

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 17 September 2026

D. Holmgren
B. Newbold
Bluesky Social
16 March 2026

Authenticated Transfer Repository and Synchronization
draft-holmgren-at-repository-01

Abstract

This document specifies the repository and synchronization semantics for Authenticated Transfer (AT), a protocol for cryptographically-verifiable storage and distribution of structured user-controlled data. It defines the AT repository that serves as the fundamental data storage model. It further specifies synchronization mechanisms that allow efficient distribution of repository changes to interested parties.

This document specifically deals with the repository and sync protocol. Overall network architecture is described further in [AT-ARCH].

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-holmgren-at-repository/>.

Source for this draft and an issue tracker can be found at <https://github.com/bluesky-social/ietf-drafts>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 September 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Repository	4
2.1. Repository Semantics	4
2.2. Repository Structure	5
2.3. User Identifiers	5
2.4. Commit Objects	6
2.5. MST Construction	7
2.5.1. Tree Structure	7
2.5.2. Layer Calculation	7
2.5.3. MST Construction Example	8
2.5.4. Empty Nodes	9
2.5.5. MST Node Schema	9
2.5.6. MST Node example	10
2.6. Commit Signatures	10
2.6.1. Signature Generation	11
2.6.2. Supported Curves	11
2.6.3. Signature Canonicalization	11
2.7. Deterministic CBOR Encoding	12
2.8. Repository Serialization Format	12
2.8.1. Header Format	12
2.8.2. Block Format	13
2.8.3. Block Ordering	13
3. Synchronization	13
3.1. Repository Revisions	14
3.1.1. Timestamp Identifier Format	14
3.2. Repository Diffs	14
3.3. Diff Verification Limitations	15
4. Real-time synchronization	16
4.1. Cursors	16
4.2. Streaming Events	17

4.2.1. Commit Events	17
4.2.2. Sync Events	18
4.3. Commit Validation	18
4.4. Re-synchronization	19
5. Security Considerations	20
5.1. CBOR Processing limits	20
5.2. MST Structure Attacks	20
5.3. Repository Import Validation	20
6. References	20
6.1. Normative References	20
6.2. Informative References	21
Authors' Addresses	21

1. Introduction

The Authenticated Transfer (AT) repository and synchronization protocol addresses the challenges of building decentralized applications that require consistent data replication across distributed multi-party infrastructure. Traditional web platforms maintain user data at a single network location, creating vendor lock-in and limiting user agency over their digital identity and published content.

In the AT model, user data is stored in cryptographically signed repositories that can be hosted, synchronized, and distributed by any compatible server while preserving data authenticity and user ownership. Each repository consists of a set of CBOR-encoded objects called records, organized lexicographically by type. The cryptographic structure allows repository contents to be re-distributed and cached by any network participant without requiring trust in intermediary hosts.

The synchronization system provides efficient mechanisms for propagating repository state changes across the network, supporting both real-time streaming updates and bulk synchronization scenarios. The protocol can detect dropped or withheld updates and provides cryptographic proofs for all operations, including record deletions, ensuring that consumers can maintain accurate and complete views of repository state.

This document specifically deals with the repository and sync protocol. Overall network architecture is described further in [AT-ARCH].

2. Repository

An AT repository provides a sorted key-value interface where values are CBOR-encoded objects referred to as records. Applications interact with repositories through standard CRUD operations while the underlying Merkle tree structure enables cryptographic verification of all modifications.

Repository authority is established through Decentralized Identifiers (DIDs). Each repository is associated with exactly one DID, which resolves to the cryptographic key material necessary for verifying repositories.

The repository structure provides several advantages over individually signed objects:

- * Simplified key rotation through a single repository-level signature rather than record-level signatures
- * Cryptographic proofs of record deletion
- * Completeness guarantees that enable observers to detect withheld, missing, or outdated records

This document describes version 3 of the AT repository format. Both previous versions are deprecated, and implementations do not need to support them.

2.1. Repository Semantics

Records are discrete units of user data, each CBOR-encoded and identified by a unique key within the repository. The repository is schema-agnostic and provides the foundational layer for higher-level data models and application semantics.

