteration is an attempt to store in R the current value *mval* of M, and the loop terminates as soon as R holds a value equal to or larger than *mval*. Like in Alg. 1, R holds a triplet $(s, val, bits)$ where s is *val*'s sequence number, *val* is the current value, and *bits* is the encrypted set of readers of *val*. Before attempting to change R, *val* and the set of readers, once deciphered, are stored in the registers $V[s]$ and $B[s]$, from which they can be retrieved with AUDIT.

In each iteration of the repeat loop, the access pattern of WRITE in Alg. 1 to the shared register *SN* and R is preserved. After obtaining a new sequence number $s + 1$, where s is the current value of *SN* (line 24 for the first iteration, line 30 otherwise), a triple $(lsn, lval, bits)$ is read from R. If $lval \geq v$, the loop breaks as a value that is equal to or larger than v has already been written. As in Alg. 1, before returning we make sure that the sequence number in *SN* is at least as large as *lsn*, the sequence number in R.

Algorithm 2 Auditable Max Register

```
    shared registers
        $R, SN, V[0..+\infty], B[0..+\infty][0..m-1]$ as in Alg. 1
        $M$: a (non-auditable) max register, initially $v_0 = (w_0, N_0)$
    local variables: writer, reader, auditor,  as in Alg. 1
21: function READ( ), AUDIT( ): same as in Alg 1
22: function WRITEMAX($w$)
23:     $v \leftarrow (w, N)$, where $N$ is a fresh random nonce
24:     $M$.writeMax($v$); $sn \leftarrow SN$.read() + 1;
25:     repeat
26:         $(lsn, lval, bits) \leftarrow R$.read()
27:         if $lval \geq v$ then $sn \leftarrow lsn$; break
28:         if $lsn \geq sn$ then
29:             $SN$.compare&swap($sn - 1, sn$);
30:             $sn \leftarrow SN$.read() + 1;  continue
31:         $mval \leftarrow M$.read()
32:         $V[lsn]$.write($lval.value$);
33:         $B[lsn][j]$.write($true$) $\forall j$, s.t. $bits[j] \neq rand_{lsn}[j]$
34:     until $R$.compare&swap($(lsn, lval, bits), (sn, mval, rand_{sn})$)
35:     $SN$.compare&swap($sn - 1, sn$); return
```

5 Auditable Snapshot Objects and Versioned Types

We show how an auditable max register (Section 4) can be used to make other object types auditable.

5.1 Making Snapshots Auditable

We start by showing how to implement an auditable n-component snapshot object, relying on an auditable max register. Each component has a state, initially $\perp$, and a different designated writer process. A *view* is an n-component array, each cell holding a value written by a process in its component. A *atomic object* [1] provides two operations: UPDATE(v) that changes the process's component to v, and SCAN that returns a view. It is required that in any sequential execution, in the view returned by a SCAN, each component contains the value of the latest UPDATE to this component (or $\perp$ if there is no previous UPDATE). As for the auditable register, an AUDIT operation returns a set of pairs $(j, view)$. In a sequential execution, there is such a pair if and only if the operation is preceded by a SCAN by process p_j that returns *view*. Here, we want that audits report exactly those SCANS that have made enough progress to infer the current *view* of the object.

Denysuk and Woeffel [12] show that a strongly-linearizable max register can be used to transform a linearizable snapshot into its strongly linearizable counterpart. As we explain next, with the same technique, non-auditable snapshot objects can be made auditable. Algorithm 3 adds an AUDIT operation to their algorithm. Their implementation is lock-free, as they rely on a lock-free implementation of a max register. Algorithm 3 is wait-free since we use the *wait-free* max-register implementation of Section 4.

