Individual Submission
Internet-Draft
Intended status: Experimental
Expires: 3 September 2026

V. Iftode
Delft University of Technology

2 March 2026

Trust Computation for the TrustChain Bilateral Ledger Protocol
draft-viftode-trustchain-trust-00

Abstract

This document specifies trust computation, delegation, and key succession mechanisms for the TrustChain bilateral ledger protocol (draft-pouwelse-trustchain-01). The base protocol specifies a bilateral block structure for recording pairwise interactions but explicitly leaves trust computation out of scope. This document fills that gap by defining: (1) a canonical hash computation for cross-implementation compatibility, (2) an interaction graph constructed from half-block pairs, (3) a maximum network flow algorithm (Edmonds-Karp) over the interaction graph for Sybil-resistant trust scoring, (4) a composite trust score combining chain integrity with network flow, (5) a delegation protocol for transitive authority with budget splitting and revocation, and (6) a bilateral succession protocol for key rotation.

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 3 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.

Table of Contents

1. Introduction

The TrustChain protocol [I-D.pouwelse-trustchain] introduces a bilateral ledger data structure in which each participant maintains a personal chain of half-blocks. Pairwise interactions are recorded as linked proposal- agreement pairs, creating a tamper-evident record of bilateral history without global consensus. The original specification explicitly states that "trust calculations are out of scope" and defers the mechanism by which interaction records are converted into trust scores.

This document specifies the trust computation layer that operates over TrustChain’s bilateral ledger. The design is grounded in two reference implementations -- one in Rust and one in Python -- that together pass over 600 tests validating algorithmic correctness and cross-language compatibility.

The key contributions are:

* A canonical block hash computation using sorted-key JSON serialization and SHA-256, enabling deterministic hashing across implementations.

* An interaction graph construction where each half-block contributes 0.5 units of edge weight, and a maximum network flow (Edmonds-Karp) algorithm from a set of seed nodes that provides Sybil resistance.

* A composite trust score that combines chain integrity (hash chain verification) with network flow, enforcing a Sybil gate: if the network flow component is zero, the overall trust score is zero regardless of chain integrity.

* A delegation protocol that enables transitive authority via bilateral half-block pairs, with scope constraints, depth limits, TTL caps, budget splitting, and unilateral revocation.

* A bilateral succession protocol for key rotation that preserves identity continuity through the social graph.

1.1. Relationship to draft-pouwelse-trustchain-01

This document extends [I-D.pouwelse-trustchain] in several ways:

* *Additional block fields*: The HalfBlock format adds block_type, block_hash, and timestamp fields not present in the original TxBlock.

* *Canonical hashing*: The original draft does not specify a hash computation algorithm. This document defines one using sorted-key JSON and SHA-256.

* *Wire format*: JSON is used as the wire format for blocks, in contrast to the binary encoding implied by the original draft.

* *Additional block types*: Beyond the implicit proposal and agreement types, this document adds checkpoint, delegation, revocation, and succession block types.

* *Trust computation*: The entirety of Section 5 is new.

* *Delegation and succession*: The entirety of Section 6 and Section 7 is new.

A detailed comparison is provided in Appendix A.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document:

Half-Block: A single block in a participant's personal chain. Each half-block records one side of a bilateral interaction. Two linked half-blocks (a proposal and an agreement) constitute a complete interaction record.

Personal Chain: An append-only sequence of half-blocks maintained by a single participant, identified by that participant's public key.

Proposal: A half-block created by the initiator of an interaction. Its link_sequence_number is set to 0 (unknown) because the responder has not yet created their half-block.

Agreement: A half-block created by the responder to an interaction. Its link_sequence_number references the responder's position in their own chain, and the link_public_key matches the proposal's creator.

Seed Node: A well-known identity whose trust score is axiomatically 1.0. Seed nodes serve as sources in the network flow computation and anchor the trust graph.

Interaction Graph: A directed weighted graph where nodes are public keys and edge weights represent the volume of bilateral interactions between participants.

Sybil Gate: The rule that a participant with zero network flow from the seed set receives a trust score of zero, regardless of chain integrity. This prevents Sybil identities from accumulating trust without genuine interactions with the trusted subgraph.

Delegation: A bilateral agreement that grants one identity (the delegate) the authority to act on behalf of another (the delegator), subject to scope and depth constraints.

Succession: A bilateral agreement between an old key pair and a new key pair that transfers identity continuity, enabling key rotation.

3. Data Model

3.1. HalfBlock Format

A half-block is the fundamental data unit of TrustChain. Each half-block contains the fields defined in Table 1.