Repositories support individual record operations as well as batch writes that group multiple operations under a single commit, or signed mutation, to the repository. When applying batch operations, implementations should ensure that the resulting changes can be adequately represented within the synchronization system (see Section 4.2.1).

By convention, records are organized using a hierarchical two-part key structure consisting of a collection identifier and a record key. Record keys may be derived from timestamps or other monotonically increasing values, ensuring that new records are typically added to the lexicographically rightmost position within their collection.

Repository efficiency, especially in partial synchronization situations, benefits from grouping related records around lexicographically similar keys. This grouping allows for structural sharing within the repository data structure and reduces cryptographic proof sizes.

2.2. Repository Structure

AT repositories are organized as a Merkle Search Tree (<https://inria.hal.science/hal-02303490/document>) ([MST]) with a cryptographically signed commit referencing the tree root.

The MST provides several fundamental properties for repository operations. As a content-addressed structure, it enables efficient verification of data. The MST maintains lexicographic key ordering, enabling structural sharing of intermediate tree nodes for related records. It is probabilistically self-balancing, offering consistent performance characteristics. Additionally the MST exhibits unicity, meaning that any given set of keys and values will always produce the same tree structure and root hash regardless of insertion order.

Repository contents are encoded using deterministic CBOR serialization and organized as a directed acyclic graph where data objects reference each other through content hashes. These hash-identified data objects, referred to as "blocks," include three distinct types: commit objects, MST internal nodes, and user records.

Large binary data such as images and media files are not stored directly within repositories. Instead, such data is stored externally and referenced in records by a hash link.

2.3. User Identifiers

Repository authority is established through a resolvable user identifier specified in the repository commit (Section 2.4). AT employs Decentralized Identifiers (DIDs) as defined in [DID] for this purpose.

DIDs are globally unique identifiers that resolve to DID Documents containing cryptographic key material and other metadata associated with the identifier. Resolution enables independent verification of repository commits without dependence on centralized authorities.

Each repository must reference exactly one DID, and each DID may be associated with at most one AT repository.

The signing key for repository commits is specified within the DID document's `verificationMethod` array. The key entry must have an `id` field ending in `#atproto`. When multiple possible verification methods are present, implementations must use the first valid entry and ignore subsequent ones. The public key must be encoded using the `publicKeyMultibase` format as specified in [CONTROLLEDID]. The signing key must use one of the signing algorithms described in Section 2.6.2.

DID resolution may return supplementary information beyond the signing key, including canonical repository hosting locations, alternative user identifiers, or relevant service endpoints.

To ensure interoperability, AT restricts support to specific DID methods. Currently supported methods are `did:web` and `did:plc`. The resolution mechanisms and specifications for these methods are described in [DIDWEB] and [DIDPLC].

2.4. Commit Objects

Commit objects serve as the authoritative root of each repository, establishing cryptographic ownership and providing a verifiable reference to the state of a repository at a particular point in time. Each commit is digitally signed by the repository owner and contains metadata necessary for verification and synchronization.

A commit object contains the following data fields:

- * `*did*` (string, required): The resolvable user identifier associated with the repository as described in Section 2.3
- * `*version*` (integer, required): Repository format version, fixed value of `*3*` for the current specification
- * `*data*` (hash link, required): Hash pointer to the root of the repository's MST structure
- * `*rev*` (string, required): Repository revision identifier that functions as a logical clock and must increase monotonically (see Section 3.1).
- * `*prev*` (hash link, nullable): Optional pointer to the previous commit object in the repository's history chain. While included for backward compatibility with version 2 repositories, this field is typically null in version 3 implementations
- * `*sig*` (byte array, required): Cryptographic signature over the commit contents.

Commit signature generation and verification procedures are detailed in Section 2.6.1.

2.5. MST Construction

The MST structure is deterministically reproducible from any given key-value mapping, where keys are non-empty byte strings and values are hash link references to records. This deterministic construction ensures that identical input sets always produce the same root hash regardless of insertion order.

The tree's structural organization depends solely on the keys present, not on the record values they reference. When a record value changes, the new content hash propagates up through the tree nodes to the root, but the tree's shape and node organization remain unchanged.

