Confidential Telemetry at Line Rate: MA-ABE and Streaming ZK with Nova IVC and a Groth16 Decider

Johan B

2025-09-05

Abstract

I present a protocol for privacy-preserving metric reporting that combines Multi-Authority Attribute-Based Encryption (MA-ABE) with a *streaming* zero-knowledge design based on folding SNARKs: Nova for per-update folding and a Groth16 *decider* for a single succinct window proof. A provider maintains compact public state and produces *one proof per window* (for example, every 1–5 seconds) attesting that all included records satisfy the declared policy. Each record is committed and encrypted with associated data that binds provider, window, sequence, timestamp, schema, and a policy digest. To make per-record encryption practical and context-bound, MA-ABE encrypts a per-window seed `Z`; a key-encryption key derived from `Z` and the accepted header hash wraps the AEAD key `K`, so ciphertexts and keys are useless off-context and only authorized readers can decrypt. *Line rate* refers to ingestion and collection cadence; the succinct proof cadence is per window, and deployments may choose to buffer/reorder within a window while keeping all bindings deterministic.

Validators on a BFT chain verify the single full-window decider proof and the byte-exact header. Only if verification succeeds is the provider allowed to attach the off-chain batch (for example, an IPFS CID) and wrapped key material; attempts to attach prior to acceptance are rejected by design. The chain pins a canonical decider verification key; providers choose policy parameters but not the circuit.

We centralize canonical encodings, domain separation, Merkle orientation and defaults, nonce discipline, and key-binding rules; provide algorithms for streaming updates and two-phase window acceptance (commit then finalize); and specify deterministic validator logic for safety and ordering under BFT. The result complements existing collectors: transport remains familiar, while confidentiality is preserved end-to-end and correctness is elevated from trust in a collector to a succinct, third-party-verifiable proof per window.

All normative definitions and proofs of correctness are consolidated in the appendices (see Appendix, Algorithms A.1–A.7).

1 Introduction

Network monitoring is essential to operating production systems. Operators deploy tools such as Zabbix, Nagios, LibreNMS, and Prometheus to collect high-cardinality telemetry, trigger alerts, and investigate incidents. Agents and exporters report CPU load, latency, error rates, queue depths, and countless application metrics into a central collector that supports dashboards and SLOs. The model is effective and widely adopted, yet it concentrates trust and visibility in the collector and its administrators: anyone with sufficient backend access can see everything, and endpoints can still submit falsified values that the collector dutifully records because correctness is not cryptographically enforced. In multi-domain environments—such as between suppliers and regulators, or across isolated network segments—sharing telemetry across domains requires tunnels, broker services, or manual procedures that either increase risk or erode fidelity. Mainstream systems typically secure telemetry with transport-level encryption (and sometimes encryption at rest), but they do not bind values to their context or provide verifiable, window-level guarantees about policy compliance. At the same time, modern protocols reduce passive visibility in the network, making it less feasible to compensate with on-path inspection.

This paper addresses three practical shortcomings of current monitoring: confidentiality of raw values, verifiable correctness of policy compliance, and a robust cross-domain transport that avoids trusted intermediaries. We keep the familiar workflow—agents continue to emit numeric metrics at line rate—but

replace unverified trust with succinct cryptographic checks and selective decryption. Concretely, each metric vector is committed with a leaf digest that binds the provider, window, sequence, timestamp, schema, and a canonical policy digest, and is then encrypted for authorized readers. A per-record step enforces per-metric and cross-metric predicates and updates a rolling Merkle root (optionally, deployments may also maintain aggregates for observability); a folding SNARK (Nova) composes these steps across the window; and a Groth16 decider produces a small, millisecond-verifiable proof for the whole window. On chain, validators verify exactly one proof and the header commitment per window under BFT consensus. Acceptance acts as a gate: only after a window is accepted may the provider attach the off-chain batch reference and wrapped key material. Key wrapping is bound to the accepted header via a derivation keyed by an ABE-protected seed, so ciphertexts and keys are useless outside their attested context.