Let S be a linearizable, but non-auditable snapshot object. The algorithm works as follows: each new state (that is, whenever one

Algorithm 3 n-component auditable snapshot objects.

```
    shared registers
        $M$: auditable max register, initially $(0, [\bot, \ldots, \bot])$
        $S$: (non-auditable) snapshot object,
            initially $[(0, \bot), \ldots, (0, \bot)]$
    local variable: writer $p_i$, $1 \le i \le n$
        $sn_i$ local sequence number, initially 0
1:  function UPDATE($v$)                ▷ code for writer $p_i, i \in \{1, \ldots, n\}$
2:      $sn_i \leftarrow sn_i + 1$; $S.\mathsf{update}_i((sn_i, v))$
3:      $sview \leftarrow S.\mathsf{scan}()$; $vn \leftarrow \sum_{1 \le j \le n} sview[j].sn$
4:      $view \leftarrow$ the $n$-component array of the values in $sview$
5:      $M.\mathsf{writeMax}((vn, view))$; return
6:  function SCAN()
7:      $(\_, view) \leftarrow M.\mathsf{read}()$; return $view$
8:  function AUDIT()
9:      $MA \leftarrow M.\mathsf{audit}()$;
10:     return $\{(j, view) : \exists$ an element $(j, (*, view)) \in MA\}$
```

component is updated) is associated with a unique and increasing *version number*. The version number is obtained by storing a sequence number sn_i in each component i of S, in addition to its current value. Sequence number sn_i is incremented each time the ith component is updated (line 2). Summing the sequence numbers of the components yields a unique and increasing version number (vn) for the current view.

The pairs $(vn, view)$, where vn is a version number and $view$ a state of the auditable snapshot, are written to an auditable max register M. The pairs are ordered according to the version number, which is a total order since version numbers are unique. Therefore, the latest state can be retrieved by reading M, and the set of past SCAN operations can be obtained by auditing M (line 10). The current view of the auditable snapshot is stored in S.

In an UPDATE(v), process p_i starts by updating the ith component of S with v and incrementing the sequence number field sn_i. It then scans S, thus obtaining a new view of S that includes its update. The view $view$ of the implemented auditable snapshot is obtained by removing the sequence number in each component (line 4). The version number vn associated with this view is the sum of the sequence numbers. It then writes $(vn, view)$ to the max-register M (line 5). A SCAN operation reads a pair $(vn, view)$ from M and returns the corresponding $view$ (line 7). Since M is auditable, the views returned by the processes that have previously performed a SCAN can thus be inferred by auditing M (line 10).

The AUDIT and SCAN operations interact with the implementation by applying a single operation (audit and read, respectively) to the auditable max register M. The algorithm therefore lifts the properties of the implementation of M to the auditable snapshot object. In particular, when the implementation presented in Section 4 is used, effective SCAN operations are auditable, SCAN operations are uncompromised by other scanners, and UPDATE operations are uncompromised by scanners.

5.2 Proof of Correctness

Let α be a finite execution of Algorithm 3. To simplify the proof, we assume the inputs of UPDATE by the same process are unique.

We assume that the implementation of M is wait-free and linearizable. In addition, it guarantees effective linearizability and that READ operations are uncompromised by other readers. We also assume that the implementation of S is linearizable and wait-free (e.g.,[1]). Inspection of the code shows that UPDATE, SCAN and AUDIT operations are wait-free.

Since S and M are linearizable and linearizability is composable, α can be seen as a sequence of steps applied to S or M. In particular, we associate with each high-level operation op a step $\sigma(op)$ applied by op either to S or to M. The linearization $L(\alpha)$ of α is the sequence formed by ordering the operations according to the order their associated step occurs in α.

For a SCAN and an AUDIT operation op, $\sigma(op)$ is, respectively, the read and the audit steps applied to M. If op is an UPDATE with input x by process p_i, then let vn_x be the sum of the sequence numbers sn in each component of S after $update(x)$ has been applied to S by p_i. $\sigma(op)$ is the first write to M of a pair $(vn, view)$ with $vn \ge vn_x$ and $view[i] = x$. If there is no such write, op is discarded.

We first show that the linearization $L(\alpha)$ respects the real-time order between operations.

Lemma 9. *If an operation op completes before an operation op' is invoked in α, then op precedes op' in $L(\alpha)$.*

Proof. We show that that the linearization point of any operation op is inside its execution interval; the claim is trivial for SCAN or AUDIT operations.