2.5.1. Tree Structure

Each MST node contains a list of key-value entries and references to child subtrees. Entries and subtree links are maintained in lexicographic order, with all keys in a linked subtree falling within the range corresponding to that link's position. The ordering proceeds from left (lexicographically first) to right (lexicographically last).

Keys are assigned to tree levels based on a layer value computed from the key itself. Nodes at each level contain all keys with the corresponding layer value, while subtree links point to nodes containing keys that fall within specific lexicographic ranges but have lower layer values. Adjacent keys may appear within the same node, but adjacent subtrees must be separated by at least one key entry to prevent structural ambiguity.

2.5.2. Layer Calculation

The layer for a given key is calculated using SHA-256 with a 2-bit grouping scheme that provides an average fanout of 4:

1. Compute the SHA-256 hash of the key (byte string) with binary output
2. Count the number of leading binary zeros in the hash
3. Divide by 2, rounding down to the nearest integer

Examples of layer calculation:

- * key1: SHA-256 begins 100000010111... → layer 0
- * key7: SHA-256 begins 000111100011... → layer 1
- * key515: SHA-256 begins 000000000111... → layer 4

When processing the MST structure, implementations must verify the layer assignment and ordering of keys. While this verification is most essential for untrusted inputs, implementations should perform these checks consistently regardless of data source. Additional validation of node size limits and other structural parameters is required to prevent resource exhaustion attacks, as detailed in Security Considerations (Section 5).

2.5.3. MST Construction Example

The following is a Merkle Search Tree containing 9 records with keys A-I. Each key would include a pointer to some record hash, though that hash is irrelevant to the construction of the tree. Each asterisk (*) represents a hash pointer to the subtree under it.

For the sake of illustration assume the following layer calculations:

- * layer(D) = 2
- * layer(A|E|I) = 1
- * layer(B|C|F|G|H) = 0

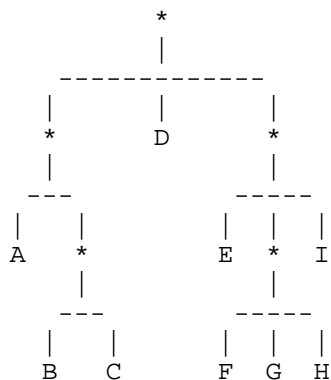


Figure 1: Example MST Structure

2.5.4. Empty Nodes

An empty repository containing no records is represented as a single MST node with no entries. This is the only case where a node without entries is permitted.

Nodes that contain no key entries but do contain subtree links are allowed at intermediate positions, provided those subtrees eventually contain key entries. However, such nodes are not permitted at the root position - the root must either contain key entries or be the special case of a completely empty repository. Similarly, nodes without key entries are not permitted at leaf positions except for the empty repository case.

This structure ensures that nodes lacking key-value entries are pruned from the top and bottom of the tree while preserving intermediate nodes that maintain proper height relationships and prevent subtree links from skipping layers.

2.5.5. MST Node Schema

Given their prevalence through the repository structure, MST nodes require a compact binary representation for storage efficiency. Keys within each node use prefix compression, where each entry specifies the number of bytes it shares with the preceding key in the array. The first entry in each node contains the complete key with a prefix length of zero. This compression applies only within individual nodes and does not extend across node boundaries. The compression scheme is mandatory to ensure deterministic MST structure across all implementations.

MST nodes contain the following fields:

- * l (hash link, nullable): Reference to a subtree node at a lower layer containing keys that sort lexicographically before all keys in the current node
- * e (array, required): Ordered array of entry objects, each containing:
 - p (integer, required): Number of bytes shared with the previous entry in this node
 - k (byte string, required): Key suffix remaining after removing the shared prefix bytes
 - v (hash link, required): Reference to the record data for this entry

- `t` (hash link, nullable): Reference to a subtree node at a lower layer containing keys that sort after this entry's key but before the next entry's key in the current node

2.5.6. MST Node example

The following example shows an MST node at layer 1 containing two subtree pointers and two key-value entries. The node contents in order are:

- * Left subtree: hash link 0x01711220643b9326...
- * Entry: key7 → record hash link 0x017112202d9aa87e...
- * Right subtree: hash link 0x0171122047e2886f...
- * Entry: key10 → record hash link 0x0171122010b6da2c...