Field	Type	Size (hex)	Description
public_key	String	64 chars	Ed25519 public key of the block creator (hex-encoded, 32 bytes)
sequence_number	Integer	-	Position in the creator's personal chain (>= 1)
link_public_key	String	64 chars	Ed25519 public key of the counterparty (hex-encoded, 32 bytes)
link_sequence_number	Integer	-	Counterparty's sequence number; 0 for proposals
previous_hash	String	64 chars	SHA-256 hash of the previous block in the creator's chain (hex-encoded, 32 bytes)
signature	String	128 chars	Ed25519 signature over the block_hash (hex-encoded, 64 bytes)
block_type	String	-	One of: "proposal", "agreement", "checkpoint", "delegation", "revocation", "succession"
transaction	Object	-	Application-defined

				JSON payload
block_hash	String	64 chars		SHA-256 hash of the block (hex-encoded, 32 bytes)
timestamp	Integer	-		Block creation time (milliseconds since Unix epoch)

Table 1: HalfBlock Fields

The sequence_number MUST be greater than or equal to 1 (GENESIS_SEQ). The link_sequence_number MUST be 0 for proposal blocks and greater than or equal to 1 for agreement blocks.

3.2. Block Types

Six block types are defined:

Proposal: The initiator's half of a bilateral interaction. The link_sequence_number is 0 because the responder's block does not yet exist.

Agreement: The responder's half of a bilateral interaction. Links back to the proposal via link_public_key and link_sequence_number.

Checkpoint: A self-referencing block that records consensus state. This is the only block type where public_key MAY equal link_public_key.

Delegation: Records the granting of authority from one identity to another. Uses the proposal-agreement flow described in Section 6.

Revocation: Records the unilateral revocation of a delegation. Created only by the delegator; no bilateral agreement is required.

Succession: Records a key rotation from an old identity to a new identity. Uses the proposal-agreement flow described in Section 7.

Block type strings are case-insensitive on input but MUST be serialized as lowercase.

3.3. Hash Computation

The block hash MUST be computed using the following algorithm:

1. Construct a key-value map containing the entries listed in Table 2.

2. Sort the map entries lexicographically by key.

3. Serialize the sorted map as compact JSON with no extraneous whitespace (no spaces after colons or commas).

4. Compute the SHA-256 digest [FIPS-180-4] of the UTF-8 encoding of the JSON string.

5. Encode the 32-byte digest as lowercase hexadecimal (64 characters).

Key	Value
"block_type"	The block’s type string
"link_public_key"	The counterparty’s public key
"link_sequence_number"	The counterparty’s sequence number (integer)
"previous_hash"	Hash of the previous block in the chain
"public_key"	The creator’s public key
"sequence_number"	The creator’s sequence number (integer)
"signature"	The empty string ""
"timestamp"	The block’s timestamp (integer, milliseconds)
"transaction"	The transaction payload (JSON value)

Table 2: Hash Input Fields

The signature field MUST always be set to the empty string "" during hash computation. This ensures the hash is deterministic and can be computed before signing.

Implementations MUST use a sorted-key map (e.g., BTreeMap in Rust, OrderedDict with sorted keys in Python) to guarantee canonical key ordering across languages and platforms.

3.4. Signature Scheme

All signatures use Ed25519 [RFC8032].

To sign a block:

1. Compute the block_hash as specified in Section 3.3.

2. Compute the Ed25519 signature over the UTF-8 bytes of the hex-encoded block_hash string.

3. Encode the 64-byte signature as lowercase hexadecimal (128 characters).

To verify a block signature:

1. Decode the public_key from hexadecimal to obtain the 32-byte Ed25519 public key.

2. Decode the signature from hexadecimal to obtain the 64-byte Ed25519 signature.

3. Verify the signature over the UTF-8 bytes of the hex-encoded block_hash string.

Note that the signature is computed over the hex string representation of the block hash, not the raw hash bytes.

3.5. Block Validation Invariants

Implementations MUST validate the following invariants for every received block. A block that fails any check MUST be considered invalid.

1. *Sequence number range*: sequence_number >= 1.

2. *Link sequence number range*: link_sequence_number is either 0 or >= 1.

3. *Public key format*: public_key is exactly 64 hexadecimal characters.

4. *Signature validity*: The Ed25519 signature over the block_hash verifies against the public_key.

5. *Link public key format*: If link_public_key is non-empty, it MUST be exactly 64 hexadecimal characters.

6. *No self-signed blocks*: public_key MUST NOT equal link_public_key, unless block_type is "checkpoint".

7. *Genesis hash consistency (forward)*: If sequence_number is 1, then previous_hash MUST be the genesis hash (64 zero characters: "0000...0000").

8. *Genesis hash consistency (reverse)*: If sequence_number is not 1, then previous_hash MUST NOT be the genesis hash.

9. *Previous hash format*: If previous_hash is not the genesis hash, it MUST be exactly 64 hexadecimal characters.

