To the Best of Knowledge and Belief: On Eventually Consistent Access Control

Florian Jacob
florian.jacob@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Hannes Hartenstein
hannes.hartenstein@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

We are used to the conventional model of linearizable access control (LAC), implemented by a trusted central entity or by a set of distributed entities that coordinate to mimic a central entity. The strength of LAC is rooted in the dependencies among entities, at the cost of reduced availability, scalability, and resilience under faults. Systems that cannot afford dependencies among entities, like the ones based on conflict-free replicated data types (CRDTs), must break with the LAC convention, but gain fundamental advantages in availability, scalability, and resilience. In this paper, we formalize eventually consistent access control (ECAC) that replaces up-front coordination with subsequent reconciliation, and study its theoretical guarantees in Byzantine environment at the practical example of Matrix, a CRDT-based group communication system. Our core finding is that ECAC implies authorization to the best of knowledge and belief: an entity *stores* an action only if the action is authorized by *immutable knowledge* derived from its *final set* of preceding actions, and *executes* an action only if it is also authorized by the entity's *mutable beliefs* derived from the *grow-only set* of concurrent actions.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Distributed systems security**; • **Software and its engineering** → **Consistency**; *Publish-subscribe / event-based architectures*; • **Information systems** → **Distributed storage**; • **Computer systems organization** → *Distributed architectures*; *Availability*; **Dependable and fault-tolerant systems and networks**; *Reliability*.

KEYWORDS

Access Control, Autonomous Decentralized Systems, Eventual Consistency, Conflict-Free Replicated Data Types, Byzantine Fault Tolerance, Logical Clocks, Logical Monotonicity, Matrix

1 INTRODUCTION

Coordination takes time and results in dependencies to other entities. Therefore, system designers often strive to make time-critical actions independent of the latency to other entities. While there is yet no unified terminology for this class of systems, the movement behind autonomous decentralized systems [24], local-first [18], coordination-avoiding [5], or wait-free [10] systems follow this design principle, which could be called 'act now, reconcile later' in accordance with a famous quote. In the realm of data consistency, this principle restricts achievable consistency models to eventual consistency and causal consistency [1]. However, what does this principle mean for access control?

In the security community, we are used to what we call linearizable access control (LAC): the access control architecture acts as a single, logically centralized entity that stores, orders, decides on, and enforces all access control policies. Of course, the logically centralized approach can be implemented as a distributed system, which leads to coordination-based linearizable access control (CLAC): a set of distributed system entities needs to coordinate on policy information, decisions, and enforcement to keep up the LAC model, and, again, is fundamentally prone to processor and network faults and latencies.

In this paper, we study an approach, which we call eventually consistent access control (ECAC), that breaks with the convention of (coordination-based) logical centralization. In ECAC, the set of system entities implements a logically decentralized access control architecture: Every system entity autonomously stores, decides on, and enforces access control policies to its best of knowledge and belief on the overall current system state. To ensure that the access control policies and decisions between entities eventually converge, up-front coordination among system entities is replaced with subsequent reconciliation:

CLAC:	**ECAC:**
(1) Coordination →	(3) Reconciliation
(2) Decision	(1) Decision
(3) Access	(2) Access

At first glance, giving up on CLAC semantics by replacing coordinated decisions with accountable best-effort decisions seems like a prohibitive trade-off to make. We argue, however, that ECAC variants are found in many deployed systems that prioritize availability, scalability, or fault tolerance: for example, offline payments in planes or offline withdrawals at ATMs prioritize availability over coordination, and reconciliation allows to audit for overdraft later. Electronic door locks also typically provide best-effort service if the network is partitioned, as the risk of unauthorized people getting in due to a stale policy is much more acceptable to business operation than the risk of authorized people not getting in due to network outage, also potentially stopping them from fixing the network outage in the first place [8]. Furthermore, current practice of PKI certificate validation in web browsers does a local decision and updates trusted (root) certificates only from time to time.

The challenge of ECAC is to deal with concurrent policy updates, in particular for revocations. Even more challenging, one needs to be able to deal with Byzantine entities. In this paper, we identify the invariants of access control under Byzantine eventual consistency. The goal is to *understand and characterize* ECAC, it is not the goal to claim 'superiority' over classical approaches.

Our approach of investigation starts with Matrix, a deployed decentralized system for group communication and data storage [30] that has implemented decentralized access control. Matrix shows wide adoption: nation states like France, Sweden, and Germany operate private federations for their public sectors [11, 19], the United Nations International Computing Center has switched to Matrix as communication platform provided to UN organizations [20], and more than 100 000 000 users on more than 100 000 servers are found in the public federation. The underlying data structure of Matrix has been shown to represent a conflict-free replicate data type (CRDT) even in Byzantine setups [13]. Figure 1 shows the paper's line of reasoning: we combine the practical approach of Matrix with the theory of Byzantine-tolerant CRDTs and logical monotonicity, and abstract it to reach our main result, a conceptual model of ECAC.

We, therefore, provide three main contributions:

- The ECAC model is our propositional answer to what kind of access control is achievable in decentralized systems. The model consists of a set of properties that are both provided guarantees to applications as well as necessary conditions on its implementing algorithms.
- An assessment of the consequences shown by partitioning, equivocation, and backdating in the ECAC model.
- Matrix and other practical systems already represent proofs by example of ECAC's implementability. For a comprehensible demonstration open to scrutiny, we provide an abstract algorithm based on the Matrix specification, and verify that it fulfills ECAC's necessary conditions.

In particular, we show that the *audit log* of ECAC, in which every entity records all actions that it decided as authorized and that is reconciled during favorable network conditions, provides a partial (causal) order of policy updates and decisions. This causal order with its corresponding concurrency allows us to separate authorizations based on definitive knowledge, i.e., knowledge that is final, from those based on mutable 'beliefs', i.e., the set of concurrent, potentially applicable policy updates is grow-only and never final due to missing consensus on the audit log.

We first present an overview of the Matrix approach in Section 2 together with fundamentals on conflict-free replicated data types. The problem statement of ECAC is presented in Section 3. We formalize the security guarantees that the ECAC model can provide in Section 4. We assess the ECAC model using a set of scenarios in Section 5, and demonstrate its implementability through an abstract algorithm based on the Matrix system. Finally, we conclude in Section 6. We include related work in-place where relevant.

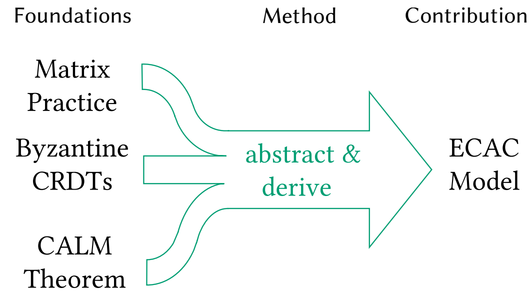


Figure 1: We take the consensus-free approach of Matrix to a decentralized access control architecture, combine it with the theory of Byzantine fault-tolerant Conflict-Free Replicated Data Types (CRDTs) and the Consistency as Logical Monotonicity (CALM) theorem, and derive a property-based model of Eventually-Consistent Access Control (ECAC).

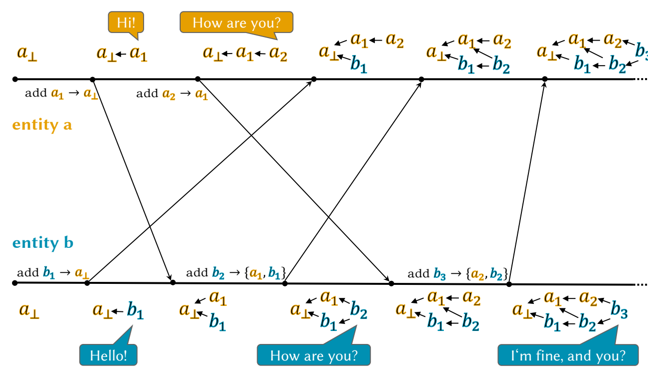


Figure 2: Matrix-based chat example for replicating a chronicle, i.e., a causally-ordered set of events. An event x_n is created by entity x, and points to its direct predecessors. The unique event without predecessors $x_\perp$ is called the genesis event, an event x_n has a longest path of length n to $x_\perp$. Both entities a, b concurrently add **new events to the chronicle. Correct entities independently verify authorization of an event before creation and before adding it to their local state.**

2 MATRIX FUNDAMENTALS

Matrix is a decentralized system[1] that provides group communication and data storage [30]. Matrix stands out from other decentralized systems due to targeting open networks with Byzantine participants, and its emphasis on decentralized access control. The state of a Matrix *communication group* (called "room" in Matrix) consists mainly of its communication history, but also includes group metadata like membership, attributes of group members and the group itself, as well as access control policies and permissions. Instead of storing mutable group state directly, Matrix stores a grow-only, partially-ordered set of immutable group state change *events*, which are executed to derive the current group state. The core of Matrix is the CRDT-based replication of event sets, exemplified in Fig. 2, among all entities participating in a communication

[1]This work speaks of decentralized systems as the subclass of distributed systems that does not employ consensus or any other form of coordination, like CRDTs.

group [14]. Events encapsulate the creating subject's action on an optional object. For example, we say a chat message is an event with an action of type `cht` with content `"Hi!"` and its sender as subject. Events include hash links to their direct predecessor events, chosen at the discretion of the event's creator. As the ordered event set describes a causality relation among events, integrity-protected by hash links, we refer to hash-linked event sets as *hash chronicles* [15] (called "event graphs" in Matrix). Similar to an organization's or individual's email server, Matrix servers act as trusted representative for their users. In this work, we assume that a server has only one user, and treat server and user as single *entity*. The required trust among servers is limited by performing decentralized access control, i.e., every entity performs its own, independent authorization decision before adding an event to its chronicle.