This node would be encoded as follows:

```
{
  l: 0x01711220643b9326...
  e: [
    {
      p: 0,
      k: "key7",
      v: 0x017112202d9aa87e...
      t: 0x0171122047e2886f...
    },
    {
      p: 3,
      k: "10",
      v: 0x0171122010b6da2c...
      t: null
    }
  ]
}
```

2.6. Commit Signatures

Commit objects are signed by the key declared by the repository owner's resolvable identifier. Neither the signature nor the signed commit object contains information about the curve type or specific public key used for signing. This information must be obtained by resolving the repository's DID as specified in Section 2.3.

The most recent commit must always be verifiable using the currently resolvable signing key. When rotating signing keys, a new repository commit must be created, even if the contents and structure of the repository remain unchanged.

2.6.1. Signature Generation

To generate a commit signature:

1. Populate all commit data fields except the sig field
2. Serialize the unsigned commit using deterministic CBOR encoding (see Section 2.7)
3. Compute the SHA-256 hash of the serialized bytes
4. Sign the hash using the current signing key associated with the repository's DID
5. Format the signature as a concatenation of the 32-byte *r* and 32-byte *s* values
6. Add the resulting 64-byte signature to the commit object as the sig field

2.6.2. Supported Curves

AT implementations must support both of the following elliptic curves and signature algorithms:

- * NIST P-256 (also known as secp256r1 or p256) [SEC2]
- * secp256k1 (also known as k256) [SEC2]

2.6.3. Signature Canonicalization

ECDSA signatures exhibit malleability, allowing transformation into distinct but equally valid signatures without access to the private key or original data. While the security impact is limited, signature malleability could enable broadcast of multiple valid versions of the same repository commit with different hashes, potentially causing synchronization confusion.

To prevent such scenarios, AT requires all ECDSA signatures to be canonicalized in low-S form. Specifically, the *s* component of the signature must satisfy $s < n/2$, where *n* is the order of the curve's base point.

2.7. Deterministic CBOR Encoding

Repository content requires consistent binary representation across all implementations to ensure identical content hashes and verifiable integrity. All records, MST nodes, and commits must be encoded using Deterministically Encoded CBOR as specified in Section 4.2 of [CBOR], with map key ordering following the original specification in Section 3.9 of [RFC7049] for historical compatibility.

For interoperability purposes, hash links between repository objects are encoded using a specific format within the CBOR structure. SHA-256 hash links are represented as CBOR byte strings under tag 42, with the byte string containing the 32-byte hash value prefixed by the fixed byte sequence 0x01711220.

Hash links that point to arbitrary binary data instead of other repository objects should be encoded similarly though prefixed by the fixed byte sequence 0x01551220.

2.8. Repository Serialization Format

Repositories are serialized for transmission and storage as a concatenated sequence of block data, where blocks represent the CBOR-encoded records, MST nodes, and commit objects that comprise the repository structure. The serialization is prefixed with a header that identifies the root block, typically the repository's commit object.

Serialized repositories may contain partial repository state, such as when transmitting cryptographic proofs for specific records. In these situations, they may not include unrelated MST nodes or records outside the proof path.

2.8.1. Header Format

The header is constructed by CBOR-encoding an object with the following fields:

- * version (integer, required): Fixed value of 1
- * root (array, required): Single-element array containing the hash link of the commit block

The CBOR-encoded header is prefixed with its byte length encoded as an unsigned LEB128 integer as described in Section 5.2.2 of [WEBASSEMBLY].

2.8.2. Block Format

Following the header, each repository block is serialized by concatenating:

1. The combined byte length of the following two components, encoded as an unsigned LEB128 integer
2. The block's content hash, prefixed with 0x01711220 as specified in Section 2.7
3. The CBOR-encoded block data

```
|----- Header -----| |----- Data -----|
[ int | Header block ] [ int | hash | block ] [ int | hash | block ] ...
```

Figure 2: Repo Serialization Layout

2.8.3. Block Ordering

Block ordering should follow preorder traversal of the included repository portion when possible, though parsers must be tolerant of other or unexpected orderings.

