ChatGPT

Capsules + STC: Formal Specification and Threat Model

Table of Contents

1. Introduction

Capsules are *authenticated data packages* containing a verifiable computation result along with associated metadata and logs. The STC (Streaming Trace Commitments) extension binds *streamed execution traces* to these capsules. This document serves as a formal reference: it defines all major objects (e.g., *CapsuleHeader*, *DA challenge*, *ManifestAnchor*, *EventChain*), formulates cryptographic security games (binding, hiding, DA soundness, policy enforcement), models adversaries and attacks, and prescribes fixes for identified gaps. References are given in footnotes. No informal claims are made without justification.

The target audience is cryptographers and security engineers; we assume familiarity with hash functions, commitments, and digital signatures. We use tight, formal language and game-based definitions following Katz–Lindell [1] [2].

2. System and Object Definitions

We model the system with the following objects:

- **TraceSpec**: A canonical description of the computation (program, inputs, constraints). The prover's trace must conform to *TraceSpec*. We represent `TraceSpec` as a concrete data structure to be hashed.

- **Statement**: A logical or algebraic assertion about the trace (e.g. "the circuit output is correct"). The prover generates a *Statement* and a proof that the trace satisfies it.

- **Proof**: A non-interactive zero-knowledge proof or SNARK attesting that the execution trace meets the Statement under the given *TraceSpec*. For soundness, we require unforgeability of proofs.

- **RowArchive**: A Merkle-hashed collection of trace chunks. We partition the execution trace into fixed-size *chunks* and compute a Merkle tree over all chunk hashes. The **row commitment** is the Merkle *root*. Individual chunks carry *Merkle proofs* for verification.

A Merkle tree committing to data: each leaf is H(data), each internal node is H(child_hashes), and the root commits to all leaves. In our design, the Merkle *root* (row_root) is stored in the header to bind all chunks. As Wikipedia explains, the root of a Merkle tree serves as a "commitment" and any leaf (chunk) can be proven to belong to that root [3].

- **ChunkHandles**: A list of references (e.g. content-addressed hashes or URIs) to all trace chunks in canonical order. We hash the list in order to compute *chunk_handles_root* for fast consistency checking.

- **Artifacts**: Auxiliary files (e.g. logs, binaries). Each artifact's integrity is checked against a committed manifest.

- **PolicyBundle**: The policy document(s) governing this capsule (e.g. access control rules). This includes an authenticated policy ID or version and optional parameters. A *policy_hash* field in the header links to the exact policy content.

- **ManifestAnchor**: A hash over system/environment information (e.g. a manifest of the binary, OS, config). This is only an attestation if the manifest is itself signed or TPM-attested. Without external attestation, the manifest is *advisory* only. We include *manifest_anchor = H(manifest)* in the header to bind environment if it can be securely obtained (otherwise only policy enforcement should rely on it).

- **EventChain**: An append-only log of events during execution. Each event record includes a timestamp and a link to the previous event. Formally, let events be a sequence $E_0, E_1, \ldots, E_n$ where each E_i contains data and a field `prev_hash=H(E_{i-1})`. We include in the header both the final event's hash ("event_chain_head") and the hash of the entire log (e.g. hash of the newline-delimited JSON log) to prevent equivocation or replay.

- **Authorship (Capsule Signature)**: A digital signature by the prover. We define a *CapsuleHeader* struct containing all binding fields (hashes of TraceSpec, Statement, chunk root, etc.) and have the prover compute $\sigma = \mathsf{Sign}_{sk}(\, H(Header)\,)$. The header includes the prover's public key or identifier and the signature. This binds the capsule to an authorized prover.

All hashes (SHA-256) are taken over *canonical encodings* of the above objects (e.g. DAG-CBOR with strict rules). Canonicalization is mandatory to prevent malleability. For example, integers must be in minimal form, map keys ordered, and floats disallowed. Practically, one specifies an unambiguous schema and encoding for each object (e.g. a `capsule-header` CBOR schema) and uses a single-bit digest function on the serialization.

