Towards System-Oriented Formal Verification of Local-First Access Control

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

Johanna Stuber
johanna.stuber@student.kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

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

Abstract

Conflict-free replicated data types (CRDTs) and the local-first concept are increasingly employed not only in small-scale collaboration systems among few users who trust each other, but also in large-scale systems, like Matrix for instant messaging and Keyhive for collaborative documents. Since mutual trust is no longer warranted, these systems require Byzantine fault tolerance and fine-grained access control. As of today, Matrix and Keyhive pair an informal specification with an unverified reference implementation. In this work, we follow a bottom-up approach towards constructing formally verified authorization algorithms for Byzantine fault-tolerant local-first systems. As intermediate target for formal verification, we contribute semantics and invariants of a replicated data type for managing simplified collaboration groups, based on capabilities for access control and hash chronicles for replication. To enable future integration into local-first systems like Matrix and Keyhive, we strive for accessibility to system engineers by using the Rust programming language for formal specification, verification, and implementation, enabled by the Verus framework using the Z3 theorem prover at zero runtime cost. We report on our experience and preliminary results following this approach, and discuss next steps towards scaling up access control expressiveness. Whether this approach can be scaled up to the complexity of real-world local-first access control systems like Matrix or Keyhive remains future work, but our findings demonstrate the potential of system-oriented formal verification with Verus.

CCS Concepts: • **Computing methodologies** → *Distributed algorithms*; • **Information systems** → *Data structures*; • **Security and privacy** → **Distributed systems security**; • **Software and its engineering** → **Access protection**.

Keywords: Authorization, Local-First Access Control, Local-First Systems, Conflict-Free Replicated Data Types, Byzantine Fault Tolerance, Matrix, Keyhive

1 Introduction

Conflict-Free Replicated Data Types (CRDTs, [1, 30]) are a common solution to concurrency conflicts in local-first systems, i.e., decentralized systems designed for low latency and availability under partition [17, 21]. CRDT-based systems do not attempt to conceal their inherent concurrency, but instead demand concurrent thinking from their designers. In the CRDT research community, the benefits of formal methods for CRDT design to ensure consistency and application invariants are well-established, especially in Byzantine contexts [6, 10], to counter the highly-concurrent and nondeterministic nature of distributed systems. However, Basin et al. recently called attention to the gap between formal methods theory and distributed systems practice relying on informal specifications and unverified reference implementations [5]. An example for this gap is the Matrix decentralized system for instant messaging and group chats [32], widely used in security-critical settings ranging from the French public sector and the German healthcare system to the United Nations and NATO [8, 9]. A recent update of the Matrix specification dealt with an algorithm violating a security invariant [7, 33], which could have been caught early by integrating formal methods into the standardization process.

Large-scale local-first systems like Matrix cannot rely on mutual trust but face arbitrary malevolence of participants, necessitating Byzantine fault tolerance and fine-grained access control. We therefore study formal verification of local-first access control in an asynchronous system with Byzantine faults. We strive for accessibility to system engineers by using Verus [22, 23], a recent verification framework that pairs the popular high-performance programming language Rust with the Z3 theorem prover at zero runtime cost.

Instead of attempting to verify the authorization algorithms of Matrix or Keyhive directly, we follow a bottom-up approach to construct simple but formally verified authorization algorithms for local-first systems, in order to gradually extend complexity and expressiveness in the future. We contribute a semantics specification of a replicated data type for capability-based access control in collaboration groups, along with a simplified but verified algorithm for local-first authorization. We report on our experience with Verus to unify specification, implementation, and verification in a single Rust source file, and publish all source code under a free license [31].

2 Local-First Systems and Related Work

Inspired by local-first software [19, 21], we use the term *local-first* to describe decentralized systems where every entity independently maintains its own local replica of system state and acts autonomously, i.e., autonomously executes query and mutate operations. Instead of necessitating up-front coordination among entities, information on state changes is exchanged asynchronously. Thus, the operations offered by an entity are available with local latency regardless of network partitions and faults in other entities. In addition, large-scale local-first systems demand algorithms that tolerate Byzantine faults, since trust among entities is no longer warranted at scale. For fault tolerance, low latency, and eventual consistency, local-first systems rely on previous work on wait-freedom [13], coordination avoidance [4], and conflict-free replicated data types [1, 30].

A *group* is what we call a replicated object together with the set of entities that replicate and collaborate on that object, i.e., a chat group or collaborative document. The group's state encompasses both data, like the document state, and metadata, like the group name or permission assignments. Group state is replicated via a partially ordered set of state change *events* using the *hash chronicle* data type [16]. Hash chronicles are Byzantine fault-tolerant CRDTs that replicate a partially ordered event set via recursive hash linking: Any event is bound to its *causal history*, i.e., the chronicle subset that contains all causal precursor events back to group creation, by recursively linking its direct causal precursors. The concept behind hash chronicles represents the current community consensus for replication in large-scale local-first systems; the concept is also known as hash-linked DAGs (directed acyclic graphs, [20]), blocklace [2], Matrix Event Graph [14], or MerkleDAG [29].

Among the most sophisticated examples of large-scale local-first systems and frameworks is the Matrix [32] group communication system, as well as the Automerge/Keyhive [3, 37] and p2panda [25] CRDT frameworks for local-first collaborative applications. While Matrix is widely deployed already, Keyhive and p2panda are still in the research project phase, but also aim to scale collaborative local-first systems up to, e.g., a decentralized Wikipedia equivalent [37].