My contribution is a specification that complements existing collection pipelines rather than replacing them. On the data plane, producers emit the same metrics they emit today; on the control plane, headers, proofs, and bindings make those metrics verifiably correct for their declared policy before they qualify for storage or attention. Authorized readers can later decrypt and recompute commitments to check equality with the accepted root; anyone can perform a non-decrypting availability check for index completeness; and challengers can present a violating record and Merkle path to trigger slashing. The result is a monitoring workflow that preserves operational ergonomics while adding cryptographic safeguards appropriate to encrypted, multi-party networks.

2 Background and Threat Model

2.1 Network Monitoring Systems

Network monitoring systems serve as essential infrastructures for collecting, analyzing, and responding to performance and security metrics. By examining indicators such as bandwidth usage, latency, error rates, and system resource consumption, these systems allow administrators to identify bottlenecks, detect intrusions, and make informed decisions on capacity planning. The significance of monitoring rises in parallel with the growing complexity of modern networks, where organizations span data centers, cloud platforms, and edge deployments. Without continuous visibility into key metrics, subtle faults or security breaches may go undetected until they cause substantial disruptions, leading to operational losses and reputational damage.

In contemporary deployments, telemetry is not solely metric oriented. Operators increasingly treat metrics, logs, traces, and discrete events as a single fabric, collected by lightweight agents or SDKs and sent through a small collection tier that normalizes, batches, and routes data to multiple backends. A small amount of logic at collection time has clear benefits: sampling and rate limiting can be applied where bursts originate, sensitive fields can be redacted or hashed before they ever reach storage, and unbounded identifiers can be mapped into stable categories so downstream systems remain fast and predictable. Because topologies change quickly in containerized and multi-cloud environments, keeping the collection path stateless and declaratively configured lets instrumentation evolve without coupling it to storage or query engines.

A common pattern is to run an edge collector on each node or pod, optionally paired with a gateway tier at domain boundaries. The edge tier provides immediate backpressure and normalization close to workloads, while the gateway tier handles tenancy, routing, and export to long-term stores. This separation preserves a stable, policy-aware interface between organizational domains and reduces the impact of misbehaving sources, since normalization and drop rules are enforced before samples enter shared infrastructure. It also provides explicit control points for enforcing cost and latency budgets under bursty load.

2.2 Centralized Models and Their Limitations

Many organizations implement monitoring through a centralized architecture. A single data collector or tightly coupled cluster ingests and consolidates metrics, providing a unified view of network status. Popular tools such as Nagios, Zabbix, and Prometheus illustrate this model by aggregating metrics from agents or by polling monitored hosts, then storing results in a central database. Administrators benefit from a simplified setup in which a single interface offers dashboards and alerts for the entire environment. This approach works effectively for smaller infrastructures, especially when the central node's load remains manageable.

As networks grow in size, this centralized design becomes a significant bottleneck. The surge in endpoints and monitored services produces data volumes that can easily overwhelm a lone collector or cluster, forcing operators to reduce data granularity or scale hardware at high cost. Even well-provisioned collectors may struggle under bursts of incoming data that need real-time processing. Moreover, the reliance on a single repository creates a clear point of failure. If the central node is compromised, misconfigured, or taken offline, visibility into the entire network is lost, exposing operators to extended periods without visibility. Attackers who breach this node can manipulate or delete logs and metrics, concealing malicious activities. The critical nature of the central repository also makes it a prime target for cyberattacks, since control over this data effectively grants control over the organization's perception of its own infrastructure.

The first scaling pressure typically comes from label cardinality rather than raw message count. Labels that encode user identifiers, request IDs, or rapidly rotating container hashes inflate the number of distinct time series a backend must keep in memory, increasing ingestion latency and query tail-latency. Horizontal scaling helps, but head-of-line blocking at aggregation points and write-ahead-log backfill during incident spikes are common failure modes. To keep central clusters healthy, operators insert a transformation stage in front of storage to cap or coarsen label spaces and to aggregate high-rate signals into policy-relevant summaries before they overload hot shards.

When a single cluster no longer suffices, two patterns are prevalent. Fan-in from many agents or edge collectors into a long-term store reduces edge state but concentrates risk at the ingest tier; federation of multiple stores behind a query layer spreads ingestion but turns some queries into distributed plans whose latency is dictated by the slowest shard. Neither pattern obviates the need to bound cardinality early, and neither addresses the trust problem that arises when transport, storage, policy evaluation, and administrative privilege are combined in one system.