These objects fit into a capsule tuple:

$$\mathrm{Capsule} = (\mathrm{Header},\ \mathrm{TraceSpec},\ \mathrm{Statement},\ \mathrm{Proof},\ \mathrm{RowArchive},\ \mathrm{Artifacts},\ \mathrm{PolicyBundle},\ \mathrm{Manifest},\ \mathrm{Events},\ \mathrm{Auth}$$

Each *CapsuleHeader* field must be cryptographically **committed** and checked by the verifier. The **binding invariants** are: - **TraceSpec binding:** header.trace_spec_hash = H(canonical(TraceSpec)). Verifier checks this matches the supplied TraceSpec. - **Statement binding:** header.statement_hash = H(canonical(Statement)). Verifier ensures the proof was generated for that Statement. - **Row commitment binding:** header.row_root = MerkleRoot(RowArchive). Verifier verifies any sampled chunk against this root. - **Chunk handles binding:** header.chunk_handles_root = H(canonical(handles_list)). The verifier loads only those chunk references that match this root and rejects duplicates. - **Backend/circuit binding:** header.backend_params_hash = H(canonical(backend_params)) for the exact prover program (e.g. AIR, FRI config, prime field). The header must specify all parameters so the verifier knows exactly *which circuit* or proof system was used. - **Policy binding:** header.policy_hash = H(policy_content). The verifier must fetch and authenticate the policy content by its ID, then check the hash. The path "policy_version" on disk is untrusted; only the committed hash matters. - **Event binding:** header.event_chain_head = H(last_event), and header.events_log_hash = H(canonical(Events log)). The log should include each event's `prev_hash`. This ensures the log is append-only and uniquely linked. - **Authorship binding:** header.authorship = σ on the header_hash. The signature keys must be certified. The verifier rejects any capsule without an *authorized* signature under the policy or ACL.

Any omission or looseness in these bindings is a security hole (e.g. missing backend parameters means the proof could be for a different computation). We assume collision-resistant hash functions and unforgeable signatures; see Section 3 below.

3. Cryptographic Preliminaries

We recall standard definitions. Let be a security parameter and `ppt` means probabilistic polynomial-time:

- **Collision-Resistant Hash Function (CRHF):** A keyed hash family where , such that for chosen by and given to the adversary, it is infeasible to find distinct with . Formally, in the *Hash-coll* experiment an adversary outputs and wins if and . We require . Katz–Lindell state "no efficient adversary can find a collision… except with negligible probability" [4] [5]. Intuitively, acts like a commitment: once you fix , you cannot find two different inputs with the same hash.

- **Commitment Scheme:** A pair of algorithms , where generates parameters and computes a commitment from a message (and randomness). A scheme is *hiding* if reveals no information about ; *binding* if it is infeasible to find a commitment that can be opened to two different messages. Formally: Katz–Lindell define hiding and binding via experiments [1], and require that for all adversaries ,
  $\Pr[\text{Hiding\_experiment} = 1] \leq 1/2 + negl(n)$ and $\Pr[\text{Binding\_experiment} = 1] \leq negl(n)$.
  Boneh–Shoup summarize "a commitment scheme … is secure if it is both hiding and binding" [2].
  In our context, we use *hash commitments*: e.g. commit = H(m // r). Collision resistance gives computational binding; if is modeled as a random oracle then hiding holds as well.

- **Digital Signature Scheme:** A triple where generates keypair , signs messages, verifies signatures. We require **existential unforgeability under chosen-message attack** (UF-CMA). Boneh–Shoup define a secure signature as one for which the adversary's chance of forging is negligible [6]. Equivalently, no efficient attacker can output a new valid pair without querying the secret key on first. We assume the signature scheme is UF-CMA-secure, e.g. ECDSA or EdDSA with collision-

resistant hash, or a NIST-approved scheme. Signatures provide origin authentication and non-repudiation.

- **Merkle Tree Commitments:** We use Merkle trees as a form of collision-resistant commitment to a list of chunks. As noted by Wikipedia, "each internal node is labeled with the cryptographic hash of the labels of its child nodes. The root of the tree is seen as a commitment and leaf nodes may be revealed and proven to be part of the original commitment" [3]. The verifier will use a Merkle *inclusion proof* (logarithmic path) to verify individual chunk integrity against the root stored in the header.

- **Pseudorandomness / Random Oracle:** We model SHA-256 as a random oracle when we need unpredictability. In particular, deriving seeds for random sampling is done by hashing a label with input; this makes the output pseudorandom to any efficient adversary who does not already know the secret seed. We assume SHA-256 is second-preimage and preimage resistant, which suffices for collision resistance and modeling as a random oracle for seed derivation.