10. *Future timestamp tolerance*: timestamp MUST NOT exceed the current time plus 300,000 milliseconds (5 minutes).

3.6. Constants

The following constants are defined for use throughout this specification:

Constant	Value	Description
GENESIS_HASH	64 '0' characters	Previous hash for the first block in any chain
GENESIS_SEQ	1	First valid sequence number
UNKNOWN_SEQ	0	Link sequence number for proposals
INTEGRITY_WEIGHT	0.5	Default weight for chain integrity in composite score
NETFLOW_WEIGHT	0.5	Default weight for network flow in composite score
MAX_DELEGATION_TTL	2,592,000,000 ms (30 days)	Maximum delegation time-to-live
FUTURE_TIMESTAMP_TOLERANCE	300,000 milliseconds (5 minutes)	Maximum acceptable clock drift
EPSILON	1e-10	Floating-point zero threshold

Table 3: Protocol Constants

4. Protocol Operations

4.1. Proposal-Agreement Flow

A bilateral interaction proceeds in four phases:

4.1.1. Phase 1a: Create Proposal

The initiator creates a proposal half-block:

1. Validate that counterparty_pubkey is exactly 64 hexadecimal characters.

2. Validate that counterparty_pubkey does not equal the initiator's own public key (self-proposals are forbidden).

3. Set sequence_number to one greater than the highest sequence number in the initiator's chain.

4. Set previous_hash to the block_hash of the current chain head, or GENESIS_HASH if this is the first block.

5. Set link_public_key to counterparty_pubkey.

6. Set link_sequence_number to 0 (UNKNOWN_SEQ).

7. Set block_type to "proposal".

8. Compute block_hash per Section 3.3.

9. Sign the block per the signature scheme.

10. Store the block in the local chain.

4.1.2. Phase 1b: Receive Proposal

The responder receives and validates the proposal:

1. Verify that block_type is "proposal".

2. Verify that link_public_key matches the responder's own public key.

3. Validate block invariants per Section 3.5.

4. Check for double-sign and double-countersign fraud (see Section 4.3).

5. Store the proposal.

Implementations SHOULD warn on sequence number gaps but MUST NOT reject blocks solely due to gaps, as blocks may arrive out of order during chain crawling.

4.1.3. Phase 2a: Create Agreement

The responder creates an agreement half-block:

1. Verify that the received block is a proposal.

2. Verify that link_public_key matches the responder’s own public key.

3. Verify the proposal’s signature.

4. Set sequence_number to one greater than the highest sequence number in the responder’s chain.

5. Set previous_hash to the block_hash of the current chain head, or GENESIS_HASH if this is the first block.

6. Set link_public_key to the proposal’s public_key.

7. Set link_sequence_number to the proposal’s sequence_number.

8. Set block_type to "agreement".

9. Set transaction to an exact copy of the proposal’s transaction. Note: delegation and succession agreements MAY modify the outcome field in the transaction (e.g., from "proposed" to "accepted") while keeping all other fields identical.

10. Compute block_hash per Section 3.3.

11. Sign the block.

12. Store the block.

4.1.4. Phase 2b: Receive Agreement

The initiator receives and validates the agreement:

1. Verify that block_type is "agreement".

2. Verify that link_public_key matches the initiator’s own public key.

3. Validate block invariants per Section 3.5.

4. Retrieve the linked proposal from the local store at link_sequence_number.

5. Verify counterparty identity: the agreement’s public_key MUST equal the proposal’s link_public_key.

6. Verify that the linked block is a proposal.

7. Verify that the agreement's transaction exactly matches the proposal's transaction.

8. Check for fraud and store the agreement.

4.2. Chain Crawling

Chain crawling is the mechanism by which a participant discovers the full chain of a counterparty. A participant MAY request any portion of another participant's chain by specifying a public key and a sequence number range.

When crawling:

* Implementations SHOULD validate each received block per Section 3.5.

* Sequence gaps SHOULD generate warnings but MUST NOT cause rejection of valid blocks.

* Hash chain continuity (each block's previous_hash matching the prior block's block_hash) SHOULD be verified.

4.3. Double-Spend Detection

Two forms of fraud are detectable through the bilateral ledger:

Double-Sign Fraud: Two different blocks exist with the same (public_key, sequence_number) pair but different content. This indicates the chain owner created a fork in their personal chain.

Double-Countersign Fraud: A block's link_public_key and link_sequence_number reference a proposal that already has a different agreement linked to it. This indicates the chain owner countersigned conflicting interactions.

Implementations MUST check for both forms of fraud when receiving blocks. Detected fraud MUST be recorded and SHOULD be propagated to peers during chain crawling.

When a participant has recorded double-spend fraud, their trust score is set to 0.0 (see Section 5.4).