We recently proposed a conceptual model and security properties for a Matrix-like local-first authorization algorithm in an informal top-down approach [16]. In this paper, we follow a bottom-up approach and focus on constructing local-first access control semantics and simplified but formally verified algorithms, instead of characterizing what is currently present in Matrix. In his work on local-first authorization, Kleppmann focuses on resolving mutual authorization revocations among equally-privileged entities [18], which are out of scope for this work. The work of Rault et al. explores CRDT-based access control under the honest-but-curious assumption instead of Byzantine faults, and focuses on delegation, revocation, and compensating actions consecutive to access control concurrency conflicts [26, 27]. Other work on local-first authorization usually has different assumptions on Byzantine faults [26, 35], while formal work on CRDTs [1] usually does not integrate authorization [6, 10].

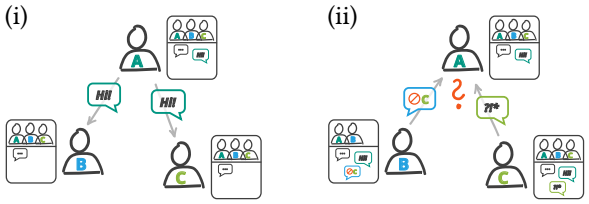

Figure 1. A chat group consisting of entities A, B, C. (i) Entity A adds a new message to its local chat history first, then broadcasts to B and C. (ii) Entity C adds a message while concurrently, B revokes the authorization that C utilized.

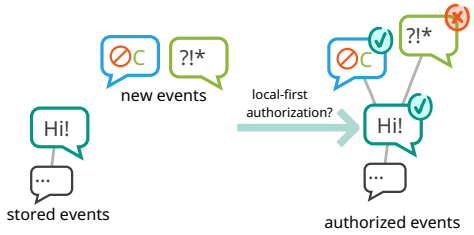

Figure 2. Given new events and stored events only, specifications for local-first authorization define how entities independently decide authorization of each event.

3 Problem Specification

As target for formal verification, pursued but not yet met in Sec. 4, we now specify the semantics and invariants of a Byzantine fault-tolerant, replicated data type for managing collaboration groups (cf. Fig. 1) via access control capabilities. While we focus on administrative access rights, i.e., rights to grant or revoke capabilities, our type also exemplarily supports querying and mutating a group name.

At the example of assigning a new group name, invocations, mutate, and query operations act together as follows: First, an invocation $v \in \mathbb{V}_{\text{assign}}$ is constructed, supplying the new group name. Second, the invocation is logged in the group chronicle via its $log(v)$ mutate operation, which combines the invocation with a timestamp to form an event. The group chronicle represents a replicated log of events, partially ordered by timestamp. For Byzantine fault tolerance, an event's timestamp is the set of identifier hashes of all its precursor events, instead of, e.g., the usual version vector [1]. Chronicle reconciliation, as described e.g. in [15], provides eventual consistency based on the *join*() mutate operation that merges two potentially diverging group chronicles to their least upper bound. Executing the *values*() query operation results in the set of latest, authorized assign events.

In access control terminology, a capability is one way to encode the access right (authorization) of a subject entity to invoke an action (operation) on a replicated object, potentially affecting a specific object entity or event. As shown in Fig. 2, the authorization algorithm gets the entity's current events and new events as input, and decides on the authorization of each event as output. The challenge of local-first authorization is to specify authorization semantics and security invariants that do not require coordination or trust among entities, but yet are as expressive as possible.

System Assumptions and Threat Model. We assume that an arbitrary strict subset f of the n overall system entities is Byzantine, i.e., $f/n < 1$. Byzantine entities may deviate arbitrarily from system algorithms, which includes malicious intent. The following assumptions restrict the abilities of Byzantine entities: Events are authenticated and integrity-protected, e.g. by digital signatures. Further, the access control system assumes a Byzantine fault-tolerant chronicle implementation that provides eventual consistency [15]. The replicated chronicle is assumed to be *append-only* for correct and Byzantine entities alike, achievable in practice through hash linking: any event includes the hashes of its precursor events (cf. [15]). Thereby, with a collision-resistant hash function, the causal history of any event is immutable: an attacker cannot forge new precursors of existing events. In practice, *recursive* hash linking allows to reduce timestamps to the hashes of the direct precursor events.

Byzantine entities have three main attack vectors: *omission*, *equivocation*, and *backdating* [15]. *Omission* allows Byzantine entities to selectively decide with which entities to share events with. Byzantine omission can be mitigated by simple anti-entropy mechanisms, as long as vector clocks and similar mechanisms relying on honest entities are avoided. *Equivocation* allows Byzantine entities to create two successor events concurrent to each other, while correct entities would usually assign an ordering between successor events. Equivocation aims to make correct entities receive events in a different ordering to exploit ordering dependencies in event processing. *Backdating* allows Byzantine entities to claim subsets of the chronicle as causal history, while correct entities would usually append events following the most recent events in their local state. Backdating can be seen as a generalization of equivocation. However, backdating aims to make new events look like events that were not received for a long time, to exploit the mechanisms that ensure order-independent event processing. While eventual consistency ensures that all correct entities eventually know of all events from equivocation and backdating, the access control system still has to decide how to deal with these events in an eventually consistent way. A secure local-first access control system must prevent Byzantine influence on the safety of event authorization, capability revocation, and query results through backdating and equivocation.