For example, to derive an unpredictable seed we use:

```
seed = H("CAPSULE_DA_SEED_V1" // capsule_hash // relay_nonce // beacon_value // challenge_id).
```

Because (e.g. 256 bits) includes entropy from a public beacon and a nonce, this seed is effectively random to an adversary until the challenge is revealed.

4. Capsule Structure

4.1 CapsuleHeader Fields

The **CapsuleHeader** is a fixed-format record containing all critical commitments. We list its fields and definitions (all hashing is SHA-256 over canonical bytes):

- `trace_spec_hash = H(encode(TraceSpec))`. Binds the header to the actual TraceSpec.
- `statement_hash = H(encode(Statement))`. Binds to the statement being proven.
- `row_root = MerkleRoot(RowArchive)`. The root of the chunk Merkle tree.
- `chunk_handles_root = H(encode(handles_list))`. Hash of the ordered list of chunk references.
- `backend_id`: an identifier for the backend/prover engine (e.g. "Plonky2-v1"). This alone is not binding and must be supplemented by the next field.
- `backend_params_hash = H(encode(backend_params))`. A hash of all circuit/program parameters (verification key, AIR constraints, field modulus, FRI config, hash functions used, etc.). *All* parameters that affect soundness must be included so the verifier knows exactly what was computed.
- `policy_hash = H(policy_content)`. Hash of the policy document. The verifier will fetch and verify the policy by this hash.

- `manifest_anchor = H(manifest_data)`. Hash of a runtime manifest. (See Section 9.) This *anchors* environment info, but alone does not guarantee trust.
- `event_chain_head = hash of final Event in sequence`. The last event's hash value.
- `events_log_hash = H(encode(all_events_log))`. Hash of entire event log (e.g. NDJSON file). Committed to prevent omission or replay.
- `timestamp`: Capsule creation time (optional but recommended for ordering).
- `author_pubkey`: The prover's public key or identity.
- `signature = Sign_sk(HeaderFields // capsule_hash)`. The prover's signature over a digest of the above fields (and/or over the entire serialized header). We often define `capsule_hash = H(encode(HeaderFields))` and then sign that. Verification requires checking the signature under and that indeed matches the header contents.

Each field (except the signature) is covered by the signature, ensuring *all* header data is bound by the prover's key. The verifier recomputes each hash and checks consistency. Any mismatch or missing element leads to rejection.

4.2 Manifest and Policy

The **manifest** is a snapshot of the prover's execution environment (e.g. OS release, binary hashes). The manifest is not tamper-evidently signed by default; we treat `manifest_anchor` as a *best-effort* advisory. If remote attestation (TPM quote) is available, the manifest can be included inside the attested fields. Otherwise, the verifier should not rely on the manifest alone to enforce security; it may simply log it.

The **policy** (e.g. bundle of rules, ACLs) is identified by the policy ID in the header. A trusted registry should sign or hash policies. The verifier, seeing , should fetch the exact policy from an authenticated source. Only if `H(policy)` matches can it trust that policy governs this capsule. We assume the policy includes (for example) a list of authorized prover public keys, rate limits, allowed statements, and so on. The policy enforcement will be checked at verification time as described in Section 5.

4.3 Data Availability (DA) Challenge

The **DA challenge** is an object issued by a *relay service* to the prover and later to the verifier. It has structure:

```
da_challenge = {
  capsule_hash: H(encode(Header)),  // header hash from above
  challenge_id: UUID,
  relay_nonce: 32-byte random,
  beacon: {
    source: (e.g. "drand"),
    round: integer,
    value: bytes,
    proof: signature or inclusion proof
  },
  signature_relay: Sign_relay( H(all above fields) )
}
```

- is the hash of the CapsuleHeader (or an inner “payload_hash” if a nested hash is used). This ties the challenge to a specific capsule. - is a unique identifier (to prevent reuse). - is fresh random data chosen by the relay. - includes a public beacon (e.g. drand) value and proof from a given round, to ensure unpredictability. - The relay signs the entire challenge to authenticate it.

The prover receives this challenge after publishing the capsule header. It uses (and the beacon) to derive sampling positions. The relay returns to the verifier (either directly or the verifier fetches it). The verifier checks is well-formed, signed by the relay, and matches the known for the capsule.