Decentralized authorizations in Matrix are expressed using the Level- and Attribute-based Access Control (LeABAC) model [12], as shown in Fig. 3. Authorizations revolve around a function *lvl* that maps entities and types of event actions to permission levels. Events that change the level function must define *lvl* for all entities and action types, both to allow multiple atomic changes at once and to prevent undesired results on executing concurrent changes. For an event e to be authorized, its creator subject $e.sbj$ must be authorized for a level greater or equal than its type of action $e.act$. Sending an event also requires a subject's group membership attribute to be *mbr*:*IN*. Administrative actions that change authorizations face additional restrictions: For an administrative event to be authorized, it must either grant authorization to its object for a level less or equal than the subject's level, or revoke authorization for a level that is less than the subject's level. Also, subjects can only perform actions on objects that have a lower level than themselves. As an exception, initial permissions and policies are at the group creator's discretion. Authorization is checked both before storing and before executing an event, on different bases: An event is only *stored* if it is authorized by the state derived from executing the *immutable set of its predecessors*, i.e., *immutable knowledge* of the entity. Hash linking enables receiving entities to detect and re-request missed events in a process called *backfilling*, and to verify that they received the complete predecessor set before deciding on authorization. An event is only *executed*, i.e., its encapsulated state change is only applied, if the event is authorized by the state derived from executing its immutable set of predecessors combined with the *grow-only set of concurrent events* currently known to the entity, i.e., its *mutable beliefs*. The process of finding an execution order and executing a partially-ordered set of events is called "state resolution" in Matrix. It extends the causal order of events stored in the chronicle to a total order via topological sorting, using a priority relation among events to break ties. Events are executed in topological order.

Matrix stores hash chronicles as hash-linked directed acyclic graphs [30], as shown in Fig. 4. A hash chronicle is based on recursive hashing of the causal history of its events [15]. The causal history of an event e in chronicle C is its downward closure $e^{\leq C} = \{x \in C \mid x \leq_C e\}$. Using a collision-resistant hash function h that concatenates its arguments, the recursive history hash $h_r(e)$ of an event e is the hash of the event and the recursive history hashes of its immediate causal predecessors $\hat{e} \in \max(e^{<C})$, formally $h_r(e) = h(e, \{h_r(\hat{e}) \mid \hat{e} \in \max(e^{<C})\})$. Together with digital signatures, recursive history hashing ensures authenticity and

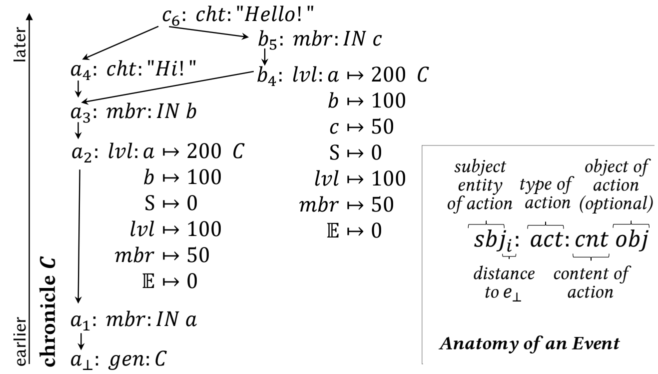


Figure 3: Example of Level- and Attribute-based Access Control in Matrix, in which attenuated authorizations flow from group creator a over b to c. Genesis event $a_\perp$ grants authorization to its creator a to send a first membership event a_1, declaring a to be `IN` the group, as well as a first permission level assignment a_2, assigning $a \mapsto 200$ and $b \mapsto 100$, so that while b is authorized both for membership changes (*mbr* $\mapsto 50$) and level assignments (*lvl* $\mapsto 100$), it cannot act against a. In a_3, a adds b to the group. Using its granted authorizations, b assigns $c \mapsto 50$ in b_4, otherwise repeating the previous `lvl`, and adds c in b_5. The chat message c_6 : *cht*:"*Hello*!" of c is indirectly authorized by a granting authorization to b.

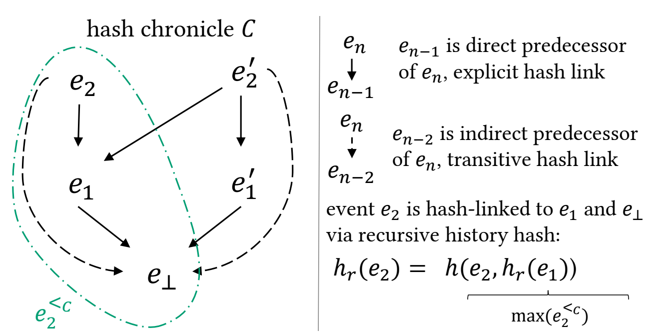


Figure 4: Example hash chronicle C. The causal order on events is divided into explicit hash links to direct predecessors and implicit, transitive links to other predecessors. The recursive history hash h_r for event e_2 is derived from e_2 and the recursive history hashes of its direct predecessors.

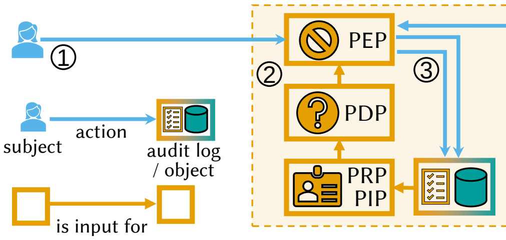
integrity of chronicle replication: for a given history hash, an entity can independently verify that it received the corresponding events completely and unaltered, regardless of the sender's correctness. In contrast to logical clocks in the crash-fault setting, recursive history hashing allows entities to create causally concurrent events where neither is the predecessor of the other. However, a Byzantine entity cannot create two different events with the same recursive history hash, due to collision resistance. Assuming a connected component of correct entities, replication based on recursive history hashing thereby ensures that every correct entity will eventually have the same chronicle state, i.e., will know about both concurrent events.

Hash chronicles and their derived data structures in Matrix are examples of Byzantine fault-tolerant conflict-free replicated data types (CRDTs, [13, 27]). CRDTs are a class of coordination-free replication algorithms that work on the premise that concurrent updates can be joined to a common state that is an advancement on previous states, without needing user interaction to resolve conflicts. Recent work has established that some CRDTs not only work in crash-fault environments, but also tolerate Byzantine faults [6, 15, 16]. Due to the autonomy of entities, CRDT-based decentralized systems may tolerate an arbitrary fraction of faulty entities, both in the crash-/omission fault model as well as in the Byzantine fault model, making them immune to Sybil attacks [17]. This is in contrast to Byzantine fault-tolerant systems that employ coordination, which typically require a form of majority of correct system entities. We say that chronicles store events in causal order, but detecting the typical notion of causality in the presence of Byzantine faults is impossible [23]. The typical notion of causality implies a total order on any set of events created by a single entity. In a similar vein to fork-join-causal consistency [21], Matrix only requires a partial order on the events created by one entity, as the predecessor set of an event is at the creator's discretion — it can only be ensured that the creator knew *at least* the predecessors, but not that they *only* knew the predecessors and did nothing concurrently. With this weakened causality definition, causality can be efficiently detected in the presence of Byzantine faults, under the assumption of collision-resistant hash functions [15].

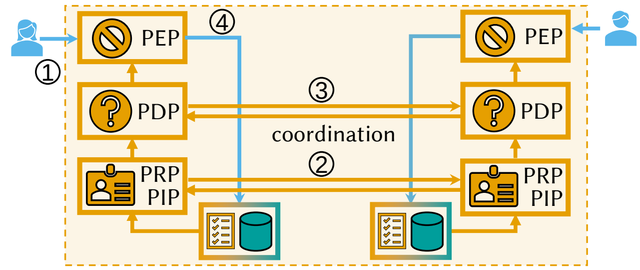
3 ECAC PROBLEM STATEMENT

In this section, we make the key challenge of ECAC explicit and provide a problem statement for which we present a solution in Section 4. We proceed by generalizing the decentralized access control approach of Matrix, contrasting it with conventional LAC.