Data Type Semantics. We discuss the semantics and invariants in mathematical form to define the intermediate goal pursued in formal verification. The Rust formalization discussed in Sec. 4 consists of different simplifications of this data type. We define the group type as a composition of the underlying group chronicle and types for the group's capabilities and name. We define query operations for capabilities, authorization, and group name based on local group state, as well as operation invocations for granting and revoking capabilities, and assigning a group name. Invocations mutate group state by being logged to the group chronicle. We assume that the group chronicles is replicated as Byzantine fault-tolerant hash chronicle. Inspired by Keyhive [37], we define authorization semantics based on capabilities [11, 28]: Authorized subject entities can grant an object entity the capability to invoke a specific operation, encoded as a capability grant event in the chronicle. As claim of authorization, invocations of authorized subject entities present the identifier of a corresponding capability grant. Authorized subject entities can revoke a capability grant again, also referencing the grant event identifier. We believe that capabilities in this "granted-present-revoke" style, analogous to "observed-remove" set CRDTs [1], are well-suited to capture user intent in a way supported by local-first authorization algorithms, while resulting in desirable conflict resolution semantics [36].

Spec. 1 contains the chronicle type as eventually consistent log of events, partially ordered by timestamps, interlinked by event identifiers. Spec. 2 extends the group chronicle with capabilities and authorizations, as well as a group name.

In Spec. 1, an event $e \in \mathbb{E}$ encodes the invocation $e.voc \in \mathbb{V}_{\mathrm{m}}$ of an operation, like a group name assignment, at time $e.tme \in \mathbb{T}$. An event e has a unique identifier $id(e) \in \mathbb{I}$ used to reference the event, i.e., a unique digest from a collision-resistant hash function. To ease notation, a timestamp $T \in \mathbb{T} \subsetneq \mathcal{P}(\mathbb{I})$ represents the causal history of an event as the downward-closed set of identifiers of all precursor events. In practice, timestamps would be compressed to the identifier set of the direct precursors. Invocations have a subtype in $\mathbb{M} = \{\text{create}, \text{grant}, \text{revoke}, \text{assign}\}$, we write $\mathbb{V}_{m \in \mathbb{M}}$ for all invocations of subtype m, and $\mathbb{E}_{m \in \mathbb{M}}$ for all events with invocations of subtype m. The group create event and its precursors, used for initial group state setup, are authorized by definition, other events e present a reference to a capability grant event, $e.voc.grnt$, as claim of authorization. A group state G is a set of events $G \in \mathbb{G} \subseteq \mathcal{P}(\mathbb{E})$, partially ordered by their timestamps.

The chronicle type provides the following operations:

$create(sbj \in \mathbb{S}) \to v \in \mathbb{V}_{\text{create}}$ constructs a group create invocation, to be logged after a set of invocations that define initial group state. Logging a create event marks the end of group setup by making the chronicle valid.

$log(G \in \mathbb{G}, v \in \mathbb{V}) \to G' \in \mathbb{G}$ logs invocation v as successor event of all events in G.

Spec. 1 A group chronicle provides query and mutate operations on its partially ordered log of events. Events combine an invocation with a timestamp. A timestamp is a set of precursor event identifiers. Identifiers are collision-resistant.

type *Event* $\mathbb{E} \subseteq \mathbb{T} \times \mathbb{V}$ ▷ *an event combines a timestamp with an invocation*
  construct *event* $(T \in \mathbb{T}, v \in \mathbb{V}) \to e \in \mathbb{E}$
  query *tme* $(e \in \mathbb{E}) \to T \in \mathbb{T}$
  query *voc* $(e \in \mathbb{E}) \to v \in \mathbb{V}$
  query $<$ $(e_1, e_2 \in \mathbb{E}) \to id(e_1) \in e_2.tme$
  query $\|$ $(e_1, e_2 \in \mathbb{E}) \to e_1 \neq e_2 \wedge e_1 \not< e_2 \wedge e_2 \not< e_1$

type *EventIdentifier* $\mathbb{I}$ ▷ *identifiers provide unique event representations.*
  construct *id* $(e \in \mathbb{E}) \to i \in \mathbb{I}$
    ensures $\forall e_1, e_2 \in \mathbb{E}\colon\ id(e_1) = id(e_2) \Leftrightarrow e_1 = e_2$ ▷ *collision resistance*

type *Timestamp* $\mathbb{T} \subseteq \mathcal{P}(\mathbb{I})$ ▷ *downward-closed sets of event identifiers*
  construct *from* $(E \subseteq \mathbb{E}) \to \{id(e) \mid e \in E\}$
  query *valid* $(T \in \mathbb{T}) \to a \in \{\bot, \top\}$ ▷ *T downward-closed?*
    $a \leftarrow \forall i \in T, e, \check{e} \in \mathbb{E}\colon\ i = id(e) \wedge \check{e} < e \Rightarrow id(\check{e}) \in T$