5. Security Definitions and Games

We now state the security properties in game form:

- **Binding of Header:** The header fields constitute a non-interactive commitment to the capsule content. We require that it is infeasible for any to produce two different capsule objects and such that both have the same signed header hash. Equivalently, no adversary can find a collision under , nor forge a signature on a new header. Formally, in the *Header-Binding* experiment the adversary outputs two capsules with the same capsule_hash; it wins if they differ in any commitment (e.g. TraceSpec or Statement) but both signatures verify. Collision resistance of plus signature unforgeability imply this probability is negligible.

- **Hiding:** In our application, **all capsule data is public** once released, so *hiding* in the usual sense is not strictly needed. (No secret is committed and later revealed.) However, if any fields were confidential (e.g. a hidden mode), one would use the standard commitment hiding definition [1] . For completeness we note: A commitment is *hiding* if an adversary given cannot guess which of two messages it commits to (with advantage > negligible) [1] .

- **DA Soundness:** ensures that if a malicious prover withholds parts of the archive, the verifier will detect it. We model the DA check as follows: Suppose the prover omits a fraction of the chunks. The verifier samples random indices (without replacement). The probability that *all* sampled indices fall outside the withheld set is at most . In other words, if more than fraction is missing, will fail with high probability. Formally:

Experiment DA-Sound: Adversary chooses an integer and a subset of positions to withhold. The challenger generates a DA seed and draws indices uniformly at random from . If none of the sampled indices is in , the adversary succeeds. We require that for any fixed , is negligibly small in ; typically is set so that is negligible. This ensures that adaptive withholding beyond is caught. (This is analogous to the **proof-of-retrievability** sampling lemma.)

- **Policy Enforcement:** We require that no adversary can produce an acceptable capsule that violates the stated policy. Model this as:

Experiment Policy-Sound: Adversary outputs a capsule and an environment `Env` (including prover identity, resource usage, etc.). Let be the policy fetched by hash . If verifies (the proof is valid, the header signatures check, and DA passes) **but** evaluates as *false*, then output . The scheme is policy-sound if for all efficient . In other words, any capsule that passes verification *must* satisfy the policy. Any policy violation must cause rejection.

We summarize formal guarantees: - **Binding:** As a hash commitment, is collision-resistant. This follows Katz–Lindell's definition [5]: no should find two distinct messages hashing to the same output. Together with UF-CMA signatures, this prevents header equivocation. - **Hiding:** (Not applicable/informal here; if needed, one uses the standard hiding game [1].) - **Soundness of DA:** The sampling check with a public seed ensures that a malicious prover cannot convincingly pass audit if it withholds >f data; we rely on Chernoff/bounded-sampling analysis. (No formal citation; this is basic probability.) - **Policy Soundness:** This is a *non-cryptographic* property: the system only accepts capsules consistent with the committed policy. We rely on strong linkage of the policy hash in the header.

6. Adversarial Threat Model

We explicitly separate different adversaries (each can act *adaptively*):

- **A_{prover} (Malicious Prover):** Controls the prover software/hardware. It can choose any TraceSpec, Statement, and proof, and can corrupt or omit data before publishing. Capabilities include **statement drift** (claiming one trace when proving another), **data withholding** (publishing only some chunks), and **artifact tampering**. It may delete DA chunks after publishing. It *cannot* break hash or signature. We model A_{prover} as any PPT that attempts to produce a capsule and pass verification.

- **A_{relay} (Malicious or Faulty Relay):** The relay issues DA challenges and stores capsule hashes. A dishonest relay may attempt to **bias the randomness** (e.g. by grinding its nonce, hence the beacon requirement), **replay old challenges**, **fork** or **equivocate on event logs** (presenting different event histories to different verifiers), or **modify challenge contents**. We assume the relay has a signing key but may deviate from honest behavior. Therefore, the design requires anchors (e.g. public beacon, signed logs) to prevent relay mischief.

- **A_{DA} (Data-Availability Provider):** An untrusted storage node (or set of nodes) that holds the chunks. It may serve corrupted or fake chunks, selectively withhold, or give different answers to different verifiers. However, since each chunk has a Merkle root commitment, any corruption is detectable. The sampling check also covers selective refusal (if a chunk is missing, the verifier sees it). For added security, one could require multi-source fetching.