Suppose that op is an UPDATE by a process p_i with input x. The sum of the sequence numbers in the components of S increases each time an update is applied to it. Hence, any pair $(vn, view)$ written to M before p_i has updated its component of S to x is such that $vn < vn_x$. Therefore $\sigma(op)$, if it exists, is after op starts. If op terminates, then it scans S after updating the ith component of S to x. The $view$ it obtains and its associated version number satisfy $view[i] = x$ and $vn \ge vn_x$. This pair is written to M. If $\sigma(op)$ is not this step, then $\sigma(op)$ occurs before op terminates. If op does not terminate and $\sigma(op)$ does exist, it occurs after op starts and thus within op's execution interval. □

Lemma 10. *Each component i of the view returned by a SCAN is the input of the last UPDATE by p_i linearized before the SCAN in $L(\alpha)$.*

Proof. Consider a SCAN operation sop that returns $view$, with $view[i] = x$. This view is read from the max register M and has version number vn. Let op be the last UPDATE by p_i linearized before sop in $L(\alpha)$, let y be its input and vn_y the version number (that is the sum of the sequence number stored in each component) of S immediately after $S.\mathsf{update}(y)$ is applied by p_i.

We denote by σ_u this low level update. Since the version number increases with each update, every pair $(vn', view')$ written into M before σ_u is such that $vn' < vn_y$. Also, every pair $(vn', view')$ written to M after σ_u and before sop is linearized satisfies $vn' \ge vn_y \implies view'[i] = y$. Indeed, if $vn' \ge vn_y$, $view'$ is obtained by a scan of S applied after the i-th component is set to y. Hence, $view'[i] = y$ because we assume that op is the last UPDATE by p_i linearized before sop in $L(\alpha)$.

Finally, step $\sigma(op)$ is a write of pair $(vn', view')$ to M with $vn' \ge vn_y$ and $view'[i] = y$. $\sigma(op)$ occurs after σ_u and before the max register M is read by sop. It thus follows that the pair $(vn, view)$ read from M in sop satisfies $vn \geq vn_y$ and has been written after σ_y. Hence, $view[i] = y = x$. We conclude that each component i of the view returned by a SCAN is the input of the last UPDATE by p_i linearized before the SCAN in $L(\alpha)$. □

Lemma 11. *An AUDIT reports $(j, view)$ if and only if p_j has a view-effective[2] SCAN in α. Each UPDATE(v) is uncompromised by a scanner p_j unless it has a view-effective SCAN with one component of view equal to v. Each SCAN by p_k is uncompromised by a scanner $p_j \neq p_k$.*

Proof. A SCAN applies a single operation on shared objects, namely a read on M. It is linearized with this step, which determines the view it returns. Therefore, a SCAN is linearized if and only if it is effective. Hence $(j, view)$ is reported by an AUDIT if and only if p_j has a $view$-effective SCAN.

Let v be the input of an UPDATE operation by some process p_i. If there is no $view$ with $view[i] = v$ written to M (line 5), UPDATE(v) can be replaced by UPDATE(v'), $v' \neq v$ in an execution α', $\alpha \overset{p_j}{\sim} \alpha'$. Otherwise, note that each $sview$ for which p_j has a $sview$-effective SCAN, we have $sview[i] \neq v$. Suppose that $view$, with $view[i] = v$ is written to M in α. Then we can replace $view$ with an array $view'$, identical to $view$ except that $view'[i] = v' \neq v$ an execution $\alpha' \overset{p_j}{\sim} \alpha$. This is because the write of $view$ is not compromised by p_j in M. By repeating this procedure until all writes to M of $view$s with $view[i] = v$ have been eliminated leads to an execution β, $\beta \overset{p_j}{\sim} \alpha$ in which there is no UPDATE(v). □

Theorem 12. *Alg. 3 is a wait-free linearizable implementation of an auditable snapshot object which audits effective SCAN operations, in which SCAN and UPDATE are uncompromised by scanners.*

5.3 Versioned Objects

Snapshot objects are an example of a *versioned type* [12], whose successive states are associated with unique and increasing version numbers. Furthermore, the version number can be obtained from the object itself, without resorting to external synchronization primitives. Essentially the same construction can be applied to any versioned object.