type *GroupChronicle* $\mathbb{G} \subseteq \mathcal{P}(\mathbb{E})$ ▷ *group chronicles are sets of events*
  construct *create* $(sbj \in \mathbb{S}) \to v \in \mathbb{V}_{\text{create}}$ ▷ *declares initial group setup*
  mutate *log* $(G \in \mathbb{G}, v \in \mathbb{V}) \to g' \in \mathbb{G}$
    $G' \leftarrow G \cup \{event(G.now(), v)\}$ ▷ *logs invocation v as event at G.now()*
  mutate *join* $(G_1 \in \mathbb{G}, G_2 \in \mathbb{G}) \to G' \in \mathbb{G}$
    $G' \leftarrow G_1 \cup G_2$ ▷ *merges G_1 and G_2 to their least upper bound*
  query *events* $(G \in \mathbb{G}) \to E \subseteq \mathbb{E}$
    $E \leftarrow \{e \in G\}$
  query *now* $(G \in \mathbb{G}) \to T \in \mathbb{T}$ ▷ *current timestamp of G*
    $T \leftarrow \mathbb{T}.from(G.events())$
  query *creation* $(G \in \mathbb{G}) \to e_c \in G \cap \mathbb{E}_{\text{create}}$ ▷ *create event of G*
    requires $G.valid()$
  query *pre* $(G \in \mathbb{G}, e \in \mathbb{E}) \to \check{G} \subseteq G$ ▷ *wind G back precursors of e*
    requires $G.valid() \wedge e \in G$
    $\check{G} \leftarrow \{\check{e} \in G \mid \check{e} < e\}$
  query *conc* $(G \in \mathbb{G}, e \in \mathbb{E}) \to \check{G} \subseteq G$ ▷ *precursor and events concurrent to e*
    requires $G.valid() \wedge e \in G$
    $\check{G} \leftarrow \{\check{e} \in G \mid \check{e} < e \vee \check{e} \| e\}$
  query *valid* $(G \in \mathbb{G}) \to a \in \{\bot, \top\}$
    $a \leftarrow \exists_{=1} e_c \in G \cap \mathbb{E}_{\text{create}}$ ▷ *create event is unique*
      $\wedge \forall e \in G\colon e < e_c \vee e = e_c \vee e > e_c$ ▷ *no events concurrent to e_c*
      $\wedge e.tme.valid()$ ▷ *timestamp is downward-closed*

join$(G_1 \in \mathbb{G}, G_2 \in \mathbb{G}) \to G' \in \mathbb{G}$ merges two diverging chronicles to their least upper bound via set union, providing the foundation for eventual consistency.

events$(G \in \mathbb{G}) \to E \in \mathbb{E}$ returns all events in G.

now$(G \in \mathbb{G}) \to T \in \mathbb{T}$ derives the current timestamp considering all events in G.

creation$(g \in \mathbb{G}) \to e_c \in \mathbb{E}_{\text{create}}$ returns the create event.

pre$(G \in \mathbb{G}, e \in \mathbb{E}) \to \check{G} \in \mathbb{G}$ winds chronicle G back to the state $\check{G}$ right before e was logged.

conc$(G \in \mathbb{G}, e \in \mathbb{E}) \to \check{G} \in \mathbb{G}$ winds G back to $\check{G}$, including only precursors to and events concurrent to e.

valid$(G \in \mathbb{G}) \to a \in \{\bot, \top\}$ verifies that G has a unique create event, that there is no event concurrent to the create event, and that all events have a valid, i.e., downward-closed, timestamp.

Spec. 2 The group capabilities type extends the chronicle with capability management and queries for authorization. Assigning the group name requires the corresponding capability. Boxes show authorization algorithm calls.

type *GroupCapabilities* $C(\mathbb{G})$ ▷ *group capabilities are sets of* grant *events*
  construct *grant* $(sbj \in \mathbb{S}, grnt \in \mathbb{I}, cap \in \mathbb{M}, obj \in \mathbb{S}) \to v \in \mathbb{V}_{\text{grant}}$
  construct *revoke* $(sbj \in \mathbb{S}, grnt \in \mathbb{I}, obj \in \mathbb{I}) \to v \in \mathbb{V}_{\text{revoke}}$

  query *caps* $(G \in \mathbb{G}) \to cs \subseteq G \cap \mathbb{E}_{\text{grant}}$ ▷ *granted, unrevoked capabilities*
    requires $G.valid()$
    $cs \leftarrow \{gr \in G \cap \mathbb{E}_{\text{grant}} \mid \boxed{G.authorizes(gr)}$
      $\wedge (\nexists rv \in G \cap \mathbb{E}_{\text{revoke}}\colon \boxed{G.authorizes(rv)} \wedge rv.voc.obj = id(gr))\}$
  query *authorizes* $(G \in \mathbb{G}, e \in \mathbb{E}) \to a \in \{\bot, \top\}$ ▷ *e authorized by G?*
    requires $G.valid()$
    requires $e.tme \subseteq G.now() \wedge e.tme.valid()$
    $a \leftarrow e = G.creation()$ ▷ *creation is authorized by definition*
      $\vee e < G.creation()$ ▷ *creation authorizes its precursors*
      $\vee ($ ▷ *other events need a matching, authorized grant precursor*
        $\exists gr \in G.pre(e) \cap \mathbb{E}_{\text{grant}}\colon G.authorizes(gr)$
          $\wedge\ id(gr) = e.voc.grnt$
          $\wedge e.voc.sbj = gr.voc.obj \wedge e.voc \in \mathbb{V}_{gr.voc.cap}$
        $\wedge \nexists rv \in G.conc(e) \cap \mathbb{E}_{\text{revoke}}\colon G.authorizes(rv)$
          $\wedge rv.voc.obj = e.voc.grnt$
        ▷ *… but no matching, authorized revocation, before or concurrent*
      $) \wedge e \in \mathbb{E}_{\text{revoke}} \Rightarrow \exists gr \in G.pre(e) \cap \mathbb{E}_{\text{grant}}\colon\ id(gr) = e.voc.obj$
        ▷ *revocations must match a grant precursor*

type *GroupName* $N(\mathbb{G})$ ▷ *group name is the latest set of* assign *events*
  construct *assign* $(sbj \in \mathbb{S}, grnt \in \mathbb{I}, n \in N) \to v \in \mathbb{V}_{\text{assign}}$
  query *values* $(g \in \mathbb{G}) \to vs \in \mathcal{P}(\mathbb{E}_{\text{assign}})$ ▷ *latest group names*
    requires $G.valid()$
    $vs \leftarrow \{\hat{e} \in g \cap \mathbb{E}_{\text{assign}} \mid \boxed{g.authorizes(\hat{e})}$ ▷ *latest, authorized assigns*
      $\wedge (\nexists e \in g \cap \mathbb{E}_{\text{assign}}\colon \boxed{g.authorizes(e)} \wedge e > \hat{e})\}$