- **A_{policy} (Malicious Policy Registry/Author):** May publish an incorrect or permissive policy under a legitimate ID, or hijack a policy ID to a weaker policy. This could allow unauthorized provers or statements. We mitigate by authenticating policy sources (e.g. registry signatures) and by versioning. If A_{policy} changes a policy to be weaker, this could degrade security; thus we assume policies are only updated by authenticated stakeholders. Downgrade attacks (using an old policy version) are prevented by version checks in the header.

- **A_{verifier-host} (Hostile Verification Environment):** The machine/software running the verifier. It might be fed the capsule (e.g. via USB or network) by an adversary. Threats include **path traversal** (e.g. chunk reference tries to escape sandbox), **resource exhaustion** (zip bombs, huge logs), or malicious artifact (e.g. library in artifacts). We assume the verifier runs in a sandbox: it must check paths are relative to a workspace, enforce file type/size limits, and use safe parsers (e.g. limited JSON). The verifier should treat all inputs (Capsule, manifests, policy files) as untrusted until validated.

If the system cannot defend against a given (e.g. file-system exploits), it should explicitly assume that environment is *trusted*. For each property, we quantify security “against any PPT adversary , under the assumption of collision-resistant hashes and UF-CMA signatures,” as in standard cryptographic modeling.

7. Protocol Specification

The capsule creation and audit protocol proceeds in three phases: **Commit** (prover publishes capsule), **DA Challenge & Sample**, and **Verify**. We give high-level pseudocode for key algorithms.

Hash Capsule Header: Compute the digest of the header fields.

```
function hash_capsule_header(header):
    // header: an object with all fields except signature
    bytes = canonical_encode(header)       // deterministic serialization
    return SHA256(bytes)
```

This `header_hash` is used for signing and challenge-binding.

Derive DA Seed: Given the capsule hash and challenge, derive a random seed for sampling.

```
function derive_da_seed(capsule_hash, relay_nonce, beacon_value, challenge_id):
    label = ASCII("CAPSULE_DA_SEED_V1")
    seed_input = label || capsule_hash || relay_nonce || beacon_value ||
challenge_id
    return SHA256(seed_input)
```

This seed is unpredictable to the prover when it first publishes the capsule (since `relay_nonce` and `beacon_value` arrive later) and unpredictable to the relay once the beacon is fixed.

Select Random Indices: Generate distinct indices in `[0 .. N-1]` from the seed.

```
function select_indices(seed, N, k):
    indices = empty_set
    ctr = 0
    while |indices| < k:
        block = SHA256(seed || toBytes(ctr))  // expand seed
        ctr += 1
        for i in 0 .. 31:                     // process 32-bit chunks of block
            idx = (block >> (i*8)) mod N
            if idx not in indices:
                indices.add(idx)
                if |indices| == k:
```

```
                    break
    return sorted(indices)
```

We use SHA-256 in counter mode for expansion. We avoid simple modulo bias by rejecting duplicates; for large N we could use more sophisticated unbiased sampling.

Commit-Then-Challenge DA Audit: The verifier's procedure is:

```
function audit_da(capsule, da_challenge, sample_count):
    H = capsule.header_hash   // already computed by prover
    // Verify challenge is for this capsule:
    if da_challenge.capsule_hash != H:
        return REJECT
    // Verify relay signature on challenge:
    if not Verify_relay( da_challenge ):
        return REJECT

    // Derive random seed and sample indices:
    seed = derive_da_seed(H, da_challenge.relay_nonce,
da_challenge.beacon.value, da_challenge.challenge_id)
    indices = select_indices(seed, N = capsule.RowArchive.num_chunks, k =
sample_count)

    // Check each sampled chunk:
    for idx in indices:
        chunk, merkle_proof = capsule.RowArchive.get_chunk_and_proof(idx)
        if not verify_merkle_proof(chunk, merkle_proof,
capsule.header.row_root):
            return REJECT     // data missing or corrupted
    return ACCEPT
```

The prover in turn does the same calculation of `indices`, but instead of checking, it just fetches those chunks and returns them to the verifier (who already has the Merkle root). If the prover withheld any chunk , it will fail here with high probability.