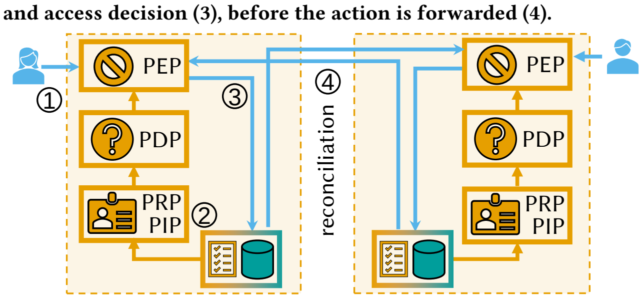
Access control architectures are characterized by the placement of their crucial components [4]: the Policy Enforcement Points (PEPs) that intercept actions, Policy Decision Points (PDPs) that issue the authorization decision, and the components that provide the basis for decision-making in terms of policies (Policy Retrieval Point, PRP) and attributes of subjects, actions, objects, or the environment (Policy Information Point, PIP) [28]. We use the following terminology. System entities create *events* by taking the conjunction of subject, i.e., themselves, action, and object. In general, we focus on actions that change state, in particular administrative actions that change access control state. Entities hand events to their access control enforcement mechanism, which implement an access control architecture via the execution monitoring method [26]: based on all previous events, the *execution monitor* decides whether the new event is authorized, and interrupts event processing if not. Authorized events are appended to the *audit log*, an ordered set of authorized events. The order of events in the audit log acts as logical timestamp of events. The execution of the audit log's events leads to the system's *app state*, which includes both the access control state as well as the state of non-administrative objects managed by the system. Events and their set of predecessors in the audit log are *immutable*, i.e., they cannot be changed after creation. *Finality* means that something cannot change after a given point in time. Immutability means finality after creation. A set is *grow-only* if its



(a) In LAC, actions of subjects are intercepted (1) by the Policy Enforcement Point (PEP), which implements a centralized execution monitor together with the Policy Information / Retrieval / Decision Point (PDP / PRP / PIP). Based on input by PRP & PIP, the PDP decides on the authorization of the action (2). Finally, to take effect on its object, the action is forwarded to the audit log (3).



(b) In CLAC, the PEP, PDP, and PRP/PIP are distributed, but still implement a centralized execution monitor via coordination. To decide authorization of an intercepted action (1), the PRP/PIPs coordinate to determine current policies and policy information (2), and the PDPs coordinate to reach consensus on the central order of events and access decision (3), before the action is forwarded (4).



(c) In ECAC, decisions are uncoordinated and autonomous, but implement a decentralized execution monitor via an eventually consistent audit log. To decide on the authorization of an intercepted action (1), PRP/PIP derive policies resp. policy information from the audit log for the PDP to decide to its best of knowledge and belief (2). The decision and basis for decision-making is recorded in the audit log, and the action is applied on the object (3). Audit logs are reconciled after the action took effect (4).

Figure 5: Comparison between Linearizable Access Control (LAC), Coordination-based Linearizable Access Control (CLAC), and Eventually Consistent Access Control (ECAC).

elements are immutable and cannot be removed, but new elements can be added. A partially-ordered set is *append-only* if it is grow-only and new elements can only be appended, i.e., a new element is either larger or concurrent to any other set element, but not smaller. Audit logs are append-only.

While audit logs are fundamental to ECAC, we also describe LAC and CLAC based on audit logs to highlight the differences in the conceptual models. The LAC, CLAC, and ECAC approaches differ in their access control architecture, as contrasted in Fig. 5, and especially in the way events are ordered in their audit log.

In LAC (Fig. 5a), there is only one instance of every access control architecture component, which represents both the centralized execution monitor ideal as well as its practical implementation based on a single, central entity. The LAC model is based on an audit log that totally orders its included events. We call this total order the central order, i.e., the order in which events become visible for the central entity. Events are executed in central order to eliminates concurrency, which indirectly resolves conflicts due to concurrent policy updates. The authorization decision for an event is based on the app state from executing all its predecessors. As the audit log is append-only, predecessor sets are immutable, and thereby, authorization decisions are immutable as well.

The CLAC approach is the usual way of distributing an access control architecture (Fig. 5b): The components of the central entity in LAC are instantiated once on every distributed entity, and coordination is required to ensure that all entities issue the same decisions based on the same policy information. Thus, coordination actually requires consensus on which event comes next, to replicate LAC's centrally-ordered audit log and to be able to deal with policy conflicts that can arise due to concurrent updates.

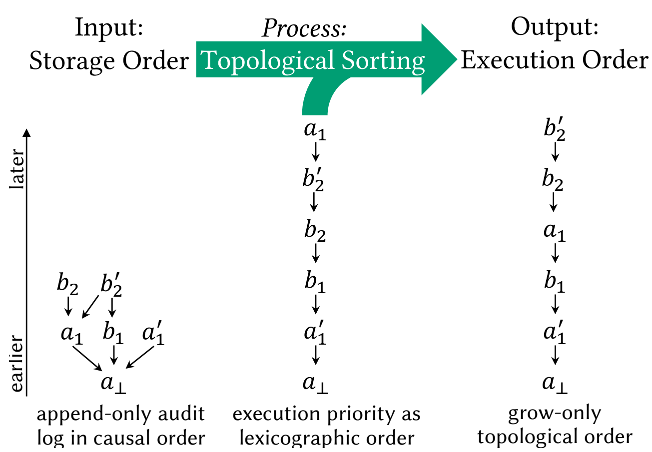


Figure 6: ECAC is based on three different orders of events: The causal storage order, which servers as input for topological sorting using the lexicographic prioritization order, to get the topological execution order as output. In the append-only causal order, the past of any included event is final, but events may be concurrent. The total lexicographic order, defined as part of the ECAC algorithm, reflects prioritization among events. Topological sorting resolves causal concurrency using the lexicographic order, leading to the topological order, a total, grow-only order. While the causal order is straightforward, the challenge of ECAC implementations is to define a lexicographic order and topological sorting so that concurrency conflicts are resolved in a way that neither compromises security nor application invariants.

While the placement of ECAC components (Fig. 5c) is the same as in CLAC, the challenge in terms of defining a conceptual model is the different communication pattern of components. Instead of coordinating with all other entities up-front to find consensus on the global knowledge, the ECAC approach is based on coordination-free access control decisions that are correct to the best of knowledge and belief locally available to the deciding entity. Thereby, the LAC conceptual model of execution monitoring that behaves as-if it was performed by a single, centralized entity is not applicable. Based on previous work on access control for weakly consistent databases like CRDTs [25, 33, 34], the problem at hand is to find a conceptual model in line with the properties of Byzantine CRDTs, like the Matrix approach described in Section 2.

In the ECAC model, the audit log is stored as a chronicle, as described in Section 2. In contrast to LAC, entities can create concurrent events that have the same set of causal predecessors, i.e., the audit log is only partially ordered. However, unlike LAC's central order, the partial causal order does not naturally resolve conflicts among concurrent events that affect each other, like authorization revocations. The idea of the Matrix approach that we generalize here (c.f. Fig. 6) is to have two orders instead of one central order: an append-only partial order for storage as the audit log, and a grow-only total order for execution. The order for execution is derived using the lexicographic order that defines execution priority among events as a refining sequence of comparison criteria, i.e., their lexicographic product. For example, authorization revocations can be prioritized before other events, or subjects with more permissions get to act before subjects with less permissions. Using the lexicographic order to resolve concurrency, entities extend the causal order via topological sorting, resulting in the topological order. While the topological order is a grow-only total order, it is not append-only: A new event can appear at any point in the order, as long as it does not violate causality. Causal and topological event orders result in two notions of authorization in ECAC: We call an event *causally authorized* if it is authorized by its causal predecessors, and *topologically authorized* if it is authorized by its topological predecessors. An event authorized by its causal predecessors may not be authorized by its topological predecessors: For example, if event b_1 is entity b adding entity c as member, but event a_1 is a concurrently revoking the authorization of b, then both events are causally authorized, but if a_1 comes first in topological order, b_1 becomes topologically unauthorized, and c is not a member.

We perform decentralized access control to protect the integrity of a replicated data structure in Byzantine environment. We look for a model of ECAC that characterizes its properties, both as set of guarantees weaker than LAC for applications to opt in, as well as necessary requirements for implementing algorithms. We are given an abstraction of Matrix, made up of three CRDT components [14]:

(1) the hash chronicle that stores a causally-ordered event set secured by recursive history hashing
(2) the topological event order derived by topological sorting
(3) the app state derived from topological event execution

To find are covering safety and liveness properties that make up a conceptual model for decentralized access control, but that do not weaken the availability, scalability, or resilience qualities of the CRDT components. Specifically, the model must allow to tolerate an arbitrary fraction of Byzantine faulty entities under an asynchronous timing assumption and provide availability under partition, from which it follows that the model must be implementable by coordination-free, autonomous decisions of correct entities. The ECAC model as a solution to decentralized access control based on the given components and constraints was found by consulting the eventual consistency properties of CRDTs [1], the theory of invariant confluence [5], and the CALM monotonicity theorem [9], and applying them to decentralized access control.

4 ECAC MODEL

We now present our model of eventually consistent access control, consisting of a set of safety and liveness properties that act as both guarantees to the application as well as necessary conditions on ECAC algorithms. We present our ECAC conceptual model in two parts: First, we cover the properties of the underlying eventually consistent data type, and then the properties of the eventually consistent authorizations based on the data type. The resulting eventually consistent access control is both rooted in the data type as well as encompasses the data type. While the properties of the data type are not concerned with access control directly, the data type ensures local availability of policies and policy information, and thereby is necessary for a coordination-free access decision. We introduce symbols and notations in-place, but list all in Table 1.