2.3 Security and Privacy Shortcomings

Traditional architectures rarely incorporate advanced cryptographic methods beyond basic encryption of data in transit. Although such encryption protects against simple eavesdropping, it does not guarantee that reported metrics are authentic or tamper-proof. A compromised or malicious endpoint can still upload falsified telemetry, and the central node typically lacks mechanisms to verify correctness without decrypting raw values. Administrators must trust the reporting entity unless they manually correlate metrics with other indicators. This assumption of trust is problematic for large or heterogeneous infrastructures, where different teams or organizations might have conflicting incentives, or where insiders could exploit privileged access.

Coarse access control further exacerbates the problem. Centralized repositories often allow individuals with administrative credentials to view all metrics without restriction, risking privacy violations. Such data can reveal an organization's footprint, application flows, and even user activity. In the absence of robust cryptographic controls—such as commitments for authenticity or zero-knowledge proofs that the data meets validity constraints—attackers or dishonest insiders could stealthily alter or fabricate metrics. Similarly, no mechanism enforces that each submitted metric has been collectively signed or validated by multiple parties, leaving the single trusted collector in a position to become a system-wide point of vulnerability.

Modern transports compound these issues. With contemporary protocols, middleboxes and taps no longer see enough stable structure to perform reliable policy checks, and attempts to regain visibility by centrally terminating encryption recreate the very trust concentration operators wish to avoid. The practical alternative is to move assurance to producers and make the artifacts portable: bind each record to its context so ciphertexts cannot be replayed or relabeled across windows or schemas, prove policy compliance across a clearly delimited window, and only then allow storage attachment. In this model, selective disclosure becomes a consequence of decryption rights, while intermediaries can still verify correctness from compact, public artifacts without learning plaintexts.

2.4 Adversary and Assumptions

We assume compromised or misconfigured endpoints that may emit falsified metrics; honest-but-curious infrastructure operators and observers; and network partitions or censorship attempts. Confidentiality targets prevent unauthorized observers from learning plaintext metrics; correctness targets ensure that accepted

windows satisfy the policy predicate for all indices. We trust the cryptographic primitives (Poseidon, AEAD, pairings) and the BFT consensus for ordering and finality.

A realistic adversary also exploits operational factors as much as cryptography. By emitting measurements with rapidly rotating or unbounded labels, an attacker can create a cardinality surge that degrades ingestion and query paths even when numeric values appear plausible. By drifting timestamps or sequence numbers within tolerated ranges, they can aim to create gaps or overlaps that complicate reconciliation. These behaviors are mitigated when normalization occurs at collection time, when sequence and window identifiers are enforced monotonically, and when window acceptance is tied to a proof over exactly the records that were committed. Within these constraints, the security goal is twofold: unauthorized parties and intermediaries should learn at most the small set of public fields necessary for routing, ordering, and verification, never the metric values themselves; and every record included in an accepted window should satisfy the published predicate, with any attempt to replay, relabel, or attach data before acceptance detected and rejected. The architecture separates transport and storage from trust by making acceptance and attachment contingent on compact, third-party verifiable artifacts rather than on the good behavior of a central repository.

3 Cryptographic Preliminaries

This section fixes the building blocks and conventions that make independently written implementations derive the same bytes and public inputs, both outside and inside circuits. The prose below is explanatory. The authoritative reference for byte-accurate rules is Algorithm A.1.

Constants (normative, brief). Two-domain cryptography (normative). The protocol uses *two* cryptographic domains: (i) MA-ABE uses a pairing-friendly curve (BLS12-381) for attribute-based encryption of the per-window seed Z (off-circuit); (ii) the streaming proof system (Nova folding and Groth16 decider) operates over a *cycle-friendly SNARK field* $\mathbb{F}$ (normative below). All in-circuit hashing (Poseidon), leaf commitments, Merkle roots, and public inputs of the decider are elements of $\mathbb{F}$. Cross-domain bindings (headers, acceptance hash, KEK derivation) are done via byte hashes (BLAKE3/HKDF) and are therefore independent of the ABE curve.