An object $t \in \mathcal{T}$ is specified by a tuple (Q, q_0, I, O, f, g), where Q is the state space, I and O are respectively the input and output sets of update and read operations. q_0 is the initial state and functions $f : Q \longrightarrow O$ and $g : I \times Q \longrightarrow Q$ describes the sequential behavior of read and update. A read() operation leaves the current state q unmodified and returns $f(q)$. An update(v), where $v \in I$ changes the state q to $g(v, q)$ and does not return anything.

A linearizable *versioned* implementation of a type $t \in \mathcal{T}$ can be transformed into a strongly-linearizable one [12], as follows. Let $t = (Q, q_0, I, O, f, g)$ be some type in $\mathcal{T}$. Its *versioned* variant $t' = (Q', q'_0, I', O', f', g')$ has $Q' = Q \times \mathbb{N}$, $q'_0 = (q_0, 0)$, $I' = I$, $O' = O \times \mathbb{N}, f' : Q' \longrightarrow O \times \mathbb{N}$ and $g' : I \times Q' \longrightarrow Q'$. That is, the state of t' is augmented with a version number, which increases with each update and is returned by each read: $f'((q, vn)) = (f(q), vn)$ and $g'((q, vn)) = (g(q), vn')$ with $vn < vn'$.

A versioned implementation of a type $t \in \mathcal{T}$ can be transformed into an auditable implementation of the same type using an auditable register. The construction is essentially the same as presented in Algorithm 3. In the auditable variant T_a of T, to perform an UPDATE(v), a process p first update the versioned implementation T before reading it. p hence obtains a pair (o, vn) that it writes to the auditable max register M. For a READ, a process returns what it reads from M. As READ amounts to read M, to perform an AUDIT a process simply audit the max-register M. As we have seen for snapshots, T_a is linearizable and wait-free. Moreover, T_a inherits the advanced properties of the underlying max-register: If M is implemented with Algorithm 2, then it correctly audits effective READ, and READ and UPDATE are uncompromised.

Theorem 13 (versioned types are auditable). *Let $t \in \mathcal{T}$, and let T be a versioned implementation of t that is linearizable and wait-free. There exists a wait-free, linearizable and auditable implementation of t from T and auditable max-registers in which READ and UPDATE are uncompromised by readers and AUDIT reports only effective READ operations.*

6 Discussion

This paper introduces novel notions of auditability that deal with curious readers. We implement a wait-free linearizable auditable register that tracks effective reads while preventing unauthorized audits by readers. This implementation is extended into an auditable max register, which is then used to implement auditable atomic snapshots and versioned types.

Many open questions remain for future research. An immediate question is how to implement an auditable register in which *only auditors can audit*, i.e., reads are uncompromised by writers. A second open question is how to extend auditing to additional objects. These can include, for example, *partial snapshots* [5] in which a reader can obtain an "instantaneous" view of a subset of the components. Another interesting object is a *clickable* atomic snapshot [17], in particular, variants that allow arbitrary operations on the components and not just simple updates (writes).

The property of uncompromising other accesses can be seen as an *internal* analog of *history independence*, recently investigated for concurrent objects [4]. A history-independent object does not allow an external observer, *having access to the complete system state*, to learn anything about operations applied to the object, but only its current state. Our definition, on the other hand, does not allow an internal observer, e.g., a reader that only reads shared base objects, to learn about other READ and WRITE operations applied in the past. An interesting intermediate concept would allow several readers *collude* and to combine the information they obtain in order to learn more than what they are allowed to.

Acknowledgments

H. Attiya is supported by the Israel Science Foundation (grant number 22/1425). A. Fernández Anta is supported by MICIU / AEI / 10.13039 / 501100011033 and ERDF, EU (grant PID2022-140560OB-I00). A. Milani is supported by the France 2030 ANR (project ANR-23-PECL-0009 TRUSTINCloudS). C. Travers is supported in part by the ANR (project ANR-20-CE48-0006 DUCAT).