Spec. 2 extends the group chronicle with operations for managing group capabilities and group name. Capability-based access control is provided by the following operations:

grant$(sbj \in \mathbb{S}, grnt \in \mathbb{I}, cap \in \mathbb{M}, obj \in \mathbb{S}) \to v \in \mathbb{V}_{\text{grant}}$ constructs an invocation in which entity *sbj* grants an entity *obj* the capability to invoke operations corresponding to *cap*. The invocation presents *grnt* as claim of authorization.

revoke$(sbj \in \mathbb{S}, grnt \in \mathbb{I}, obj \in \mathbb{I}) \to v \in \mathbb{V}_{\text{revoke}}$ constructs an invocation in which entity *sbj*, revokes the previous capability grant event referenced in *obj*, presenting *grnt* for authorization.

caps$(G \in \mathbb{G}) \to cs \in \mathcal{P}(\mathbb{E}_{\text{grant}})$ returns the set of authorized grant events in G that have not been revoked yet.

authorizes$(G \in \mathbb{G}, e \in \mathbb{E}) \to a \in \{\bot, \top\}$ whether group state G authorizes event e, mainly verifying that the precursors of e must contain a matching, authorized grant; while precursors and concurrent events must not contain a matching, authorized revocation.

The group name is managed by the following operations:

$assign(sbj \in \mathbb{S}, grnt \in \mathbb{I}, n \in \mathbb{N}) \to v \in \mathbb{V}_{\text{assign}}$ constructs an invocation in which entity *sbj* assigns name n, claiming authorization by *grnt*.

$values(G \in \mathbb{G}) \to vs \in \mathcal{P}(\mathbb{E}_{\text{assign}})$ returns the set of the most recent, authorized assign events in G, i.e., using multi-value resolution of concurrency conflicts.

In the remainder of this section, we discuss the security invariants underlying our design. The semantics of a capability grant event is to authorize successor events presenting the granted capability, while a revocation event matching the grant deauthorizes successor and concurrent events of the revocation that present the granted capability. It seems consequential that the authorization of any event must thereby depend on precursor and concurrent events only, and that successor events must not affect authorization of precursors. However, note that this is a fallacy induced by concurrency seeming like a transitive relation especially in graphical representations, while, in fact, it is non-transitive: Assume a revocation a_i de-authorizes a concurrent usage b_i. Then, a revocation c_{i+1} concurrent to a_i but successor to b_i deauthorizes a_i and thereby re-authorizes b_i, despite being a precursor of c_{i+1}. This anomaly follows from the expansion of the scope of effect of revocations to concurrent events. However, the expansion is necessary to ensure safety of revocations against backdating: Byzantine entities can backdate to exclude a revocation from the precursors of new events, in an effort to circumvent the revocation. Therefore, to make backdating ineffective, authorized revoke events must affect authorization of successors as well as concurrent events [18].

Related work differentiates between two notions of event authorization, which we call *precursive* authorization and *discursive* authorization (cf. storage and execution authorization in [16]). These notions do not define *how* authorization is performed, but differentiate subsets of group state on which authorization is based on. *Precursive authorization* is based on the precursors of event e in group G, i.e., given by $G.pre(e).authorizes(e)$. As the precursor set is immutable, precursive authorization is immutable as well. Correct entities broadcast and merge precursively authorized events only. *Discursive authorization* is based on precursors and events concurrent to e, given by $G.authorizes(e)$. However, as discursive authorization of concurrent events depends on *their* concurrent events, any event in G may affect discursive authorization of e, regardless of the relation to e. Thereby, discursive authorization is mutable: events may lose or regain discursive authorization due to new revocation events.

We characterize invariance of event authorization with the exception of revocations in the *authorization safety* invariant. Spec. 2 realizes authorization safety through collision-resistant event identifiers and timestamps, only revocations can lead to changes in discursive authorization due to the negated existential quantifier for revocations in *authorizes*().

Authorization Safety For group state G and new event e_{m}, causal authorization of any event $e \in G$ is invariant. Concurrent authorization is invariant with the exception of authorized revocations.

$$\begin{aligned}&\forall G \in \mathbb{G}, e_m \in \mathbb{E}, G' = G \cup \{e_m\}, G.valid() \land G'.valid():\\ &\quad \forall e \in G:\\ &\qquad G.pre(e).authorizes(e) = G'.pre(e).authorizes(e)\\ &\qquad \land e_m \notin \mathbb{E}_{\text{revoke}} \lor \neg G'.authorizes(e_m)\\ &\qquad \Rightarrow G.authorizes(e) = G'.authorizes(e)\end{aligned}$$

In the end, access control on group state is about controlling the effect of logged events on queries, which we formalize as *query safety*. Discursive authorization is a necessary condition for an event to affect query results, and, per contrapositive, query results must be invariant under any discursively unauthorized event. In Spec. 2, query safety is realized by *caps*(), *authorized*(), and *values*() queries acting on discursively authorized events only.