Preorder traversal enables streaming verification of repositories, allowing parsers to walk the MST structure and output key-to-record mappings while maintaining minimal MST state in memory. This approach supports efficient processing of large repositories without requiring complete buffering of the serialized data.

3. Synchronization

The AT synchronization model operates on the principle that any participant can independently verify repository updates. This allows sync to occur between any client and server without requiring that the server is a canonical or trusted host.

AT supports multiple synchronization patterns: full repository synchronization for complete replicas, partial synchronization for specific record subsets, and proof-only synchronization for cryptographic verification without content retrieval.

The typical synchronization workflow establishes baseline state through full synchronization, then maintains currency through incremental updates. Full synchronization is performed by fetching a complete serialized repository over HTTPS.

3.1. Repository Revisions

Each repository maintains a rev field (short for “revision”) that functions as a logical clock for the progression of the contents of the repository over time. The revision value is a short string value that must increase lexicographically with each new commit.

Revisions may be used when comparing two repositories, especially when obtained from a non-canonical host, to determine which is more recent.

3.1.1. Timestamp Identifier Format

The recommended mechanism for generating revision values is the Timestamp Identifier (TID) format.

TIDs provide a standardized revision format with the following properties:

- * 64-bit integer with big-endian byte ordering
- * Base32-sortable encoding using characters
234567abcdefghijklmnopqrstuvwxyz
- * Fixed 13-character length with no padding (integer zero encodes as 2222222222222)

The layout of the 64-bit integer is:

- * The top bit is always 0
- * The next 53 bits represent microseconds since the UNIX epoch. 53 bits is chosen as the maximum safe integer precision in a 64-bit floating point number, as used by Javascript.
- * The final 10 bits are a random "clock identifier."

3.2. Repository Diffs

Repository diffs enable efficient synchronization by containing only the data that changed between two repository revisions. A diff includes the commit object, MST nodes, and records that differ between an older baseline revision and the current revision. Applying a diff to the baseline repository reconstructs the complete current repository state.

Diffs use the same serialization format as complete repositories, with the commit block serving as the root. A diff must include:

- * The new commit block
- * All created and updated record blocks
- * All MST nodes in the current repository that did not exist in the baseline revision

Required blocks must be included in the diff regardless of their presence in earlier repository history. For example, if an MST node was previously present in the repository, then deleted, and subsequently reintroduced during the range that the diff represents, then the diff must include that block even though it appeared in prior revisions.

Deleted records and past versions of updated records are excluded from diffs.

With the exception of deleted record data, the diff may include additional blocks which receivers should ignore.

3.3. Diff Verification Limitations

Repository diffs present verification challenges for consumers who do not maintain complete repository state. These consumers often wish to authenticate repository content and utilize records without persisting the entire repository structure, making diffs an attractive option for lightweight verification.

Diffs partially support this use case by providing a signed commit and the relevant portions of the Merkle tree, creating a verifiable proof chain for record creations, updates, and deletions. When a recipient possesses both a diff and a corresponding list of operations, they can use the diff contents to cryptographically verify that the operations are authentic.

However, observers without knowledge of the complete baseline repository state cannot reliably enumerate all operations by examining the diff contents alone. While comprehensive diffs reveal created or updated records by traversing to the leaf nodes, they provide no information about deletion operations that occurred during the period that the diff represents.

This means that while diffs enable verification of a known operation list, they cannot be used to exhaustively reconstruct the complete operation list from diff contents alone. However, if a recipient has a complete repository structure from some prior revision and receives a diff representing changes since that revision, they can compute the complete set of operations that occurred between the two versions.

This asymmetry means diffs alone cannot substitute for complete state tracking when comprehensive operation enumeration is required. An efficient mechanism for cross-verification of a diff and enumerated operation list against the prior repository commit state is described in {#streaming-validation}.

4. Real-time synchronization

AT supports real-time synchronization, enabling applications to receive repository updates with minimal latency through a pull-based WebSocket connection.

Real-time streams of repository updates are often referred to as the “firehose”. The firehose delivers events containing repository diffs along with supporting metadata necessary for verification and processing.