5. Trust Computation

5.1. Interaction Graph Construction

The interaction graph is a directed weighted graph G = (V, E, w) where:

* V is the set of all public keys that have created at least one block.

* E is the set of directed edges between public keys.

* w: E -> R+ assigns a non-negative weight to each edge.

The graph is constructed as follows:

1. For each public key p in the block store: a. Retrieve all blocks in p’s chain. b. For each block b in p’s chain:

    * If b.public_key equals b.link_public_key, skip (exclude self-loops).

    * Add 0.5 to the edge weight w(b.public_key, b.link_public_key).

Each half-block contributes 0.5 units of edge weight. A complete bilateral interaction (one proposal and one agreement) contributes 1.0 total: 0.5 from the proposal’s creator and 0.5 from the agreement’s creator.

Self-loops (blocks where public_key equals link_public_key) MUST be excluded from the graph. The only block type permitting this equality is "checkpoint", and checkpoint blocks do not represent interactions between distinct parties.

5.2. Maximum Network Flow

The network flow trust score for a target identity t is computed using the Edmonds-Karp maximum flow algorithm with a super-source construction.

5.2.1. Super-Source Construction

1. Let S be the set of seed nodes.

2. Create a virtual super-source node s_0.

3. For each seed node s_i in S: a. Compute the total outflow: outflow(s_i) = sum of w(s_i, v) for all v adjacent to s_i. b. Add an edge from s_0 to s_i with capacity equal to outflow(s_i). c. Accumulate the total seed outflow: total_outflow += outflow(s_i).

4. If total_outflow is 0, the trust score for any target is 0.0.

The super-source capacity for each seed equals that seed's total outflow in the interaction graph. This ensures that a seed's influence on trust scores is proportional to its interaction volume.

5.2.2. Edmonds-Karp Algorithm

The trust score for target t is:

trust_netflow(t) = min(max_flow(s_0, t) / total_outflow, 1.0)

where max_flow(s_0, t) is computed using the Edmonds-Karp algorithm (BFS-based Ford-Fulkerson):

```
 function edmonds_karp(capacity, source, sink):
     total_flow = 0.0
     loop:
         // BFS phase: find shortest augmenting path
         parent = {}
         visited = {source}
         queue = [source]
         while queue is not empty:
             node = dequeue(queue)
             if node == sink:
                 break
             for each neighbor next with capacity[node][next] > EPSILON:
                 if next not in visited:
                     visited = visited + {next}
                     parent[next] = node
                     enqueue(queue, next)
         if sink not in parent:
             break  // no augmenting path exists
         // Find bottleneck capacity along the path
         path_flow = infinity
         node = sink
         while node in parent:
             prev = parent[node]
             path_flow = min(path_flow, capacity[prev][node])
             node = prev
         // Update residual graph
         node = sink
         while node in parent:
             prev = parent[node]
             capacity[prev][node] -= path_flow
             capacity[node][prev] += path_flow
             node = prev
         total_flow += path_flow
     return total_flow
```

The EPSILON threshold (1e-10) is used to treat near-zero capacities as zero, avoiding infinite loops due to floating-point imprecision.

If the target is a seed node, the trust score is 1.0 by definition, without running the max-flow computation.

5.2.3. Complexity Note

The Edmonds-Karp algorithm runs in O(V * E^2) time. For typical TrustChain deployments, the interaction graph is sparse and the max-flow computation is fast relative to graph construction. Implementations MAY substitute alternative max-flow algorithms (e.g., Dinic's algorithm) provided they produce identical results.

5.3. Chain Integrity Score

The chain integrity score measures the fraction of a participant's chain that passes structural verification:

```
function chain_integrity(pubkey):
    chain = get_chain(pubkey)
    if chain is empty:
        return 1.0
    total = length(chain)
    for i = 0 to total - 1:
        expected_seq = i + 1
        if chain[i].sequence_number != expected_seq:
            return i / total  // sequence gap detected
        if i == 0:
            expected_prev = GENESIS_HASH
        else:
            expected_prev = chain[i - 1].block_hash
        if chain[i].previous_hash != expected_prev:
            return i / total  // hash chain break detected
        if not verify_signature(chain[i]):
            return i / total  // invalid signature detected
    return 1.0
```

An empty chain returns 1.0 (no evidence of misbehavior). The score represents the fraction of the chain that is internally consistent before the first detected anomaly.

If a checkpoint is available and a block's sequence number falls at or before the checkpoint's recorded head for that public key, implementations MAY skip signature verification for that block (the checkpoint attests to its validity).