All protocols assume **no hash cycles**. In particular, we ensure `capsule_hash = H(header)` is computed *before* any DA fields are added, so the header commitment is independent of the challenge. The steps are:

1. **Prover Commit:** The prover assembles , signs it to get and publishes the capsule (Header + data + signature). Importantly, *no part of the challenge is included in the capsule when hashing*, avoiding circular dependency.
2. **Relay Challenge:** The prover (or verifier) contacts the relay with to get a signed DA challenge. The relay returns with a fresh nonce and beacon.

3. **Prover Samples:** The prover computes and samples the indicated chunks from its stored RowArchive. It sends the chunks and proofs to the verifier (or the chunks might be already stored and the relay merely instructs the verifier where to fetch them).
4. **Verifier Verify:** The verifier checks: (a) header signature is valid and fields match the capsule; (b) hashes of TraceSpec, Statement etc. match the header; (c) proof verifies for the Statement; (d) DA challenge is authentic and capsule_hash matches; (e) DA audit as above; (f) policy and event log consistency; (g) authorship ACL, etc. If all checks pass, output ACCEPT.

8. Attacks and Mitigations

Below we describe concrete attacks and countermeasures:

- **Adaptive Chunk Withholding:** A malicious prover might *only upload some chunks* and expect to evade detection. The remedy is the **commit-then-challenge** protocol above. Because the seed incorporates an unpredictable beacon and nonce, the prover cannot know in advance which chunks will be sampled. Omitting even a single chunk has ≤(1–f)^k chance of escaping k random checks. Increasing makes this negligible. We require a *non-malleable challenge*: hence the signed relay nonce and public beacon [7] [8]. (Without the beacon, a malicious relay or verifier acting as RNG could bias by grinding, so we explicitly tie to an external randomness source.) In summary: **fix** by using an external beacon and deriving the seed via SHA-256 as above.

- **Statement Drift:** The prover could "prove one statement and label it as another." For example, the trace might satisfy Statement , but the header and verifier expect . We prevent this by cryptographic binding: the header contains hashes of both *TraceSpec* and *Statement*, and the verifier re-checks that the actual proof was constructed for exactly those values. In other words, the proof verification algorithm takes and the actual (or their hashes) as input, so any mismatch causes failure. **Fix:** Always include and hashes in and verify consistency [9].

- **Event Log Equivocation:** If the relay or prover forks the event log (showing different histories), it undermines audit. To prevent this, each event record must include the hash of the previous one. We enforce *chain hashing*: for events , compute where is the hash of . The final in the header and the hash of the entire event stream commits to the sequence. The verifier checks that each matches . If an adversary tries to insert or delete events, this breaks the hash chain and is detected. We *always* require an append-only log format (e.g. JSONL) with each event linked.

- **Multi-Backend Ambiguity:** If the system supports multiple proof backends, one must not confuse them. A capsule must explicitly bind the proof to the correct backend parameters. **Fix:** Don't rely on a simple name or version; the header must include a hash of the exact proving key or circuit parameters (field, modulus, AIR constraints, hash function). That way, the verifier knows *exactly* which circuit was used, and can reject a proof from any other circuit. In practice this means adding , , and any generator matrices into the committed data [10].

- **Policy Downgrade or Authentication Failure:** A malicious policy provider might trick the system by providing a weak policy under a trusted ID. **Fix:** Policies must be stored in an authenticated registry or repository. The header only stores , so the verifier must fetch the policy from a signed registry or by out-of-band agreement. We assume a PKI or trusted consortium signs policy versions. The verifier also checks that the prover's public key (in) is *authorized* by the policy's ACL. E.g. the policy may list

allowed prover key IDs. If not matched, reject. In short: **trust the hash, not the file path** (compare header hash to actual policy bytes).

- **Manifest Tampering:** If the prover supplies a manifest it altered after execution, it could misrepresent the environment. We only include in the header after computing it. But if the manifest is not itself attested, this provides limited security: a user could change the manifest file without breaking hashes (in principle, a collision, but we assume CRHF). In practice, one should store the manifest out-of-band (e.g. on a trusted filesystem or via a quote from a TPM). We note this as an assumption: *manifest contents come from a trusted process*. If not, the manifest field can be ignored or flagged.