Query Safety For group state G and new event e_m, if e_m causes varying results of a query operation $G.query()$, it must be discursively authorized.

$$\begin{aligned}&\forall G \in \mathbb{G}, e_m \in \mathbb{E}, G' = G \cup \{e_m\}, G.valid() \land G'.valid():\\ &\qquad G.query() \neq G'.query() \Rightarrow G.authorizes(e_m)\end{aligned}$$

The expansion of the scope of effect of revocations to include concurrent events introduces non-monotonic logic [12]: Depending on the reception order, correct entities may exhibit non-monotonic anomalies where they disagree on discursive authorization of backdated events at first, until eventually reaching consistency on G and thereby consistency on discursive authorization [16]. To make backdating ineffective and revocations safe, we state the *revocation safety* invariant to ensure that any backdated version of a discursively unauthorized event is also discursively unauthorized. In Spec. 2, the *authorized*() query ensures revocation safety by looking for concurrent revocation events matching the grant of the event to authorize.

Revocation Safety For group state G and invocation v, if v is unauthorized at timestamp $G.now()$, then v is also unauthorized at any earlier timestamp $T \subseteq G.now()$.

$$\begin{aligned}&\forall G \in \mathbb{G}, v \in \mathbb{V}, T \subseteq G.now(), G.valid():\\ &\qquad \neg G.authorizes(event(G.now(), v))\\ &\qquad\quad \Rightarrow \neg G.authorizes(event(T, v))\end{aligned}$$

As the semantics in Spec. 2 support our security invariants in any group state, i.e., including events of Byzantine entities, we call them Byzantine fault-tolerant.

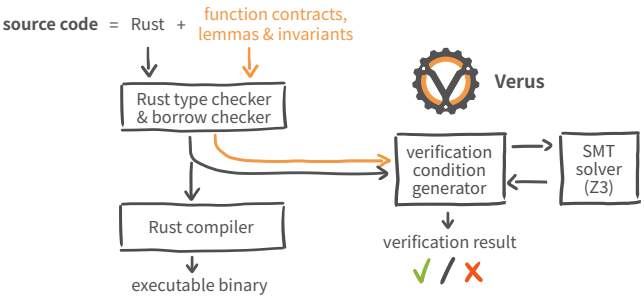

Figure 3. Verus extends Rust with formal verification at zero runtime cost by adding an SMT solver at compile time [22].

```
spec fn authorizes(&self: GroupSpec, event: EventSpec) -> bool {
  match event.operation.operator() {
    Operator::Create => event == self.bot(),
    operator @ _ => {
      Self::has_cap(self.c.pre(event), event.subject, operator)
      ...
exec fn log(&mut self, event: Event) -> (res: Option<EventID>)
  requires exists|id: EventID| !old(self)@.c.has_element(id), ...
  ensures res is Some <==> old(self)@.authorizes(event@), ... {
    if !self.authorizes(&event) { None } else {
      ...
```

Figure 4. Authorization rules formalized with Verus – a function contract for the executable function log ensures that only authorized events are applied to the system state, as defined by the specification function authorizes.

4 Verification

As illustrated in Fig. 3, Verus [22] is an SMT-based verification framework for Rust [24] code. Specification and verification add no runtime overhead, and Rust's ownership model rooted in linear type systems obviates the need for additional memory reasoning in separation logic. An implementation is annotated with and verified against its formal specification in an extended Rust syntax (cf. Fig. 4). Functions with the exec keyword are executed at runtime, i.e., make up the implementation. Functions with the spec keyword make up the specification, connected with exec functions via ensures and requires conditions. Specification code is called "ghost code" [23] due to being present only during verification.

4.1 Methodical Considerations

Byzantine fault tolerance. We abstract Byzantine fault tolerance of the underlying hash chronicle by assuming that authorization algorithms operate on a group object based on a valid chronicle. However, valid chronicles still contain Byzantine behaviour, like events from equivocation or backdating. Verifying the security invariants for authorization algorithms for any valid chronicle thereby also shows their Byzantine fault-tolerant implementation.

Total functions terminate and yield a result of the expected return type [34]. Proving that all employed functions are total even in Byzantine environment is necessary for latency and fault tolerance of local-first systems. Verus verifies totality for spec functions and termination of any loops and recursive calls of exec functions. Rust does not have an exception system, but functions can abort execution and unwind the call stack on severe error conditions via *panic*!(). To guarantee that functions always yield a result, we must ensure that functions do not *panic*!(), which can be specified by the Verus keyword *no_unwind*. However, the vstd – Verus' specification of Rust's std standard library – does not yet provide *no_unwind* guarantees for any of the functions we used. With a modified vstd, and under the assumption that std functions terminate and system calls for memory allocation and random number generation do not panic, we proved totality of all our functions.

Pure functions deterministically derive the same output given the same input, and are free of side effects. We verify strong convergence of query results by proving that all queries are total, pure functions of group state. As spec functions in Verus are guaranteed to be pure, we can prove purity of query functions by specifying that the return value of the exec query is equal to its corresponding spec query.

4.2 Authorization Algorithm Verification Case Study

We successfully used Verus for formalizing and verifying a simplification of the group type semantics specification from Sec. 3 (source code cf. [31]). We approached formalization in three steps with increasing complexity: In the first step, capability grants are possible before, but not after group creation. The second step enables capability grants after creation, but is still monotonic, i.e. does not support revocation. In the third step, only revocations are supported after group creation, which introduces non-monotonic effects.