SNARK field (normative). All proofs use a cycle-friendly SNARK field $\mathbb{F}$ suitable for Nova (e.g., the Pasta cycle). We fix $\mathbb{F} = \mathrm{Fr}(\text{Pallas})$ for Poseidon, leaf commitments, Merkle roots, and Groth16 public inputs. Outside the circuit we hash with BLAKE3-256. Poseidon is instantiated over $\mathbb{F}$ with width 3, rate 2, capacity 1, S-box exponent 5, and round counts 8 (full) and 56 (partial). *Round constants and the MDS matrix are derived* ***deterministically*** *by the pinned generator in Algorithm A.2 from the public seed*

`POSEIDON_PALLAS_t3_alpha5_RF8_RP56:v1:maabe-streaming`

and *are part of the protocol version.* Implementations **must** use this generator and seed; the resulting constants **must** be shipped with test vectors. Exact encodings live in Algorithm A.1.

Canonical helpers and encodings

We centralize all byte and field conventions in Algorithm A.1. Integers are encoded at fixed width in big-endian. Field elements use canonical 32-byte encodings, and byte-to-field mapping rejects out-of-range integers. Arbitrary byte strings that may exceed a field element are absorbed as 31-byte limbs so Poseidon sees an unambiguous sequence. Domain-separation strings are pinned ASCII tokens for leaves, internal nodes, the step relation, attestation, and policy serialization; their exact spellings and lengths are absorbed into Poseidon so structure is part of the hash.

Public-input packing (normative). Groth16 public inputs are field elements. For any 32-byte value (e.g., pid and $\mathrm{H}_{\mathrm{step}}$), we *pack* it into two field elements by splitting into two 16-byte big-endian limbs and mapping each limb with $I2F_{\mathrm{be}}^{\mathrm{var}}$. The decider circuit therefore takes exactly **13** public inputs in a fixed order:

$$(\mathsf{pid}^{(0)}, \mathsf{pid}^{(1)},\ \mathsf{wid},\ \mathsf{seq}_{\min},\ \mathsf{seq}_{\max},\ R,\ t_{\min},\ t_{\max},\ N_{\mathrm{win}},\ \mathsf{schemaID},\ \mathrm{Hash}(\theta),\ \mathrm{H}_{\mathrm{step}}^{(0)},\ \mathrm{H}_{\mathrm{step}}^{(1)}),$$

where $x^{(0)}$ is the first (high) 16-byte limb and $x^{(1)}$ the second (low) 16-byte limb. Validators *derive* this vector from SERHDR using Algorithm A.1 and **byte-bind** it during finalization (Algorithm A.4).

Nonce discipline and AEAD key uniqueness (normative). AEAD nonces follow the 96-bit rule chosen for circuit-simple uniqueness: in the single-agent profile the nonce encodes the window-local *record order index*; in the multi-agent profile it encodes the agent id followed by the record order index, with a deployment-enforced bound on window length. *For each accepted window, provider **MUST** ensure:* (i) the per-window AEAD key K is **unique** to that window (fresh from a CSPRNG, or via a deterministic KDF bound to the accepted header; see Algorithm A.1); (ii) at most one ciphertext exists for any $(\texttt{agent\_id}, i)$ pair under that K. Duplicate indices **must** be rejected before encryption.

The step relation is bound by a byte-level digest `H_step` that commits to the relaxed R1CS encoding of the compiled matrices using a dedicated domain separator. The decider hardcodes this digest (as two field elements corresponding to the two 16-byte limbs) and enforces equality to public inputs $\mathrm{H}^{(0)}_{\mathrm{step}}, \mathrm{H}^{(1)}_{\mathrm{step}}$. The Nova-folded instance exposes the same pair; any change in the compiled step relation invalidates proofs without touching higher-level code.

Window headers are serialized as a single byte string, `SERHDR`, that includes identities, sequence and timestamp bounds, the Merkle root, `schemaID`, the policy digest, `decider_id`, `H_step`, and (in the extended form) the declared records payload size and its digest, in a fixed order and width. The attestation hash `HdrHash` binds `SERHDR` to the chain and the finalized acceptance height; validators compute it on acceptance and it becomes the anchor for key material. These layouts—field widths, order, and domain separators—are normative and checked byte-for-byte during validation.