- **Verifier Host Attacks:** If the verification software is compromised, all bets are off. We assume the host enforces file size and path checks. For example, if a chunk handle in has an absolute path, the verifier should reject it. Implementers must run the verifier in a sandbox or container. Verifier code should validate all inputs (e.g. reject overly large JSON manifest). This is a **deployment issue** more than a protocol flaw; the specification requires that verifiers be hardened (e.g. no "zip bombs").

- **Missing Randomness:** Earlier versions omitted the beacon or mis-derived the seed. We fix this by the protocol above. Avoid common pitfalls like truncating the seed to 64 bits or excluding the beacon. In our design we use the full 256-bit digest as a seed, and include a public beacon to remove trust in the relay. Katz–Lindell and others warn against naive RNG usage; indeed, blockchain designs also require unpredictability [11] [8].

In each case, we recommend explicit fixes. For example, adding an unpredictable beacon removes *last-revealer bias* [11], and hashing the backend parameters prevents substitution of one proving scheme for another. All such recommendations are present in the schema: missing fields are to be considered critical vulnerabilities.

9. Additional Considerations

Beyond the core protocol, we briefly address these topics:

- **Canonical Encoding:** All structured data (TraceSpec, Handles list, etc.) must use a *deterministic*, canonical encoding (e.g. DAG-CBOR) before hashing. This avoids malleability (e.g. JSON white space or map order changes). One must forbid floating-point or indefinite-length constructs. We advise implementing a cross-language test suite: given a JSON or YAML spec, its canonical encoding and SHA-256 digest should be platform-independent.

- **Verifier Sandboxing:** The verifier, on processing a capsule, should unpack any archive (Artfacts) into a restricted directory. It must disallow absolute or ".." paths. If verifying policy or manifest content, it should treat them as untrusted data. All signature verification and hash checks should be done before evaluating any code or logic that the capsule might contain.

- **ACL Logic:** The policy bundle may include access-control lists (ACLs). For example, a policy might say "only host 123 may verify this capsule". The verifier must enforce these ACLs. Typically, the policy will list allowed prover or owner identities. The header's `author_pubkey` is checked against the policy

ACL. If not allowed, the capsule is rejected (even if cryptographically valid). This binding of *who* to *what* is part of the policy enforcement property above.

- **Malicious Policy Outputs:** In adversarial environments, the verifier's trust in the policy itself must be carefully managed. We assume policies are published via an authenticated channel (e.g. HTTPS, code-signed documents). If a policy is updated, older capsules with old policy hashes can still verify against the new policy if backward-compatible; otherwise they fail the hash check.

- **Performance and Parameters:** Sampling parameter (number of chunks checked) must be chosen based on the maximum tolerable data-loss fraction . If an adversary withholds 50% of chunks, sampling 50 randomly has failure probability in catching all, which is about . In general, set so that . For example, to catch 1% of missing chunks with 99.999% confidence, one needs ~1,150 samples. These parameters should be published in policy or system docs.

- **Recovery and Retries:** If a DA audit fails (some chunk missing or invalid), the verifier should *not* trust the capsule. The user could retry fetching data from alternative mirrors, but no formal fallback exists: failure = reject.

In sum, *Capsules + STC* relies on well-known cryptographic primitives (hashes, signatures, Merkle trees) and on classical commit–challenge sampling. Our construction is essentially a specialized commitment scheme with extra fields. By following the binding checklist above and the protocol steps, one achieves **authenticity** (signed header), **integrity** (hash chains), **binding** (CRHF), **data availability** (random audit), and **policy-compliance** (authenticated policies).

References: Katz–Lindell's textbook [1] [5] and Boneh–Shoup [2] [6] provide formal definitions of commitments and signatures. The Merkle tree model is standard [3], and the use of public randomness beacons is motivated by systems like drand [8]. This document encapsulates those principles into a cohesive specification tailored to Capsules + STC.

---

[1] [4] [5] Introduction_to_Modern_Cryptography,_3rd_Edition_by_Jonathan_Katz_[bibis.ir].pdf
file://file-Ad5YUoBWBJwdBRBMyteXfe

[2] [6] graduatecourseincrypto.pdf
file://file-MvbUmV3EXLBJyXqn3vzYTL

[3] Merkle tree - Wikipedia
https://en.wikipedia.org/wiki/Merkle_tree

[7] [9] [10] [11] help_v2.txt
file://file_00000000a54471f5a7fe3763f2e07570

[8] drand Explained | drand
https://docs.drand.love/about/