1. Allow on creation. This algorithm assumes that grants are defined before creation and are immutable thereafter, i.e., the grant events logged before creation only grant the assign capability, but neither grant nor revoke capabilities. This algorithm serves as baseline for other algorithms, focusing on authorization safety: Because authorization is immutable, discursive authorization is equal to precursive authorization. As only assign events are ever authorized, the conditions for verifying query safety also simplify substantially.

2. Deny on creation, grant later. This algorithm assumes that initial group setup only contains a single grant event, which grants the group creator the capability to grant entities the assign capability. As no entity has the capability to revoke grants, capabilities evolve monotonically, and precursive and discursive authorization is still equivalent. This step focuses on authorization and query safety, simplified by the equality between precursive and discursive authorization.

3. Allow on creation, revoke later. This algorithm assumes that all entities are granted the assign capability during group setup. Additionally, the group creator is granted the revoke capability. At first glance, a revoke-only access

control system seems to be equally monotonic as a grant-only system: the capability set *caps*() is now monotonically shrinking, instead of growing. However, for any event e, the set of discursively authorized events that are concurrent to e, i.e., the set of events that may influence query results, can now both grow and shrink. Consequently, the system behaviour is no longer monotonic: In the light of new information, i.e., a revocation, discursive authorization of events can change, and with them previous beliefs on system state, requiring the revocation safety invariant. [12]. By serving as the base case for non-monotonic behaviour, this algorithm allows to focus on the non-monotonic aspects of authorization and query safety invariants.

5 Discussion

Methodical Discussion. For our verification, we needed to add one additional assumption missing in Verus' `vstd` specification of Rust's `std` standard library, namely that cloning preserves the elements of a `HashSet`. Like all other `vstd` specifications, this is true under the assumption that the `std` implementation is correct.

While many of our `exec` functions are straightforward translation of their `spec` counterpart to regular Rust code, Verus' limited support for the `Iterator` trait and combinator functions like `filter`, `map`, `any`, etc., required us to often implement such functionality with `for` loops instead. Verifying `for` loops in place of what otherwise would have been a chain of iterator combinators was one of the most time-consuming tasks of the overall verification process. We expect that future work will greatly benefit from iterator support in Verus currently in development.

Result Discussion. Under the assumption of Byzantine fault-tolerant eventual consistency of the underlying hash chronicle [15], our *query results are eventually consistent* due to the proof of purity and totality of query functions.

In our verified algorithms, concurrently issued assign and revoke events are the only possible authorization-related concurrency conflict, avoiding mutual authorization revocations among equally-privileged entities. This allowed us to directly specify and verify simplifications of mutate and query operations from Spec. 2 for our algorithms, which ensure *query safety* via filtering for authorized events. *Precursive authorization safety* is directly translated into a type invariant of the group type. *Discursive authorization safety* and *revocation safety* are ensured by the granted-present-revoke construction implemented in *authorizes*(). The operation specifications, together with the definition of *authorizes*(), are strong enough to show that the invariant holds.

Next Steps: Increasing Expressiveness. Although monotonic capabilities would drastically reduce the required complexity of local-first authorization, the ability to both grant and revoke capabilities allows entities to undo erroneous grants, and trust towards an entity might change over time. The next step in expressiveness therefore is to *support both* grant *and* revoke *events in a single algorithm*. This necessitates to prove more complicated formulas with nested quantifiers to preserve the invariants. An extension with a query capability to be presented on invoking query operations as a way to represent group membership should then be sufficient for basic group management.

The next level is to support *delegation*, i.e., granting of grant and revoke capabilities after group creation. Delegations require handling or prevention of mutual, concurrent revocations, and will substantially increase verification complexity. In addition, group members require the capability to self-revoke their query capability to be able to leave a group at their discretion. Publicly-joinable groups even require granting the capability to self-grant the query capability to all system entities, so that they can join the group at their discretion by granting themselves the query capability.

In continuation, the semantics should allow for a form of *sparse replication* of the group object without impeding its security guarantees, as well as support seamless *changeover* from one group to the next for archival, garbage collection, or protocol upgrade purposes. A verified implementation of these concepts would provide practical advantages that surpass the current state of the art.

6 Conclusion

We proposed a bottom-up approach to construct formally verified local-first authorization algorithms as a starting point towards the future goal of verifying algorithms with the expressiveness of real-world local-first systems such as Matrix and Keyhive. We presented a first specification of capability-based semantics and invariants for local-first access control on a replicated data type for managing collaboration groups. We also provided a proof of concept by formally verifying a simplified specification and implementation: Assuming an eventually consistent chronicle, but allowing for backdating by Byzantine entities, we verified our security invariants. Verus promises to be an accessible, future-proof tool for bridging the gap between current systems specifications and formal methods. However, scaling up formalization and verification towards the level required for practical local-first access control remains a challenge.

Acknowledgments

This work was funded by the Helmholtz Pilot Program Core Informatics. We like to thank Martin Kleppmann for his thoughts on local-first access control, as well as the anonymous PaPoC 2026 reviewers for their thoughtful and substantial reviews that will help us progress with our research.

References

[1] Paulo Sérgio Almeida. 2024. Approaches to conflict-free replicated data types. *ACM Comput. Surv.* 57, 2 (Nov. 2024), 51:1–51:36. doi:10.1145/3695249

[2] Paulo Sérgio Almeida and Ehud Shapiro. 2025. The Blocklace: A Byzantine-repelling and Universal Conflict-free Replicated Data Type. doi:10.48550/arXiv.2402.08068 arXiv:2402.08068 [cs].