5.4. Score Composition

The composite trust score for a participant with public key p is:

```
function compute_trust(p):
    if double_spend_fraud_recorded(p):
        return 0.0

    integrity = chain_integrity(p)

    if seed_nodes are configured:
        netflow = trust_netflow(p)
        if netflow < EPSILON:
            return 0.0  // Sybil gate
        score = INTEGRITY_WEIGHT * integrity
              + NETFLOW_WEIGHT * netflow
        return clamp(score, 0.0, 1.0)

    return integrity
```

The Sybil gate is critical: when seed nodes are configured and the network flow score is below EPSILON (1e-10), the composite trust score MUST be 0.0 regardless of the chain integrity score. This prevents an attacker from creating isolated chains with perfect integrity to gain trust without genuine interactions with the trusted subgraph.

When no seed nodes are configured, the trust score degrades to chain integrity only. This mode provides no Sybil resistance and SHOULD only be used in closed deployments where all participants are pre-authenticated.

5.5. Incremental Graph Updates

The interaction graph MAY be cached and updated incrementally. An implementation SHOULD track, for each public key, the highest sequence number already incorporated into the graph. When recomputing trust:

1. For each public key, compare the current chain length against the previously recorded sequence number.

2. Only process new blocks (those with sequence numbers beyond the recorded position).

3. Update edge weights in the cached graph.

This avoids rescanning the entire block store on every trust query.

6. Delegation

6.1. Delegation Record

A delegation is represented by a DelegationRecord with the following fields:

Field	Type	Description
delegation_id	String	Unique ID (hex)
delegator_pubkey	String	Delegator's public key
delegate_pubkey	String	Delegate's public key
scope	Array	Allowed types; empty=all
max_depth	Integer	Sub-delegation depth cap
issued_at	Integer	Issuance time (ms epoch)
expires_at	Integer	Expiry time (ms epoch)
delegation_block_hash	String	Delegator's proposal hash
agreement_block_hash	String?	Delegate's agreement hash
parent_delegation_id	String?	Parent delegation ID
revoked	Boolean	Revocation flag
revocation_block_hash	String?	Revocation block hash

Table 4: DelegationRecord Fields

The delegation_id is computed as the SHA-256 hash of the string "delegator_pubkey:delegate_pubkey:timestamp", encoded as hexadecimal. Fields marked with "?" are nullable (may be absent). The scope field is a JSON array of interaction type strings; an empty array indicates unrestricted delegation. The max_depth field controls sub-delegation: 0 means no sub-delegation is permitted, with a maximum value of 2. Timestamps (issued_at, expires_at) are in milliseconds since the Unix epoch.

6.2. Scope and Depth Constraints

Scope: A delegation's scope is a list of interaction type strings (e.g., ["compute", "storage"]). An empty scope means the delegate is unrestricted. When creating a proposal that involves a delegated identity, the proposal's interaction_type in the transaction MUST be included in the delegate's scope (if the scope is non-empty).

Depth: The max_depth field controls sub-delegation. A value of 0 means the delegate MUST NOT further delegate. A value of 1 means the delegate may create one level of sub-delegation. The maximum permitted value is 2.

Sub-delegation constraints:

* A sub-delegation's max_depth MUST be strictly less than the parent delegation's max_depth.

* If the parent's scope is non-empty (restricted), the sub-delegation's scope MUST also be non-empty and MUST be a subset of the parent's scope. An unrestricted sub-delegation (empty scope) under a restricted parent is forbidden, as it would constitute privilege escalation.

* The parent delegation MUST be active at the time of sub-delegation.

6.3. Delegation Acceptance

Delegation follows the bilateral proposal-agreement flow:

6.3.1. Delegator Creates Proposal

1. Validate max_depth does not exceed 2.

2. Compute delegation_id = SHA-256("delegator_pubkey:delegate_pubkey:now_ms") encoded as hex.

3. Enforce sub-delegation constraints if the delegator is itself a delegate.

4. Compute expires_at = now_ms + ttl_ms, where ttl_ms MUST NOT exceed MAX_DELEGATION_TTL (2,592,000,000 milliseconds = 30 days).

5. Create a proposal half-block with:

    * block_type = "delegation"

* link_sequence_number = 0

* transaction containing:

```
{
    "interaction_type": "delegation",
    "outcome": "proposed",
    "scope": ["compute", "storage"],
    "max_depth": 1,
    "expires_at": 1735689600000,
    "delegation_id": "a1b2c3..."
}
```

6.3.2. Delegate Accepts

1. Verify block_type is "delegation".

2. Verify link_public_key matches the delegate's own public key.

3. Verify the proposal's signature.

4. Verify the transaction contains a valid delegation_id (non-empty string), expires_at (integer), and max_depth (integer).

5. Verify the current time is before expires_at.

6. Create an agreement half-block with block_type = "delegation".

7. Create and store a DelegationRecord.

6.4. Delegated Trust

When computing trust for a delegate identity:

1. If the identity has an active delegation: a. Walk the parent_delegation_id chain to find the root delegator. b. Compute the root delegator's trust via the standard algorithm (Section 5.4). c. Count the number of active delegations at the root level (N). d. The delegate's effective trust = root_trust / max(N, 1). e. Clamp to [0.0, 1.0].