4.1 Eventually Consistent Data Type

Intuitively, eventual consistency is a consistency model that guarantees that over time, the state of correct system entities converges, to eventually reach a common, consistent state. Eventual consistency was originally defined in [29], consisting of the two properties that a) each update is eventually propagated to all system entities, and b) that non-commutative updates are executed in the same order by all system entities [1]. We assume that all functions executed by entities terminate, e.g., by using the total functional programming model [31] that achieves guaranteed termination by restriction to total functions and well-founded recursion. Because the execution order of commutative updates does not affect the resulting state by definition, those properties guarantee that eventually, every system entity will be in the same state. A weaker notion of eventual consistency, "if no new updates are made […], eventually all accesses will return the last updated value", was later popularized by Vogels [32], but a variation of the original definition gained traction under the name of strong eventual consistency as the consistency model for Conflict-Free Replicated Data Types (CRDTs) [1, 27]. Later, it was proven that the same definition originally conceived for crash fault environments also works in Byzantine environments, while needing different mechanisms to ensure its properties [16]. As recently suggested [1], we use the original definition of eventual consistency applied to CRDTs, consisting of the properties *eventual visibility* and *strong convergence* as follows:

Eventual Visibility (Liveness) An update visible for any correct entity is eventually visible for all correct entities.

Strong Convergence (Safety) Entities that see the same set of updates have equivalent state.

Note that strong convergence is based on the *set* of updates, i.e., applies regardless of differences in update visibility order between entities. In our context of ordered set of events, the properties that comprise eventual consistency become part of the ECAC model as *eventual event visibility* and *strong event set convergence*. We define $\mathbb{E}$ as set of all valid events, S as the set of correct system entities, and vis_s as the set of all events visible to entity s. We write C_s for the chronicle state of entity s, i.e., the causally ordered event set of s, and T_s for the topologically ordered event set of s. To execute a totally-ordered set of events T, we write $x(T) \subseteq T_s$, which returns an event set that describes the resulting app state.

Eventual Event Visibility An event visible for any correct entity is eventually visible for all correct entities.

$$\forall a, b \in S\colon e \in vis_a \implies \Diamond e \in vis_b \tag{1}$$

Strong Event Set Convergence Correct entities that see the same events have the same chronicle, topological order, and app state.

$$\forall a, b \in S\colon vis_a = vis_b \implies \tag{2}$$

$$C_a = C_b \land T_a = T_b \land x(T_a) = x(T_b) \tag{3}$$

We now define invariants that characterize the causally- and topologically-ordered event sets of correct entities at any given time. The causal event storage C_s of a correct entity $s \in S$ needs to satisfy *chronicality*, i.e., it must be downward-closed, partially-ordered set directed at $e_\perp$. The topological order T_s derived from C_s of a correct entity $s \in S$ must fulfill *topological totality*, i.e., be a total order. In addition, T_s must fulfill *causal consistency* and *lexicographic consistency*, i.e., follow the causal order given by the chronicle C_s, and when ambiguous, fall back to the lexicographic order L, as defined by the ECAC algorithm implementing the model. If an event e_1 is a causal predecessor of event e_2 in a partially-ordered event set E, we write $e_1 \leq_E e_2$. If two events e_1 and e_1' are unordered in E, e.g., because they are causally concurrent, we write $e_1 \parallel_E e_1'$. A partially ordered set E is a subset of E' if both the set and the order are subsets of each other, $E \subseteq E' \iff \forall e \in E, e' \in E'\colon e \in E' \land (e \leq_E e' \implies e \leq_E' e')$. We write $C_\cup = \bigcup_{s \in S} C_s$ for the global chronicle, i.e., the union of all local chronicles.

S	set of correct system entities
S^+	set of all system entities
$\mathbb{E}$	set of all events
$\mathbb{E}_z$	set of authorization events
$e.act$	type of action of event $e \in \mathbb{E}$
$e.obj$	optional object of event $e \in \mathbb{E}$
vis_s	set of events visible to $s \in S$
C_s	local chronicle, causal order of events stored by $s \in S$
$C_\cup$	global chronicle, the union of all local chronicles
T_s	topological order derived from C_s of $s \in S$
L	lexicographic priority order of events, defined by the algorithm implementing ECAC
$x(T)$	app state set as set of events resulting from executing totally-ordered event set T
$x^{<e}(E)$	app state set resulting from executing the topological order of event set E up to, but not including event e
$z(X, e)$	Boolean, $\top$ if event $e \in \mathbb{E}$ is authorized by app state X

Table 1: All symbols and notations used in the ECAC model.

Chronicality The local set C_s is always a chronicle, i.e., a partially-ordered event set, Eq. (4), that is downward-closed, Eq. (5), and directed at the global minimum $e_\bot$, Eq. (6).

$$\forall s \in S, \forall a, b \in C_s: \quad a \leq_{C_s} b \vee b \leq_{C_s} a \vee a \parallel_{C_s} b \tag{4}$$

$$\forall s \in S, \forall a \in C_s, \forall c \in C_\cup: \quad c \leq_{C_\cup} a \implies c \in C_s \tag{5}$$

$$\forall s \in S, \forall a \in C_s: \exists e_\bot \in C_s: \quad e_\bot \leq_{C_\cup} a \tag{6}$$

Topological Totality The topological order T_s is total.

$$\forall s \in S, \forall a, b \in T_s: a \leq_{T_s} b \vee b \leq_{T_s} a \tag{7}$$

Causal Consistency The topological order T_s preserves causality and contains the same events as C_s.

$$\forall s \in S, \forall e \in \mathbb{E}: C_s \subseteq T_s \wedge e \in C_s \iff e \in T_s \tag{8}$$

Lexicographic Consistency The topological order T_s orders causally concurrent events in accordance with the lexicographic order L.

$$\forall s \in S: \forall a, b \in C_s: a \parallel_{C_s} b \wedge a \leq_{T_s} b \implies a \leq_L b \tag{9}$$

We now define properties that characterize the evolution of the causally- and topologically-ordered events sets of correct entities over time. We are especially interested in monotonicity and immutability properties, as they hold independently of differences in event visibility orders [9]. We prime variables to denote a future state of the unprimed variable, e.g., local chronicle C_s evolves into C'_s. Monotonicity properties are safety properties that demand that if a future set of visible events vis'_s is greater or equal than the current set vis_s, then a derived value, like the chronicle, is also greater or equal than the current value. Immutability properties are the specialization where the derived value stays equal. As *vis* is grow-only, monotonically derived values represents certain knowledge that is not fallible in light of new information. For chronicles, we demand that the set of causal predecessors of any event must be immutable, which we formalize as *causal predecessor immutability*. In addition, we demand that the next chronicle state must include all previous events, which we formalize as *chronicle monotonicity*. We also require *topological monotonicity*, i.e., observing and ordering new events must not remove or change the order of old events.

Causal Predecessor Immutability

$$\forall s \in S, \forall e \in C_s: vis_s \subseteq vis'_s \implies e^{\leq_{C_s}} = e^{\leq_{C'_s}} \tag{10}$$

Chronicle Monotonicity The chronicle of a correct entity evolve monotonically, i.e., after observing new events, a correct entity inflates its local chronicle C_s to C'_s only by adding new events and their causal relations, while preserving old events and their causal predecessors.

$$\forall s \in S: vis_s \subseteq vis'_s \implies C_s \subseteq C'_s \tag{11}$$

Topological Monotonicity The topological order of a correct entity evolves monotonically, i.e., after observing new events, a correct entity inflates its topological order T_s to T'_s only by adding new events and new relations.

$$\forall s \in S: vis_s \subseteq vis'_s \implies T_s \subseteq T'_s \tag{12}$$

Note that topological monotonicity implies *topological predecessor monotonicity*, $e^{\leq_{T_s}} \subseteq e^{\leq_{T'_s}}$, but not immutability – in contrast to LAC's central order, new topological predecessors can always become visible.

4.2 Eventually Consistent Authorization

To complete the ECAC model, we now define properties regarding authorizations derived from and applied to the different event sets of the data type. Authorizations determine the actions that a subject is allowed to execute on which objects. Policies define the relation between policy information, like attributes of subjects and objects, and the subjects' authorizations. Authorizations therefore depend on both the specification of policies as well as the required policy information. We assume that both policies and policy information are encoded as attributes of subjects and objects. We speak of authorization events $\mathbb{E}_z \subseteq \mathbb{E}$ as the subset of events that potentially changes the set of authorizations, i.e., policy or policy information update events. Authorization events can grant an authorization for causally succeeding events, or revoke an authorization in causally succeeding as well as causally concurrent events.

The base requirement for eventually consistent access control is that authorizations are independent of the order in which authorization events become visible, whereby entities will not end up in a split-brain situation where convergence is impossible due to conflicting events by Byzantine entities. We formalize these requirements as follows: *eventual authorization visibility* means that an authorization event visible for one correct entity is eventually visible for all correct entities. *Strong authorization convergence* means that two entities that see the same authorization events conclude the same authorizations, and thereby perform the same authorization decisions. As the data type does not distinguish between authorization events and other events, those properties directly follow from eventual event visibility and strong event set convergence.

We now define invariants that characterize the role of causal and topological authorization in ECAC. An event is causally authorized if it is authorized by the app state resulting from executing its causal predecessors. From causal predecessor immutability follows *causal authorization immutability*: $vis_s \subseteq vis'_s \implies x^{<e}(C_s) = x^{<e}(C'_s) \implies z(x^{<e}, C_s)) = z(x^{<e}, C'_s)$. As soon as an event's predecessor set is known, the entity can issue an immutable causal authorization decision that is independent of the order in which events became visible, which is why we say that causal authorization is eventually consistent access control to the entity's best of knowledge. Correct entities only send causally authorized events, as they would not use an authorization they do not possess. While faulty entities can send causally unauthorized events, those events will never pass causal authorization at correct entities. We thereby demand that chronicle replication verifies causal authorization: correct entities must only store events in their chronicle that are causally authorized, which we formalize as *storage authorization*.