[3] Automerge Contributors. 2026. *Automerge: Version Control for Your Data.* https://automerge.org/

[4] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. 8, 3 (2014), 185–196. doi:10.14778/2735508.2735509

[5] David Basin, Nate Foster, Kenneth L. McMillan, Kedar S. Namjoshi, Cristina Nita-Rotaru, Jonathan M. Smith, Pamela Zave, and Lenore D. Zuck. 2025. It takes a village: bridging the gaps between current and formal specifications for protocols. *Commun. ACM* 68, 8 (Aug. 2025), 50–61. doi:10.1145/3706572

[6] Liangrun Da. 2024. *Design and verification of Byzantine fault tolerant CRDTs.* Master's thesis. TUM. https://github.com/TUM-DSE/research-work-archive/blob/main/archive/2024/winter/docs/msc_liangrun_da_design_and_verification_of_byzantine_fault_tolerant_crdts.pdf

[7] Kegan Dougal. 2025. Project Hydra: improving state resolution in Matrix. https://matrix.org/blog/2025/08/project-hydra-improving-state-res/

[8] Element Software GmbH. 2025. *Matrix in Germany.* https://element.io/matrix-in-germany

[9] Element Software GmbH. 2024. NATO NI2CE Messenger utilises the power of decentralised communication. https://element.io/blog/nato-ni2ce-messenger-utilises-the-power-of-decentralised-communication/

[10] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–28. doi:10.1145/3133933

[11] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. 2013. A Capability-Based Security Approach to Manage Access Control in the Internet of Things. *Mathematical and Computer Modelling* 58, 5 (2013), 1189–1205. doi:10.1016/j.mcm.2013.02.006

[12] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM: When Distributed Consistency Is Easy. *Commun. ACM* 63, 9 (2020), 72–81. doi:10.1145/3369736

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

[14] 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. doi:10.1109/ACCESS.2021.3058576

[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. doi:10.1145/3642976.3653034

[16] Florian Jacob and Hannes Hartenstein. 2025. To the Best of Knowledge and Belief: On Eventually Consistent Access Control. In *Proceedings of the 15th ACM Conference on Data and Application Security and Privacy.* Association for Computing Machinery (ACM), 107–118. doi:10.1145/3714393.3726520

[17] Martin Kleppmann. 2015. *A Critique of the CAP Theorem.* arXiv:1509.05393 [cs] doi:10.48550/arXiv.1509.05393

[18] Martin Kleppmann. 2025. Keynote: Byzantine eventual consistency and local-first access control *(12th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '25)).* ACM. https://martin.kleppmann.com/2025/03/31/papoc-keynote-byzantine.html

[19] Martin Kleppmann. 2024. The Past, Present, and Future of Local-First. (2024). https://martin.kleppmann.com/2024/05/30/local-first-conference.html

[20] Martin Kleppmann. 2022. Making CRDTs Byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '22).* Association for Computing Machinery, New York, NY, USA, 8–15. doi:10.1145/3517209.3524042

[21] 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.* ACM, Athens Greece, 154–178. doi:10.1145/3359591.3359737

[22] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: a practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles.* ACM. doi:10.1145/3694715.3695952

[23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 286–315. doi:10.1145/3586037

[24] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology.* ACM. doi:10.1145/2663171.2663188

[25] p2panda Contributors. 2026. *p2panda: building blocks for peer-to-peer applications.* https://p2panda.org/

[26] Pierre-Antoine Rault. 2024. *Access control mechanisms for collaborative systems without central authority.* Ph. D. Dissertation. Université de Lorraine. https://hal.univ-lorraine.fr/tel-05079538

[27] Pierre-Antoine Rault, Claudia-Lavinia Ignat, and Olivier Perrin. 2023. Access Control Based on CRDTs for Collaborative Distributed Applications. https://inria.hal.science/hal-04224855

[28] R.S. Sandhu and P. Samarati. 1994. Access Control: Principle and Practice. *IEEE Communications Magazine* 32, 9 (1994), 40–48. doi:10.1109/35.312842

[29] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. 2020. *Merkle-CRDTs: Merkle-DAGs Meet CRDTs.* arXiv:2004.00107 [cs] doi:10.48550/arXiv.2004.00107

[30] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer Berlin Heidelberg, 386–400. doi:10.1007/978-3-642-24550-3_29

[31] Johanna Stuber and Florian Jacob. 2026. Towards System-Oriented Formal Verification of Local-First Access Control: Source Code. https://codeberg.org/kit-dsn/towards-formal-verification-local-first-access-control

[32] The Matrix.org Foundation CIC. 2025. Matrix Specification v1.16. https://spec.matrix.org/v1.16/

[33] The Matrix.org Foundation CIC. 2025. Room Version 12. https://spec.matrix.org/v1.16/rooms/v12/

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

[35] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2016. Access Control for Weakly Consistent Replicated Information Systems. In *Security and Trust Management* (2016) *(Lecture Notes in Computer Science).* doi:10.1007/978-3-319-46598-2_6

[36] Matthew Weidner. 2022. *Designing Data Structures for Collaborative Apps.* https://mattweidner.com/2022/02/10/collaborative-data-design.html

[37] Brooklyn Zelenka and Alex Good. 2025. *Keyhive: Local-first Access Control.* https://www.inkandswitch.com/keyhive/notebook/