2. If the identity was previously a delegate but has no active delegation (revoked or expired), the trust score is 0.0.

3. If the identity is a delegator: for each delegation (active or revoked), if the delegate has recorded double-spend fraud, the delegator's trust is 0.0 (fraud propagation).

The budget split (root_trust / N) ensures that delegating authority dilutes the delegator's trust across all active delegates, preventing trust amplification through delegation.

6.4.1. Active Delegation Predicate

A delegation is active at time now_ms if and only if:

```
active(d, now_ms) = NOT d.revoked
                    AND now_ms >= d.issued_at
                    AND now_ms < d.expires_at
```

The issued_at bound is inclusive; the expires_at bound is exclusive.

6.5. Revocation

Delegation revocation is unilateral: only the delegator signs.

1. Verify the delegation exists and the caller is the delegator.
2. Verify the delegation is not already revoked.
3. Create a half-block with:
    * block_type = "revocation"
    * link_public_key = delegate's public key
    * link_sequence_number = 0 (unilateral, no counterparty block)
    * transaction:

```
{
    "interaction_type": "revocation",
    "outcome": "revoked",
    "delegation_id": "a1b2c3..."
}
```

1. Mark the DelegationRecord as revoked, recording the revocation block hash.

Revocation takes effect immediately. The delegate's trust score becomes 0.0 upon the next trust computation.

6.6. TTL Limits

The time-to-live for any delegation MUST NOT exceed MAX_DELEGATION_TTL (2,592,000,000 milliseconds, equivalent to 30 days). Implementations MUST reject delegation proposals that specify an expires_at timestamp more than MAX_DELEGATION_TTL milliseconds after the current time.

7. Succession

7.1. Key Rotation

Succession enables key rotation while preserving identity continuity in the social graph. A succession is bilateral: both the old key and the new key must sign.

7.1.1. Old Key Creates Proposal

1. The old key MUST have at least one block in its chain (cannot succeed from an empty chain).

2. Compute succession_id = SHA-256("old_pubkey:new_pubkey:now_ms") encoded as hex.

3. Create a proposal half-block with:

    * block_type = "succession"

    * link_public_key = new key’s public key

    * link_sequence_number = 0

    * transaction:

```
{
    "interaction_type": "succession",
    "outcome": "proposed",
    "succession_id": "d4e5f6..."
}
```

7.1.2. New Key Accepts

1. Verify block_type is "succession".

2. Verify link_public_key matches the new key’s public key.

3. Verify the old key’s signature on the proposal.

4. Create an agreement half-block with block_type = "succession".

5. Store a SuccessionRecord mapping old_pubkey to new_pubkey.

7.2. Identity Resolution

After succession, the old public key resolves to the new public key. Resolution follows the succession chain:

```
function resolve_identity(pubkey):
    current = pubkey
    seen = {}
    loop:
        if current in seen:
            break  // cycle guard
        seen = seen + {current}
        if succession_record exists for current:
            current = succession_record.new_pubkey
        else:
            break
    return current
```

The cycle guard prevents infinite loops in case of malformed succession chains.

Identity resolution SHOULD be applied before trust computation so that interactions under an old key contribute to the current key's trust score.

8. Deployment Model

8.1. Sidecar Proxy

The RECOMMENDED deployment model is a sidecar process that runs alongside each agent or service. The sidecar:

1. Listens on a local HTTP proxy port (default: 8203).

2. Intercepts outbound HTTP requests from the co-located agent.

3. For each outbound request: a. Resolves the target's public key (via DNS, registry, or peer-to-peer discovery). b. Creates a proposal half-block recording the interaction. c. Forwards the request to the target. d. Receives the agreement half-block from the target's sidecar. e. Stores both half-blocks.

4. Exposes trust query endpoints for the co-located agent.

The agent sets HTTP_PROXY to the sidecar's address and requires no code changes. Trust is handled transparently at the infrastructure layer.

For HTTPS, the sidecar handles the CONNECT method and performs a TrustChain handshake before establishing the TLS tunnel. Implementations MUST NOT proxy CONNECT requests to loopback addresses, RFC 1918 private addresses, or link-local addresses to prevent server-side request forgery (SSRF).

8.2. Offline Operation

TrustChain operates without continuous network connectivity:

* Each participant's personal chain is stored locally and travels with the participant.
* Trust computation uses only locally available chain data.
* When connectivity is restored, chains synchronize via the crawling mechanism (Section 4.2).
* Delegation and succession records are stored locally and synchronized alongside block data.