AEAD is fixed per deployment; the default is ChaCha20-Poly1305 with 96-bit nonces and standard limits. Each record plaintext (metrics vector, per-record randomness, sequence, and timestamp) is encrypted under a per-window key `K` with the AAD above. To make keys portable yet context-bound, MA-ABE encrypts a per-window seed `Z`; a KEK derived via HKDF-SHA256 with a fixed salt and an info string that includes the accepted `HdrHash` wraps the per-window AEAD key. The attachment posted after acceptance carries the content identifier for the batch and the two wrapped artifacts; authorized readers recover $\texttt{Z} \to \mathrm{KEK} \to \texttt{K}$, decrypt each record with the same AAD, recompute leaves, and aggregate to the published root. Exact byte strings and encodings are normative in Algorithm A.1.

Rationale. Implementations most often diverge at the boundaries—byte width and endianness, limb splitting, domain separation, Merkle orientation, header packing, and the exact AAD for AEAD. Centralizing these rules in Algorithm A.1 prevents unintentional divergence and lets the rest of the system speak in higher-level terms without restating encodings. Any mismatch in these byte-exact conventions is observable either as a failed decider verification or as a header/leaf/root inequality during retrieval.

4 Formal Model

The provider maintains a compact public state for each window and a private accumulator that captures every checked update. Public state consists of the Merkle root over per-record commitments, window-local bounds for timestamps, sequence bounds, the window length, and the identifiers for provider, window, schema, and policy digest. Deployments may *optionally* maintain per-field aggregates for observability (min, max, sum, sum of squares) outside the succinct statement. The private state is a Nova accumulator over the same step relation that is later summarized by a succinct Groth16 decider proof. Canonical encodings, domain separation, Merkle orientation and defaults, and header packing are fixed in Algorithm A.1; the streaming step is fixed in Algorithm A.3.

Scope. The step relation is deliberately small and regular so that each record contributes a constant, predictable amount of work. The circuit checks ranges for each metric component and for the timestamp; authenticates the policy by hashing its canonical byte serialization and comparing it to the public policy digest; enforces strictly increasing sequence numbers (allowing gaps) and a window-local *record order index* that increments by one per processed record; verifies Merkle membership for the old leaf at this index; recomputes the parent path to a new root with the fresh commitment; evaluates the policy predicate over the record; updates timestamp bounds; and folds the transition into the Nova accumulator. Deployments may additionally require non-decreasing timestamps; when enabled, the step enforces that as well. (Aggregates

may be maintained off-circuit or by a variant circuit; they are not part of the canonical header or decider inputs.)

Purpose. At window end, a single succinct proof certifies that every included record satisfied the declared policy and contributed to the published root, without revealing plaintexts. The step relation binds values to context inside the leaf commitment, binds policy and schema through public inputs, and enables later recomputation by authorized readers who decrypt records and re-derive leaves.

Mechanism. For an incoming metrics vector with timestamp and sequence, the provider computes a leaf commitment with per-record randomness and runs one step of Algorithm A.3. Outside the circuit, the provider encrypts the record under a per-window key using AEAD with the canonical AAD that binds provider, window, sequence, timestamp, schema id, policy digest, and decider id, and a 96-bit nonce derived from the window-local record order index (and optionally an agent id in multi-agent deployments). The tuple of leaf digest, ciphertext, sequence, and timestamp is appended to the off-chain batch buffer. The Nova accumulator captures the exact sequence of checked transitions so that the decider can later attest the entire window with one proof.

Usage. Agents emit vectors at line rate; the provider applies the step for each record and accumulates tuples locally. When the window ends, the provider finalizes the public state and produces a succinct decider proof over the Nova accumulator; both are consumed by the on-chain protocol to gate storage and key attachment. Authorized readers later decrypt records, recompute leaves, and aggregate to the root to check equality with the accepted header. The byte-exact definitions that make these steps interoperable are centralized in Algorithm A.1; the streaming relation itself is Algorithm A.3.