An event is topologically authorized if it is authorized by the app state resulting from executing its topological predecessors. Due to topological predecessor monotonicity, topological authorization decisions are mutable and never final, which is why we say that topological authorization is eventually consistent access control to the entity's best of beliefs. Specifically, authorization revocations are the cause of non-monotonicity of topological authorization decisions: a correct entity cannot state anything about the future topological authorization of an event currently deemed as topologically authorized or unauthorized, as learning about causally concurrent but topologically earlier authorization revocation events can always lead to changes in the topological authorization decision. On event execution, topologically unauthorized events must be ignored, but kept in case the become (re-)authorized later. Without revocations, we would end up with a monotonic protection system [7], for which a decentralized implementation could provide a strong, unconditional "topological authorization monotonicity" guarantee, akin to causal authorization immutability. However, in Byzantine environments, we need the possibility to revoke authorizations from Byzantine entities, e.g., if a member of a group chat posts spam messages and ought to get its group membership and messaging authorizations revoked. While correct entities only send events that are topologically authorized to the best of their belief, i.e., their topological order T_s, due to causally concurrent revocation events, we cannot demand that every event in any T_s must be topologically authorized. Instead, *execution authorization* prescribes that only topologically authorized events can have an effect on the app state set $x(T)$ resulting from executing T. In addition, *app state authorization* prescribes that all events in the app state set $x(T)$ must be topologically authorized.

We write $z(X, e)$ for the function that determines whether the event e is authorized based on the state set X, returning a truth value from the Boolean lattice $\mathcal{B} = \{\bot, \top\}$. Whether $z(X, e)$ verifies causal authorization or topological authorization depends on whether X is the result of executing causal or topological predecessors. The genesis event is the only event authorized by the empty set, $z(\emptyset, e_\bot) = \top$. To speak about different state sets for authorization, we define the shorthand notation for executing all events in the topological order of the partially-ordered set $E \subseteq T_s$ up to, but not including event e, $x^{<e}(E) = x(T_s \cap e^{<E})$.

Storage Authorization Every event e stored in a correct replica's state C_s is causally authorized.

$$\forall s \in S\colon e \in C_s \implies z(x^{<e}(C_s), e) \tag{13}$$

Execution Authorization Executed events are authorized by their topological past, i.e., removing topologically unauthorized events from T_s leads to the same app state set.

$$\forall s \in S\colon x(T_s) = x(\{e \in T_s \mid z(x^{<e}(T_s))\}) \tag{14}$$

App State Authorization Every event e included in the app state $x(T_s)$ is topologically authorized.

$$\forall s \in S\colon e \in x(T_s) \implies z(x^{<e}(T_s), e) \tag{15}$$

Up until now, all discussed properties only indirectly influence the exposed app state of the system. We established that app state must be topologically authorized, but that topological authorization is mutable due to concurrent authorization revocations. Now, we combine the eventually consistent data type and eventually consistent authorization to characterize the evolution of the exposed app state itself, namely the app state set $x(T_s)$ of correct entities over time. We require that app state must evolve monotonically if there are no revocations, and that revocations are the only source of non-monotonicity, which we formalize as *app state confluence*. Specifically, an authorization revocation event concurrent with an event that uses the revoked authorizations are in conflict, and lead to an order dependency where entities decide differently depending on the order in which they see the events. An entity which first sees the revocation event and then the usage event exposes monotonically-evolving app state, as the usage event is not executed. However, an entity which first sees the usage and then the revocation must roll back its execution result to an earlier event, which is not monotonic, but allowed under app state confluence. Still, due to eventual event visibility and event set convergence, entities eventually decide "as if they had known" of concurrent events, and the app state eventually converges.

An event describes the execution of an action of type *e.act* on an object *e.obj*. For example, an event could describe that a group chat administrator subject performs the action of changing the name of the group chat object. The events in the app state set resulting from the execution $x(T)$ of a totally ordered set of events T either describe the attributes of objects, i.e., have distinct $(e.act, e.obj)$ combinations, or have no object defined, i.e., the events that make up the communication history.

App State Confluence If an event e of the app state set is replaced by successor event e' with the same action and object in a later state set, the successor is either equal to or topologically larger than the predecessor, or the predecessor lost its topological authorization.

$$\forall s \in S, \forall e \in x(T_s), \forall e' \in x(T_s'), \tag{16}$$

$$e.act = e'.act, e.obj = e'.obj\colon \tag{17}$$

$$vis_s \subseteq vis_s' \implies e \leq_{T_s'} e' \vee \neg z(x^{<e}(T_s'), e) \tag{18}$$

4.3 Classification of ECAC Model Properties

Eventual event visibility is the liveness property of ECAC, other properties are safety properties. We now characterize the ECAC safety properties regarding invariant confluence and monotonicity. In essence, all properties characterize ECAC's independence of event visibility ordering. Monotonicity- and immutability-related properties describe entity state evolution *while* events become visible in arbitrary order, while the others describe entity state *after* arbitrary-order visibility.

Decentralized systems, in the sense of coordination-free distributed systems, cannot provide arbitrary services. They are limited to the concept of invariant confluence [5]: An invariant is confluent if when every entity ensures it locally based on its partial knowledge of events, the invariant also holds globally based on complete knowledge of all events. For example, an invariant that a set is grow-only is confluent, while an invariant that limits the maximum size of the set is not. As part of eventual consistency, strong convergence is an invariant-confluent property – otherwise, CRDTs would require coordination to ensure it. Strong convergence is ensured by every correct entity applying the same total function [31] on the set of updates that they see, and thereby, strong convergence holds globally. The same line of reasoning also applies to all ECAC properties: they do not rely on coordination, but only on total functions that derive entity state, like the current chronicle, from the unordered set of visible events.

In general, decentralized systems cannot hide the inherent nondeterminism of distributed systems in form of concurrency and reordering. The CALM theorem [9] ("Consistency as Logical Monotonicity") characterizes the subclass of invariant-confluent problems and algorithms whose outputs are also invariant to reordering of inputs as exactly the class of *monotonic* problems and algorithms.

A problem or algorithm is monotonic if when their input is greater or equal than another input, the output is also be greater or equal. Monotonicity alleviates nondeterminism induced by the system's distributed nature, whereby this subclass is especially suited for decentralized systems. Monotonicity is stronger than invariant confluence, and thereby, the monotonicity properties of ECAC are also invariant confluent. App state confluence is positioned between monotonicity and invariant confluence: It describes the condition under which app state is either monotonic, or only invariant confluent, i.e., can expose some form of 'time travel anomaly' depending on the order in which events become visible. We conclude that all ECAC safety properties are invariant confluent.

5 ASSESSMENT

5.1 ECAC Model Enforceability

We now discuss the enforceability of the ECAC model by a decentralized execution monitor. We said that entities that form the decentralized execution monitor intuitively do so by performing policy decisions to the best of knowledge and belief, i.e., in the conviction of their correctness but under the condition of fallibility due to incomplete knowledge on previous and concurrent events in the system. We substantiated that notion by decomposing it into a set of properties, i.e., the ECAC conceptual model of Section 4. In this set, eventual event visibility is the only liveness property, while all other properties are safety properties.

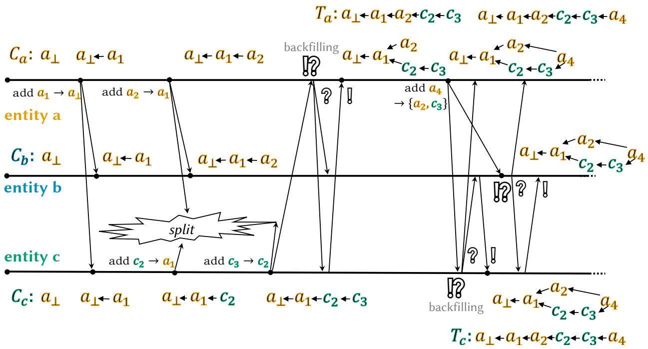
In his seminal work on the enforceability of security policies [26], Schneider states that security policies enforceable by execution monitors must be safety properties, and must be enforceable by terminating the subject to prevent the violation. As liveness properties are not enforceable by termination, they are out of scope for execution monitors and have to be ensured independently of the well-behavior of potentially Byzantine entities. For ECAC, eventual event visibility is ensured by backfilling, but only under the assumption of a connected component of correct entities. For a correct entity performing an ECAC algorithm, locally created events fulfill all safety properties by definition. For remote events from other, possibly incorrect entities, all safety properties are enforceable by terminating further processing of offending events, i.e., by denying them causal or topological authorization. While Schneider's work is concerned with centralized execution monitors, he already notes the idea of decentralized execution monitors: "the security policy for a distributed system might be specified by giving a separate security automaton for each system host. Then, each host would itself implement the [...] mechanisms for only the security automata concerning that host". For enforcement by a decentralized execution monitor, we need to combine the work of Schneider and the work of Bailis et al. on invariant confluence [5]: To be enforceable by a decentralized execution monitor, a safety property must also be invariant confluent. As all ECAC safety properties are also invariant confluent, we conclude that ECAC safety properties are enforceable by decentralized execution monitors.