[2] Namely, p_i has a SCAN operation that returns $view$ in all indistinguishable executions.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (1993), 873–890. doi:10.1145/153724.153741

[2] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. 2012. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM* 59, 1 (2012), 2:1–2:24.

[3] Hagit Attiya, Antonio Fernández Anta, Alessia Milani, Alexandre Rapetti, and Corentin Travers. 2025. *Auditing without Leaks Despite Curiosity*. Technical Report 2505.00665. arXiv. doi:10.48550/arXiv.2505.00665 Full version of this paper.

[4] Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2024. History-Independent Concurrent Objects. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing (PODC)*. Association for Computing Machinery, New York, NY, USA, 14–24. doi:10.1145/3662158.3662814 Full version in https://doi.org/10.48550/arXiv.2403.14445.

[5] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. 2008. Partial snapshot objects. In *20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Association for Computing Machinery, New York, NY, USA, 336–343. doi:10.1145/1378533.1378591

[6] Hagit Attiya, Antonella Del Pozzo, Alessia Milani, Ulysse Pavloff, and Alexandre Rapetti. 2023. The Synchronization Power of Auditable Registers. In *27th International Conference on Principles of Distributed Systems (OPODIS)*, Vol. 286. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:23. doi:10.4230/LIPICS.OPODIS.2023.4 Full version in https://doi.org/10.48550/arXiv.2308.16600.

[7] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. 2023. As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy! *J. Parallel Distributed Comput.* 181 (2023), 104743. doi:10.1016/J.JPDC.2023.104743

[8] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Seredinschi. 2022. 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*. IEEE, 34–44. doi:10.1109/ICDCS54860.2022.00013

[9] Vinicius Vielmo Cogo and Alysson Bessani. 2021. Brief Announcement: Auditable Register Emulations. In *35th International Symposium on Distributed Computing (DISC)*, Vol. 209. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 53:1–53:4. doi:10.4230/LIPICS.DISC.2021.53 Full version in http://arxiv.org/abs/1905.08637.

[10] cpp [n. d.]. https://en.cppreference.com/mwiki/index.php?title=cpp/thread&oldid=179906#Operations_on_atomic_types.

[11] Antonella Del Pozzo, Alessia Milani, and Alexandre Rapetti. 2022. Byzantine Auditable Atomic Register with Optimal Resilience. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, IEEE, 121–132.

[12] Oksana Denysyuk and Philipp Woelfel. 2015. Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability. In *29th International Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 60–74. doi:10.1007/978-3-662-48653-5_5

[13] Davide Frey, Mathieu Gestin, and Michel Raynal. 2023. The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList. In *37th International Symposium on Distributed Computing, (DISC)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:23. doi:10.4230/LIPICS.DISC.2023.21

[14] Oded Goldreich. 2003. Cryptography and cryptographic protocols. *Distributed Comput.* 16, 2-3 (2003), 177–199. doi:10.1007/S00446-002-0077-1

[15] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

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

[17] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. 2024. MemSnap: A Fast Adaptive Snapshot Algorithm for RMWable Shared-Memory. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing, PODC 2024, Nantes, France, June 17-21, 2024*, Ran Gelles, Dennis Olivetti, and Petr Kuznetsov (Eds.). Association for Computing Machinery, New York, NY, USA, 25–35. doi:10.1145/3662158.3662820

[18] Frank Miller. 2024. *Telegraphic code to insure privacy and secrecy in the transmission of telegrams*. BoD–Books on Demand.

[19] Antonella Del Pozzo and Thibault Rieutord. 2022. Fork Accountability in Tenderbake. In *5th International Symposium on Foundations and Applications of Blockchain 2022, FAB 2022, June 3, 2022, Berkeley, CA, USA (OASIcs, Vol. 101)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:22. doi:10.4230/OASIcs.FAB.2022.5

[20] Gilbert S. Vernam. U.S. Patent 1,310,719, July 1919. Secret Signaling System. https://patents.google.com/patent/US1310719