Trust scores computed offline may be stale. Implementations SHOULD record the freshness of chain data and MAY reduce trust scores for data that has not been synchronized recently.

9. Security Considerations

9.1. Sybil Attacks

The primary defense against Sybil attacks is the network flow computation (Section 5.2). An attacker who creates many identities can build long chains with perfect integrity, but without genuine interactions with seed-adjacent nodes, the network flow from the seed set to the Sybil identities is zero. The Sybil gate (Section 5.4) ensures that zero network flow results in zero trust, regardless of chain integrity.

The effectiveness of this defense depends on:

* Seed node selection: Seeds MUST be well-known, stable identities with genuine interaction histories. Compromised seed nodes directly undermine Sybil resistance.

* Graph connectivity: The defense is strongest when the legitimate interaction graph is well-connected. Isolated clusters may have low network flow even for legitimate participants.

9.2. Seed Node Compromise

If a seed node is compromised, an attacker can create interactions between the seed and Sybil identities, granting them non-zero network flow. Mitigations include:

* Using multiple seed nodes so that compromising a single seed has limited impact.

* Monitoring seed node interaction patterns for anomalies.

* Rotating seed nodes periodically.

9.3. Chain Truncation

A malicious participant could present a truncated version of their chain to hide unfavorable interactions. Defenses include:

* Cross-referencing: If participant A has a block linking to participant B at sequence number 5, but B only presents a chain of length 3, this indicates potential truncation.

* Social verification: Participants who have interacted with the malicious party hold copies of their half-blocks and can detect omissions.

9.4. Temporal Attacks

The future timestamp tolerance (5 minutes) limits the ability of an attacker to pre-date blocks. However, timestamp manipulation within the tolerance window is possible. Implementations SHOULD:

* Use NTP-synchronized clocks.

* Record both the claimed timestamp and the local receipt time.

* Treat large timestamp discrepancies as a signal of potential manipulation.

9.5. Network-Level Protections

When deploying the sidecar proxy:

* The CONNECT method handler MUST block proxying to loopback, private (RFC 1918), and link-local addresses to prevent server-side request forgery (SSRF). This includes IPv4 loopback, the three RFC 1918 private ranges, IPv4 and IPv6 link-local ranges, and the IPv6 loopback address.

* HTTP request bodies SHOULD be limited in size (RECOMMENDED: 1 MiB) to prevent denial-of-service via oversized payloads.

* QUIC [RFC9000] connections SHOULD implement per-source rate limiting to prevent connection flooding.

* Private key files SHOULD be stored with restrictive permissions (e.g., 0600 on Unix systems).

9.6. Eclipse Attacks

An attacker who controls all of a participant's network peers can manipulate which chains are crawled, presenting a curated view of the graph that makes Sybil nodes appear connected to seeds. Mitigations include:

* Obtaining chain data from multiple independent sources.

* Caching previously crawled chains locally and detecting inconsistencies when the same chain is presented differently by different peers.

* Using out-of-band channels to verify seed node chain data.

9.7. Collusion

A group of legitimate high-trust nodes can collude to inflate a Sybil node's trust by creating interaction blocks with Sybil identities. The network flow computation limits the damage: the total trust allocated to the Sybil cluster is bounded by the colluding nodes' combined outflow toward the cluster. Using multiple independent seed nodes reduces the impact of any single colluding seed.

9.8. Computational Denial of Service

The Edmonds-Karp algorithm runs in O(V * E^2) time. An attacker who creates many low-weight edges (via cheap interactions) can force expensive max-flow computations. Implementations SHOULD:

* Cache the interaction graph and use incremental updates (Section 5.5).

* Set a maximum computation timeout for trust queries.

* Rate-limit interaction creation from any single identity.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

[FIPS-180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <https://csrc.nist.gov/publications/detail/fips/180/4/final>.

[I-D.pouwelse-trustchain] Pouwelse, J., "TrustChain: A Sybil-resistant scalable blockchain", Work in Progress, Internet-Draft, draft-pouwelse-trustchain-01, October 2018, <https://datatracker.ietf.org/doc/draft-pouwelse-trustchain/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

11.2. Informative References

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[TAMAS] Iftode, V.-G., "TAMAS: A Trust-Aware Multi-Agent System benchmark", arXiv 2511.05269, 2025, <https://arxiv.org/abs/2511.05269>.

Appendix A. Differences from draft-pouwelse-trustchain-01

This section summarizes the differences between this document and the original TrustChain specification [I-D.pouwelse-trustchain].

Data Model: The original TxBlock is extended with three new fields: block_type (string), block_hash (SHA-256 hex), and timestamp (milliseconds). The insert_time field from the original is removed in favor of timestamp.

Wire Format: This document specifies JSON as the wire format with canonical sorted-key serialization. The original draft implies a binary encoding.