5.2 Partition, Equivocation, and Backdating

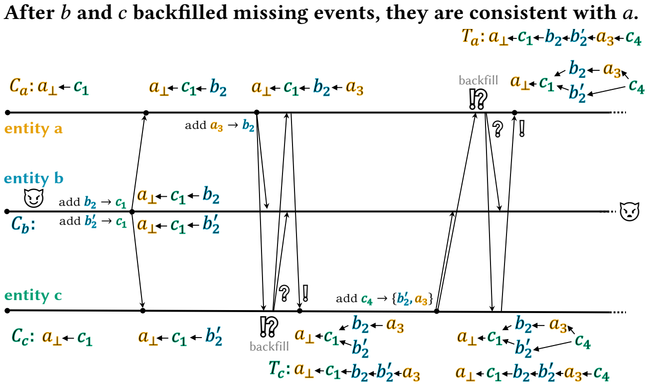
We now show the behavior of ECAC in critical scenarios, namely partition, equivocation, and backdating, where eventually consistent access control behaves differently compared to centralized access control. We assume a system $S^+ = \{a, b, c\}$ of three entities, entity b may exhibit Byzantine behavior. While coordination-based approaches are still viable with two correct and one faulty entity, this simplification is for illustration purposes: due to a's and b's autonomous decisions to their best of knowledge and belief, the assessment would be unchanged by any number of additional Byzantine entities. The key point of these scenarios depicted in Fig. 7 is to show how the causal order of events with its immutable predecessors enables immutable causal authorization under partition and Byzantine misbehavior, while the topological order enables non-monotonic revocation of authorizations to still be strongly convergent. As practical example, we take an electronic health record (EHR) stored as chronicle, featuring a patient a, their health insurer b, and their general practitioner c as entities. The EHR is replicated among all entities, to ensure availability of reads and writes without internet access. The EHR consists of medical findings and therapeutic schedules by practitioners, associated cost coverage declarations of insurers, the patient's master data, as well as the EHR authorizations, all described as events.

For the partition scenario displayed in Fig. 7a, there is a temporary partition between c and $\{a, b\}$, leading to events $\{a_2, c_2, c_3\}$ not reaching every entity. In the LAC model, entities would be unable to verify the authorization of affected events and reject them, i.e., be unavailable under partition. In the ECAC model, entities that received the events accept the events as causally authorized, and store them. After the partition is over, all entities eventually notice the lost events due to the references to unknown predecessors in newly-incoming events. To decide causal authorization, entities try to gather lost events by backfilling, and eventually succeed if a correct entity has seen them. For the EHR access control example, we say that event c_2 is a master data update of the patient by their general practitioner, and c_3 is a new medical finding. Event a_2 revokes the authorization of practitioner c to update master data, but still allows to add new findings. As a_2 and c_2 are sent concurrently during the partition, entity c uses its authorization to create c_2, as it has not yet heard of the revocation. Due to causal authorization immutability and eventual event visibility, entity a eventually decides c_2 as causally authorized. We assume that the lexicographic order L prioritizes authorization events, i.e., $a_2 \leq_L c_2$. Then the revocation a_2 is topologically earlier, and revokes topological authorization of c_2. Thereby, the EHR at c exhibits non-monotonic behavior: while c executed c_2 during the partition and updated the master data, it will ignore c_2 and restore the old master data as soon as it learned about a_2, i.e., $a_2 \in T_c \Rightarrow \neg z(x^{<c_2}(T_c), c_2) \Rightarrow c_2 \notin x(T_c)$. This scenario shows the effects of app state confluence, which allows non-monotonicity only if an event loses topological authorization.

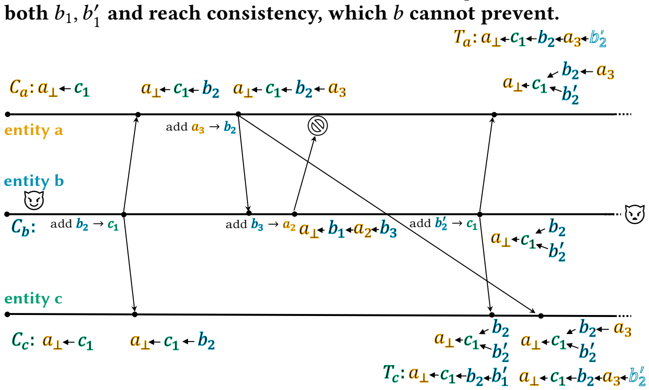
For the equivocation scenario in Fig. 7b, we assume that Byzantine entity b tries to create an inconsistency between a and c by sending them different but concurrent events, i.e., b performs equivocation using events b_2, b'_2. In the LAC model, this scenario does not exist: all events are totally ordered in the order in which they become visible for the (logical) central entity, whereby there are no concurrent events. While a and c temporarily only know of one of the equivocation events, due to the same mechanics that came into play during the network partition scenario of Fig. 7a, both a and c will eventually see both b_2, b'_2. Thereby, they eventually end up with



(a) Due to a network partition, entity a misses event c_2, b misses $\{c_2, c_3\}$, and c misses a_2. When a sees c_3, a cannot verify causal authorization due to the missing c_2. Entity a starts backfilling, eventually receives $\{c_2, c_3\}$, verifies causal authorization, and adds them to C_a. After b and c backfilled missing events, they are consistent with a.



(b) Byzantine entity b performs equivocation: It creates concurrent events b_1, b_1' and sends b_1 to a and b_1' to c. As both b_1, b_1' are causally authorized, b managed to create an inconsistency between a, c. Eventually, a, c exchange new events that refer to b_1, b_1'. They backfill, see both b_1, b_1' and reach consistency, which b cannot prevent.



(c) Byzantine entity b sends event b_1 that a finds offensive, creating a_2 that revokes the authorization of b to send further events. To evade revocation, b manipulates its local execution monitor to act as if it still had the necessary authorization for b_3, which fails causal authorization at a's local execution monitor. Then, b pretends to have not seen a_2, and sends a backdated event b_1' concurrent to a_2, passing causal authorization. Entities a, c cannot distinguish whether b_1' was created before or after b saw a_2. However, revocations may act against causally concurrent events on execution: assuming $a_2 \leq_L b_1'$, event b_1' does not pass topological authorization, and is not executed.

Figure 7: Partition, Equivocation, and Backdating Scenarios

a consistent chronicle, topological order, and state set, and eventual event visibility and strong event set convergence are fulfilled. For the EHR access control example, assume that events b_2, b_2' are conflicting cost coverage declarations: b_2 declares cost coverage for treatment schedule c_1 to the patient a and grants a with the authorization to accept the treatment, while b_2' declares that the cost of treatment schedule c_1 are not covered, and treatment access of a is revoked. At first, a and c will report a different cost coverage status, and the practitioner would deny treatment to the patient. Based on the cost coverage b_2, a accepts the treatment in a_3. The eventual event visibility and strong event set convergence properties of the system ensure that eventually, a and c have causal authorization and certain knowledge and proof that insurer b concurrently sent both b_2 and b_2', and can hold b accountable for its equivocation. The concurrent changes will be executed in accordance with the lexicographic order, i.e., the resulting cost coverage depends on the lexicographically larger event of b_2, b_2' for both $\{a, c\}$. Assuming $b_2' \leq_L b_2$, the cost coverage grant is executed before the revocation, and practitioner c records treatment execution and results in c_4.

For the backdating scenario in Fig. 7c, we assume that Byzantine entity b tries to evade an authorization revocation done by a in a_3 by manipulating its local execution monitor. In the LAC model, this scenario does not exist: as all events are executed in the central order, anything sent by b after the central entity executed a_3 is subject to the revocation described by a_3. For the EHR access control example, we assume that b_2 is a positive cost coverage declaration for treatment schedule $a_\perp$, but includes a patient's contribution. Patient a is discontent with the contribution and revokes further EHR write access of insurer b, intending to switch to another insurer. Insurer b wants to send a negative cost coverage declaration now, trying to evade the revocation. Insurer b first manipulates its local execution monitor to ignore the revocation in a_3 to send the negative cost coverage b_3 anyway, which fails at the execution monitor of a as causally unauthorized. In a second attempt, b dates back negative cost coverage b_2', listing only c_1 as predecessor, thereby stating $b_2' \parallel b_2$. As b_2' is causally authorized, correct entities $\{a, c\}$ cannot differentiate whether b_2' was created after b already knew about a_3, or whether b_2' just happened to be in transit for a very long time, and must accept both as causally authorized. This exemplifies that entities can claim any causal predecessors with impunity, as long as the event is causally authorized — akin to the fork-join model of causality [22]. Assuming that negative cost coverage declarations are executed before positive declarations, i.e., $b_2' \leq_L b_2$, practitioner c would perform treatment under the positive declaration despite knowing both concurrent declarations. Backdating underlines the importance of the prioritization rules of causally concurrent events through their lexicographic order: assuming that $a_3 \leq_L b_2'$, event b_2' is never executed by $\{a, b\}$, as they have seen a_3 first.

Overall, the scenarios show the effect of replacing up-front coordination with subsequent reconciliation: The overall system exhibits high resilience, i.e., continues to provide availability under detrimental circumstances, and can tolerate Byzantine behavior. In essence, the price to pay for the beneficial properties of eventually consistent access control is the mutability of the topological order, i.e., the execution of events to the entity's best of beliefs on the overall set of events, that takes effect on executing inherently non-monotonic actions like authorization revocations.