Each event includes a monotonic cursor that establishes a total ordering across all repository changes from a given host. This ordering enables reliable event replay and ensures that consumers can maintain consistent state even when reconnecting after network interruptions.

AT allows consumers to maintain fully-verified copies of repository records without storing the underlying Merkle tree structure, providing an efficient method for applications that need authenticated content access without the overhead of complete repository replication.

4.1. Cursors

Real-time synchronization streams include per-message cursors to improve transmission reliability. Cursors are positive integers that increase monotonically across the stream. Cursor semantics are flexible, and they may contain arbitrary gaps between consecutive messages.

Consumers track the last cursor value they successfully processed and can specify this cursor when reconnecting to receive any missed messages within the provider’s rollback window. Providers maintain no persistent consumer state across connections, relying entirely on the cursor values supplied by consumers during reconnection.

Stream behavior depends on the cursor value specified during connection:

- * ***No cursor specified***: The provider begins transmitting from the current stream position, providing only new messages generated after the connection is established.
- * ***Future cursor***: When the requested cursor exceeds the current stream cursor, the provider sends an error message and closes the connection.
- * ***Cursor within rollback window***: The provider transmits all persisted messages with cursor numbers greater than or equal to the requested cursor, then continues with the real-time stream once caught up.
- * ***Cursor older than rollback window***: The provider sends an informational message indicating that the requested cursor is too old, then begins transmission at the oldest available event, sends the entire rollback window, and continues with the real-time stream.
- * ***Cursor value of 0***: The provider treats this as a request for the complete available history, starting at the oldest available event, transmitting the entire rollback window, then continuing with the real-time stream.

4.2. Streaming Events

The real-time stream delivers two types of events: commit and sync.

4.2.1. Commit Events

Commit events represent an atomic set of repository modifications and consist of a repository diff combined with some supporting metadata.

The diff **MUST** include the new commit and all blocks in the Merkle proof chain for any modified key, as well as blocks for keys directly adjacent to the modified keys. The rationale for including adjacent keys is detailed in Section 4.3.

The metadata provides additional context required for processing and verification and includes:

- * The revision of the repository after the modifications
- * The revision of the repository before the diff
- * The root hash of the repository MST before the diff

- * A description of the operations contained in the diff with each containing
 - the key
 - the hash of the new record at the key (in the case of a create/update)
 - the hash of the old record at the key (in the case of an update/delete)

A single commit events must contain no more than 200 repository operations and the full serialized event should be no larger than 2MB. Mutations that do not fit in these limits should instead be communicated through Sync Events.

4.2.2. Sync Events

Sync events declare the current state of a repository, regardless of the previous state.

Sync events are emitted when commit events cannot adequately describe the transition between repository revisions. This may occur in several scenarios:

- * Large mutations that exceed the practical size limits for commit events
- * Data loss or corruption that breaks the continuity of commit history
- * Account migration between different infrastructure providers

In these cases, a sync event provides a reset point that encourages consumers to resynchronize against the current authoritative state without requiring knowledge of the intervening changes.

4.3. Commit Validation

Commit validation occurs through a two-step process that ensures both the validity of the repository transition and the consumer's resulting synchronization state.

First, the consumer validates that the commit represents a valid transition from a previous repository revision (revA) to the new revision (revB). Second, the consumer confirms that they last observed the repository at revA. Together, these steps establish that the repository is now definitively at revB.

The validation process inverts all operations against the partial MST provided in the diff. That is, each “create” operation will be inverted as a “delete” operation on the same key and applied to the tree. Each “delete” will become a “create” of the same record, and every “update” will be updated back to the previous value.

If the operation list is complete and accurate, applying the inverse operations will reconstruct the tree state as it existed before the commit. The hash of this reconstructed tree must match the previous root hash of the MST as specified in the commit event. If the hashes match then the provided list of operations is accurate and exhaustive.

Because the previous MST root hash is included in the commit event, commits can be validated for internal consistency independent of any local state. If the operation inversion process fails to produce a tree hash matching the declared previous root, the entire commit event should be treated as invalid.

If the commit is internally consistent but its declared previous root does not match the previous MST root stored locally, then the consumer has become desynchronized, indicating missed events or a disjunction in the producer’s commit history.