Hash Algorithm: The original draft does not specify a hash algorithm. This document uses SHA-256 over canonical JSON.

Block Types: The original draft has an implicit two-type model (request and response, equivalent to proposal and agreement). This document adds checkpoint, delegation, revocation, and succession types.

Trust Computation: The original draft states "trust calculations are out of scope." This document specifies interaction graph construction, Edmonds-Karp maximum network flow, chain integrity scoring, composite trust scores, and the Sybil gate.

Delegation: Not present in the original. This document introduces delegated authority with scope constraints, depth limits, budget splitting, TTL caps, and unilateral revocation.

Succession: Not present in the original. This document introduces bilateral key rotation with identity resolution.

Validation: The original draft specifies five validation rules. This document extends these to ten rules, adding public key format checks, genesis hash consistency, previous hash format, and future timestamp tolerance.

Constants: This document defines an explicit constants table (Table 3) including GENESIS_HASH, GENESIS_SEQ, trust weights, delegation TTL caps, and floating-point thresholds. No such table exists in the original.

Deployment Model: This document specifies a sidecar proxy deployment model with HTTP_PROXY interception and HTTPS CONNECT tunneling. The original draft does not define a deployment model.

Appendix B. Reference Implementations

Two reference implementations exist:

Rust: https://github.com/viftode4/trustchain -- workspace with four crates (trustchain-core, trustchain-transport, trustchain-node, trustchain-wasm). 296 tests.

Python: https://github.com/viftode4/trustchain-py -- Python SDK, PyPI package trustchain-py. 310 tests.

Both implementations maintain wire compatibility through canonical JSON hashing with sorted keys, integer millisecond timestamps, and hex-encoded keys and signatures. Cross-language integration tests verify interoperability. The implementations have been evaluated using the TAMAS benchmark [TAMAS].

Appendix C. Trust Score Examples

This appendix provides worked examples of trust score computation.

C.1. Example 1: Simple Three-Node Network

Consider three participants: Alice (A), Bob (B), and Carol (C). Alice is a seed node.

Interactions: - A and B complete 2 bilateral interactions (2 half-blocks per side, 4 total). - B and C complete 1 bilateral interaction (1 half-block per side, 2 total).

Interaction graph edges: - w(A, B) = 2 * 0.5 = 1.0 (A’s 2 proposals to B) - w(B, A) = 2 * 0.5 = 1.0 (B’s 2 agreements with A) - w(B, C) = 1 * 0.5 = 0.5 (B’s 1 proposal to C) - w(C, B) = 1 * 0.5 = 0.5 (C’s 1 agreement with B)

Super-source construction: - outflow(A) = w(A, B) = 1.0 - Edge: s_0 -> A with capacity 1.0 - total_outflow = 1.0

Trust for B: - max_flow(s_0, B) = 1.0 (path: s_0 -> A -> B, bottleneck 1.0) - netflow(B) = 1.0 / 1.0 = 1.0 - Assuming perfect chain integrity: trust(B) = 0.5 * 1.0 + 0.5 * 1.0 = 1.0

Trust for C: - max_flow(s_0, C) = 0.5 (path: s_0 -> A -> B -> C, bottleneck 0.5) - netflow(C) = 0.5 / 1.0 = 0.5 - Assuming perfect chain integrity: trust(C) = 0.5 * 1.0 + 0.5 * 0.5 = 0.75

C.2. Example 2: Sybil Attack Scenario

An attacker creates 10 Sybil identities (S1..S10) and has each pair interact extensively. None of the Sybil identities interact with any seed-adjacent node.

For any Sybil identity S_i: - Chain integrity = 1.0 (perfect chains) - netflow(S_i) = 0.0 (no path from seed set) - Sybil gate applies: trust(S_i) = 0.0

C.3. Example 3: Delegation Budget Split

Alice (seed, trust 1.0) delegates to Bob and Carol.

* root_trust = trust(Alice) = 1.0
* active_delegation_count = 2
* trust(Bob) = 1.0 / 2 = 0.5
* trust(Carol) = 1.0 / 2 = 0.5

If Alice later revokes Bob’s delegation: - trust(Bob) = 0.0 (revoked delegate) - active_delegation_count = 1 - trust(Carol) = 1.0 / 1 = 1.0

Acknowledgments

The bilateral ledger concept originates from Johan Pouwelse’s work on TrustChain at Delft University of Technology. The author thanks Prof. Pouwelse for creating the foundational protocol and for his encouragement of this extension.

The reference implementations benefited from the py-ipv8 project’s BarterCast reputation system, which informed early design decisions around interaction-based trust, though the final NetFlow design diverges significantly.

Author’s Address

Vlad-George Iftode
Delft University of Technology
Email: v.g.iftode@student.tudelft.nl