5.3 ECAC Implementation Simplicity

For a comprehensible ECAC implementability demonstration open to scrutiny, we now describe a simple ECAC algorithm that makes an abstraction from the complex ECAC implementation of Matrix (c.f. Section 2). The algorithm is based on previous work on abstracting Matrix [14] as a composition of CRDTs, but adapted to match the terms of the ECAC model. While walking through the algorithm, we show how it fulfills all ECAC model properties defined in Section 4.

In Algorithm 1, the data type foundation of Matrix is described as a CRDT for hash chronicles. We assume an unlimited number of Byzantine entities that participate in the CRDT, but also assume that correct entities form a connected component, i.e., cannot be stopped from communicating by Byzantine entities. CRDTs can be categorized as state-based and operation-based CRDTs [27]. The hash chronicle CRDT falls into the class of delta-state CRDTs [2, 3], whose state is a join-semilattice that converges by exchanging deltas. Deltas are also elements of the join-semilattice, and applied by joining them with an entity's current state. Here, the join-semilattice is defined as the set of all sets of valid events with set union as join operation, $(\mathcal{P}(\mathbb{E}), \cup)$. Specifically, the local state of a hash chronicle is the entity's set of visible events *vis*. The local state is initialized with the pre-shared genesis event $e_\perp$, as an anchor for access control that authorizes the chronicle creator to create the first membership and level assignment events. The query function $<_C$ determines whether an event e_1 is causally earlier than another event e_2 by looking for a chain of recursive hash links from e_2 to e_1, based on the set of recursive hashes $e.pre$ of the direct causal predecessors an event e. The query function $C(vis)$ derives the entity's current chronicle C from *vis* by traversing the set of recursive predecessor hashes $e.pre$ in reverse order, i.e., going up from $e_\perp$. The result is the largest downward-closed subset directed at $e_\perp$, which fulfills the *chronicality* property. The query also verifies the causal authorization of any event before adding it to the resulting chronicle, whereby *storage authorization* is fulfilled as well. The mutate function $add(e)$ creates a δ-update from new event e by assigning e's set of direct predecessor hashes $e.pre$ with the set of maximal elements of the entity's current chronicle as timestamp. On calling the $add(e)$ function, the entity joins *vis* with δ to apply the update, and gossips δ to all other entities. On receiving a δ, entities verify that only the genesis event has no predecessors, and add it to their visible event set. As events are immutable and only added to *vis*, *vis* is grow-only. The recursive hash links in $e.pre$ that unambiguously define the causal history of any event e ensure *causal predecessor immutability*. As *vis* is grow-only and $C(vis)$ query result is a subset *vis*, *chronicle monotonicity* is ensured. Periodically, entities gossip their maximal chronicle events, and backfill by requesting events for which they have the recursive hash, but not the event itself. Gossiping and backfilling ensures *eventual event visibility* under the assumption of a connected component of correct entities. As $C(vis)$ as well as $T(E)$ and $x(T)$ in Algorithm 2 are total functions that, by definition, terminate and deterministically return the same output when given the same input, *strong event set convergence* is fulfilled. Due to eventual event visibility and strong event set convergence in Byzantine environment, the algorithms represent a Byzantine-tolerant CRDT, which was already shown in prior work [13, 14].

Let us now discuss the functions of Algorithm 2 that build on the hash chronicle CRDT of Algorithm 1. The topological ordering function $T(E)$ performs topological sorting on the chronicle subset $E \subseteq C$. It takes the set of causally earliest, yet unsorted events first, fulfilling *causal consistency*, to then take the lexicographically earliest event, fulfilling *lexicographic consistency*. The resulting event is used as next event in the topological order, which ensures *topological totality*. Due to chronicle monotonicity and as $T(E)$ operates on chronicle subsets and only extends them with additional relations to form a total order, *topological monotonicity* is ensured. The event execution function $x(T)$ takes totally-ordered chronicle subset of events, i.e., a result of function $T(E)$. The function walks through the total order and executes events in order. It ignores topologically unauthorized events, which ensures *execution authorization*. Topologically authorized events are added to the resulting app state X, which ensures *app state authorization*. If a topologically later event assigns an attribute to an object, it replaces the previous event for that attribute. Due to topological order execution, the only way that a topologically later event is replaced by a topologically earlier event when $T \subseteq T'$ is that the later event is ignored as topologically unauthorized, which ensures *app state confluence*.

We finally discuss the lexicographic order $<_{L(X)}$ and the authorization function $z(X)$ of this algorithm. The lexicographic order $L(X)$ defined by $<_L (X)$ orders two events e_1, e_2 based on an app state set X as returned by $x(T)$. As first criterion, the lexicographic order prioritizes authorization events, i.e., e_1 is before e_2 if e_1 is an authorization event but e_2 is not. This criterion ensures that authorization revocations are executed before concurrent non-authorization events, in order to prevent revocation evasion. If both events are either authorization events or non-authorization events, the next criterion look at the permission level of the subjects of e_1 and e_2. Events of higher-level subjects are executed first, to ensure that events, especially authorization revocations, by higher-level subjects are executed before any events of lower-level subjects. The final criterion is based on the recursive hash value of the events, the event with the lower hash value is topologically earlier. The hash comparison ensures that the lexicographic order is total even in the presence of Byzantine entities: the hash function's collision resistance ensures that this criterion always orders any two events. However, it is only the last criterion, as a Byzantine entity can easily create an event with a smaller recursive hash than the recursive hash of any given event. On every bit flip in the Byzantine event, there is a 50 % chance for the Byzantine entity that the hash is smaller than the average other event, i.e., a random sequence of $\{0, 1\}$. The authorization function $z(X, e)$ decides whether event e is authorized given the app state set X, based on the Level- and Attribute-based Access Control model [12] employed by Matrix. Event authorization is decided by four criteria: authorization for the group, the action, the object, and level, which all must be fulfilled to be authorized. The event is authorized for the communication group if there was a previous action of type `mbr` that declared the subject $e.sbj$ to be `IN` the communication group. The event is authorized for its action if the action type $e.act$ is assigned with a level less or equal than the event's subject $e.sbj$. The event is authorized for its object if either has no object, or the object is the subject, or

Algorithm 1 Delta-state hash chronicle (run by each entity $s \in S$). Given are the universe of events $\mathbb{E}$, the genesis event $e_\bot$, and the recursive history hash function h_{rh}.

state visible event set $vis \subseteq \mathbb{E}$
initial $vis \leftarrow \{e_\bot\}$
query $<_C (e_1, e_2 \in C) : p \in \{\bot, \top\}$
  $p \leftarrow \exists e \in C\colon h_{rh}(e) \in e_2.pre \wedge e_1 <_C e$
query $C\ (vis) : C \subseteq vis$
  $C \leftarrow \{e_\bot\}$ ▹ largest downward-closed subset directed at $e_\bot$
  repeat
    $C^\dagger \leftarrow \{e \in vis \setminus C \mid e.pre \subseteq \{h_{rh}(e) \mid e \in C\}\}$
    $C^\dagger \leftarrow \{e \in C^\dagger \mid z(x^{<e}(C \cup \{e\})\}$
    $C \leftarrow C \cup C^\dagger$
  until $C^\dagger = \emptyset$
mutate $add\ (e \in \mathbb{E}) : \delta \subseteq \mathbb{E}$
  $e.pre \leftarrow \{h_{rh}(\hat{e}) \mid \hat{e} \in \max_C(C(vis))\}$
  $\delta \leftarrow \{e\}$
on operation($add(e)$)
  $\delta = add(e)$
  $vis' \leftarrow vis \cup \delta$
  $gossip(\delta)$
on receive($\delta \subseteq \mathbb{E}$)
  if $\forall e \in \delta\colon e.pre \neq \emptyset \vee e = e_\bot$ **then**
    $vis' \leftarrow vis \cup \delta$
periodically
  $gossip(\max_C(C(vis)))$
  $request(\bigcup\{e.pre \mid e \in vis\} \setminus \{h_{rh}(e) \mid e \in vis\})$

the object is assigned with a strictly lesser permission level than the subject. Thereby, equally-leveled subjects cannot remove each other from the communication group, which avoids revocation cycles. Finally, if the event assigns levels, it is only authorized if it does not raise the level of any entity or action type o above the level of the event subject $e.sbj$, and does not lower the level of something above the subject's level.

6 CONCLUSION

In this paper, we defined the ECAC model for eventually consistent access control. Leveraging the concepts of monotonicity and invariant confluence from the field of replicated database systems, we defined a set of security properties for access control based on a form of partially-ordered event "logbook" conflict-free replicated data types. While permission revocations show non-monotonicity in general, our analysis shows that revocations can be invariant confluent. The explication of the properties show that applications have to cope with some form of "time travel anomaly", which we describe as providing access control to the best of knowledge and belief. Thereby, this paper provides the necessary foundation to formal security verification of the access control aspects of the Matrix specification for decentralized group communication systems. The semantics and security notions of eventually consistent access control are highly relevant for practical, geo-distributed, resilient systems that can cope with arbitrary network and process faults. In contrast to centralized models, an ECAC access decision is immediate and optimally fault tolerant even in a Byzantine environment.