4.4. Re-synchronization

When a consumer detects desynchronization, either through a disjunction in commit history or a sync event that does not match their local state, they must perform a complete re-synchronization process to restore consistency with the current repository state.

Re-synchronization requires fetching and processing the full repository structure, though the record contents themselves are optional depending on the consumer’s needs. If the repository data is delivered in pre-order traversal, it can be validated incrementally as it streams in, producing a mapping of keys to record hashes that represents the complete repository state.

This key-to-hash mapping can be compared against existing local state to identify discrepancies and verify the integrity of the re-synchronization. Once validated, this mapping establishes the new baseline state against which future commit events can be applied.

During the re-synchronization process, any incoming commit events for the repository should be buffered rather than processed immediately. Once re-synchronization completes successfully, these buffered commits can be validated and applied in sequence to bring the consumer fully up to date with the current repository state.

5. Security Considerations

Repositories constitute untrusted input as account holders have complete control over repository contents and repository hosts control binary encoding. Implementations must handle potential denial of service vectors from both malicious actors and accidental conditions such as corrupted data or implementation bugs.

5.1. CBOR Processing limits

Generic precautions must be followed when processing CBOR data, including enforcement of maximum serialized object size, maximum recursion depth for nested structures, and memory budget limits for deserialized data. While some CBOR implementations include these protections by default, implementations should verify and configure appropriate limits regardless of library defaults.

5.2. MST Structure Attacks

The efficiency of MST data structures depends on a uniform distribution of key hashes. Since account holders control record keys, they can perform key mining to generate sets of keys with specific layer assignments and sorting characteristics, resulting in inefficient tree structures. Such attacks can cause excessive storage overhead and network amplification during synchronization.

To mitigate these attacks, implementations should:

- * Limit the number of entries per MST node to a statistically reasonable maximum
- * Impose limits on overall repository height
- * Monitor and restrict other structural parameters that could be exploited through sophisticated key mining

5.3. Repository Import Validation

When importing repositories, implementations should verify the completeness and integrity of the repository structure. Serialized repositories may contain additional unrelated blocks beyond those required for the repository structure. Care should be taken during storage to avoid resource waste on unreferenced blocks and to prevent potential storage exhaustion attacks.

6. References

6.1. Normative References

- [CBOR] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <<https://www.rfc-editor.org/rfc/rfc8949>>.
- [CONTROLLEDID] Longley, D., Sporny, M., Sabadello, M., Reed, D., Steele, O., and C. Allen, "Controlled Identifiers v1.0", May 2025, <<https://www.w3.org/TR/2025/REC-cid-1.0-20250515/>>.
- [DID] Sporny, M., Longley, D., Sabadello, M., Reed, D., Steele, O., and C. Allen, "Decentralized Identifiers (DIDs) v1.0", July 2022, <<https://www.w3.org/TR/2022/REC-did-core-20220719/>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/rfc/rfc7049>>.
- [SEC2] Standards for Efficient Cryptography Group, "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <<https://www.secg.org/sec2-v2.pdf>>.
- [WEBASSEMBLY] Rossberg, A., "WebAssembly Core Specification", March 2026, <<https://www.w3.org/TR/wasm-core-2>>.

6.2. Informative References

- [AT-ARCH] Newbold, B. and D. Holmgren, "Authenticated Transfer: Architecture Overview", March 2026.
- [DIDPLC] Holmgren, D., "did:plc Method Specification v0.1", May 2023, <<https://web.plc.directory/spec/v0.1/did-plc>>.
- [DIDWEB] Gribneau, C., Prorock, M., Steele, O., Terbu, O., Xu, M., and D. Zagidulin, "did:web Method Specification (Draft)", July 2024, <<https://w3c-ccg.github.io/did-method-web/>>.
- [MST] Auvolat, A. and F. Taani, "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks", October 2019, <<https://inria.hal.science/hal-02303490/document>>.

Authors' Addresses

Daniel Holmgren
Bluesky Social
Email: daniel@blueskyweb.xyz

Internet-Draft

AT Repo and Sync

March 2026

Bryan Newbold
Bluesky Social
Email: bryan@blueskyweb.xyz