Algorithm 2 Topological Order T, lexicographic order L, execution function $x(T)$, Level- and Attribute-based Authorization Function $z(X, e)$

function $T\ (E \subseteq C) : T \supseteq E$
  $T \leftarrow \emptyset$ ▹ topological ordering of chronicle subset E
  for $n = 0$ to $|E|$ **do**
    $E_{\min} \leftarrow \min_C(E)$ ▹ set of causally smallest events
    $X_n \leftarrow x(T)$
    $e_n \leftarrow \min_{L(X_n)}(E_{\min})$ ▹ lexicograpically smallest event
    $T_n \leftarrow e_n$ ▹ assign next event in topological order
    $E \leftarrow E \setminus \{e\}$
function $x\ (T) : X \subseteq T$
  $X \leftarrow \emptyset$ ▹ app state set of executing totally-ordered T
  for $n = 0$ to $|T|$ **do**
    $e \leftarrow T_n$ ▹ next event to execute in topological order
    if $z(X, e)$ **then** ▹ ignore if topologically unauthorized
      if $e.obj \neq \bot$ **then** ▹ replace previous event
        $e_x \leftarrow e_x \in X \mid e_x.act = e.act \wedge e_x.obj = e.obj$
        $X \leftarrow (X \setminus \{e_x\}) \cup \{e\}$
      else
        $X \leftarrow X \cup \{e\}$
function $<_{L(X)}\ (e_1, e_2 \in \mathbb{E}) : p \in \{\bot, \top\}$
  ▹ whether $e_1 <_{L(X)} e_2$ in lexicographic order $L(X)$
  $lvl \leftarrow x.cnt \mid x \in X\colon x.act = \texttt{lvl}$
  $p \leftarrow e_1 \in \mathbb{E}_z \wedge \neg e_2 \notin \mathbb{E}_z$ ▹ prioritize authorization events
  if $e_1 \in \mathbb{E}_z \wedge e_2 \in \mathbb{E}_z$ **then**
    $p \leftarrow lvl(e_1.sbj) < lvl(e_2.sbj))$
    ▹ if both / neither is authorization, order by level
    if $lvl(e_1.sbj) = lvl(e_2.sbj)$ **then**
      $p \leftarrow h_{rh}(e_1) > h_{rh}(e_2)$
      ▹ if equal subject level, order by recursive hash
function $z\ (X \in \mathcal{P}(\mathbb{E}), e \in \mathbb{E}) : z \in \{\bot, \top\}$
  ▹ whether e is authorized by app state set X
  $lvl \leftarrow x.cnt \mid x \in X\colon x.act = \texttt{lvl}$
  $m_{sbj} \leftarrow x.cnt \mid x \in X\colon x.act = \texttt{mbr} \wedge x.obj = e.sbj$
  $group_z \leftarrow m_{sbj} = IN$
  $action_z \leftarrow lvl(e.act) \leq lvl(e.sbj)$
  $object_z \leftarrow e.obj = \bot \vee e.obj = e.sbj \vee lvl(e.obj) < lvl(e.sbj)$
  $level\_z = \top$
  if $e.act = \texttt{lvl}$ **then** ▹ cap new levels by subject level
    $lvl' \leftarrow e.cnt$
    $level\_z \leftarrow \forall o \in lvl'$:

$$lvl'(o) \geq lvl(o) \Rightarrow lvl'(o) \leq lvl(e.sbj)$$

$$lvl'(o) \leq lvl(o) \Rightarrow lvl(o) < lvl(e.sbj)$$

  $z \leftarrow group_z \wedge action_z \wedge object_z \wedge level_z$

However, their systemic difference outlined in this paper needs to be taken into account.

ACKNOWLEDGMENTS

This work was funded by the Helmholtz Pilot Program Core Informatics. We like to thank Marc Shapiro for his suggestion to study application invariants above of CRDTs, and Matthew Weidner for his thoughts on CRDT composition.

REFERENCES

[1] Paulo Sérgio Almeida. 2023. Approaches to Conflict-free Replicated Data Types. https://doi.org/10.48550/arXiv.2310.18220 arXiv:2310.18220 [cs]

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-Based CRDTs by Delta-Mutation. In *Networked Systems*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Vol. 9466. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-26850-7_5

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta State Replicated Data Types. *J. Parallel and Distrib. Comput.* 111 (Jan. 2018). https://doi.org/10.1016/j.jpdc.2017.08.003

[4] Ross Anderson. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems* (third edition ed.). John Wiley & Sons, Indianapolis. https://doi.org/10.1002/9781119644682

[5] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014). https://doi.org/10.14778/2735508.2735509

[6] Davide Frey, Lucie Guillou, Michel Raynal, and François Taïani. 2023. Process-Commutative Distributed Objects: From Cryptocurrencies to Byzantine-fault-tolerant CRDTs. https://doi.org/10.48550/arXiv.2311.13936 arXiv:2311.13936 [cs]

[7] Michael A. Harrison and Walter L. Ruzzo. 1978. Monotonic Protection Systems. In *Foundations of Secure Computation*. Atlanta, Georgia, USA.

[8] Alex Heath. 2021. Locked out and totally down: Facebook's scramble to fix a massive outage. https://www.theverge.com/2021/10/4/22709575/facebook-outage-instagram-whatsapp

[9] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM: When Distributed Consistency Is Easy. *Commun. ACM* 63, 9 (Aug. 2020). https://doi.org/10.1145/3369736

[10] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

[11] Matthew Hodgson. 2023. Matrix 2.0. https://archive.fosdem.org/2023/schedule/event/matrix20/

[12] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. 2020. Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT '20)*. ACM, New York. https://doi.org/10.1145/3381991.3395399

[13] Florian Jacob, Carolin Beer, Norbert Henze, and Hannes Hartenstein. 2021. Analysis of the Matrix Event Graph Replicated Data Type. *IEEE Access* 9 (2021), 28317–28333. https://doi.org/10.1109/ACCESS.2021.3058576

[14] Florian Jacob and Hannes Hartenstein. 2023. On Extend-Only Directed Posets and Derived Byzantine-Tolerant Replicated Data Types. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23)*. ACM, New York. https://doi.org/10.1145/3578358.3591333

[15] Florian Jacob and Hannes Hartenstein. 2024. Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Athens Greece, 37–43. https://doi.org/10.1145/3642976.3653034

[16] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data* (Rennes, France) *(PaPoC '22)*. ACM, New York. https://doi.org/10.1145/3517209.3524042

[17] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *arXiv:2012.00472 [cs]* (Dec. 2020). arXiv:2012.00472 [cs]

[18] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[19] Steve Loynes. 2024. *Element Signs Partnership Deal with Tele2 for Secure Collaboration.* https://element.io/blog/element-signs-partnership-deal-with-tele2-for-secure-collaboration/

[20] Steve Loynes. 2024. *UNICC Selects Element for Secure Communications.* https://element.io/blog/unicc-selects-element-for-secure-communications/

[21] Prince Mahajan. 2012. *Highly Available Storage with Minimal Trust.* Ph. D. Dissertation. University of Texas at Austin. http://hdl.handle.net/2152/16320

[22] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2011. *Consistency, Availability, and Convergence.* Technical Report TR-11-22. Computer Science Department, University of Texas at Austin. http://www.cs.utexas.edu/users/dahlin/papers/cac-tr.pdf

[23] Anshuman Misra and Ajay D. Kshemkalyani. 2022. Detecting Causality in the Presence of Byzantine Processes: There Is No Holy Grail. In *2022 IEEE 21st International Symposium on Network Computing and Applications (NCA)*, Vol. 21. 73–80. https://doi.org/10.1109/NCA57778.2022.10013644

[24] K. Mori. 1993. Autonomous Decentralized Systems: Concept, Data Field Architecture and Future Trends. In *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems.* https://doi.org/10.1109/ISADS.1993.262725

[25] Pierre-Antoine Rault, Claudia-Lavinia Ignat, and Olivier Perrin. 2023. Access Control Based on CRDTs for Collaborative Distributed Applications. In *The International Symposium on Intelligent and Trustworthy Computing, Communications, and Networking (ITCCN-2023), in Conjunction with the 22nd IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom-2023).*

[26] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Transactions on Information and System Security* 3, 1 (Feb. 2000), 30–50. https://doi.org/10.1145/353323.353382

[27] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer, Heidelberg. https://doi.org/10.1007/978-3-642-24550-3_29

[28] David Spence, George Gross, Cees de Laat, Stephen Farrell, Leon HM Gommans, Pat R. Calhoun, Matt Holdrege, Betty W. de Bruijn, and John Vollbrecht. 2000. *AAA Authorization Framework.* Request for Comments RFC 2904. Internet Engineering Task Force. https://doi.org/10.17487/RFC2904

[29] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems.* 140–149. https://doi.org/10.1109/PDIS.1994.331722

[30] The Matrix.org Foundation CIC. 2024. *Matrix Specification v1.10.* Technical Report. https://spec.matrix.org/v1.10/

[31] D. Turner. 2004. Total Functional Programming. *JUCS - Journal of Universal Computer Science* 10, 7 (July 2004), 751–768. https://doi.org/10.3217/jucs-010-07-0751

[32] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[33] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2016. Access Control for Weakly Consistent Replicated Information Systems. In *Security and Trust Management (Lecture Notes in Computer Science)*, Gilles Barthe, Evangelos Markatos, and Pierangela Samarati (Eds.). Springer International Publishing, Cham, 82–97. https://doi.org/10.1007/978-3-319-46598-2_6

[34] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. 2021. On the Impossibility of Confidentiality, Integrity and Accessibility in Highly-Available File Systems. In *Networked Systems (Lecture Notes in Computer Science)*, Karima Echihabi and Roland Meyer (Eds.). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-91014-3_1