5 Protocol

The on-chain protocol enforces a strict, two-phase lifecycle—*commit* then *finalize*—followed by a post-acceptance *attach*. Commit exposes a byte-exact header (`SERHDR`) but no storage or keys; finalize verifies one succinct proof for the whole window and computes the acceptance hash; attach is permitted only after acceptance and binds key material to the accepted header. Deterministic validator logic enforces ordering, parameter binding, and byte equality throughout. Canonical message layouts and hashes are defined in Algorithm A.1; the lifecycle is captured in Algorithms A.4, A.5, and A.8.

Flow. At window end the provider serializes `SERHDR` with provider id, window id, sequence bounds, window length, timestamp bounds, Merkle root, schema id, policy digest, decider id, and `H_step`, and broadcasts `CommitWindow`. Validators check monotonicity of window identifiers and sequences, record the pending header, and accept no storage at this stage. When the decider proof is ready, the provider broadcasts `FinalizeWindow` with public inputs that must *byte-bind* to SERHDR via the **canonical packing** in Algorithm A.1. Validators verify the Groth16 proof under the canonical verification key selected by the decider id; on success they mark the window *Accepted* at a finalized height and compute `HdrHash` over the extended header bytes. Only after acceptance may the provider attach storage and key artifacts by calling `AttachBatch` with the batch CID and the wrapped key material.

Purpose. The protocol gates storage and key release on a succinct, third-party-verifiable statement that the entire window satisfied the declared policy, while preserving confidentiality of plaintext metrics. Headers are immutable, byte-exact commitments to the public state; proofs are succinct checks of the private computation; attachments are context-bound via `HdrHash` so they are useless off-context.

Mechanism. *Commit.* Validators accept a pending header only if identifiers and sequence bounds are strictly monotone with the provider's last accepted window and the declared length matches the number of records committed (not the sequence span). No batch reference or key material is accepted at this stage. *Finalize.* The decider proof is checked against the canonical verification key for the decider id; validators

compute the public-input vector from SERHDR (Algorithm A.1) and require equality to the provided public inputs. On success, the acceptance hash binds the header to chain and height, and the provider's rolling state is updated. *Attach.* Off chain, the provider uploads the batch to content-addressed storage to obtain the CID, derives a KEK via HKDF-SHA256 from the ABE-encrypted window seed and the acceptance hash, and wraps the per-window AEAD key, with AAD that binds the policy digest and acceptance hash. The chain records the batch CID and the two wrapped artifacts for the accepted window. In the base protocol, validators and the chain never handle plaintext metrics.

Usage. Producers run the streaming step for each record and maintain a local batch. At window end they commit the header, then finalize with the decider proof, and upon acceptance attach the batch. Any party can perform a non-decrypting availability check by rebuilding the Merkle root from public tuples and comparing it to the accepted header (Algorithm A.6). Authorized readers decrypt by recovering $\mathtt{Z} \rightarrow$ KEK $\rightarrow \mathtt{K}$, validating AAD, recomputing leaves, and aggregating to the root (Algorithm A.7). If a violating record is discovered, a challenger presents the record and Merkle path tied to the accepted header; on verification the provider is slashed (Algorithm A.10). The validator-side lifecycle and state transitions are normative and captured in Algorithm A.8.

6 Off-Chain Retrieval, Availability, and Decryption

Authorized parties must be able to validate an accepted window without learning plaintext metrics and, when authorized, selectively recover plaintexts in a way that preserves binding to the accepted header. The protocol provides two complementary paths. A non-decrypting availability check verifies that the off-chain batch is complete and matches the accepted Merkle root using only public tuples. An authorized retrieval path uses MA-ABE keys to recover a per-window seed, derives a KEK that is cryptographically tied to the accepted header, unwraps the per-window AEAD key, and then decrypts records with context-bound AAD so every recovered plaintext recomputes to its committed leaf and aggregates back to the published root. All byte layouts, encodings, and domain separators are canonical and fixed by Algorithm A.1; the procedures are captured in Algorithms A.6 and A.7.

Purpose. The availability path gives any observer a deterministic test for index completeness and root equality without access to secrets or plaintexts. The authorized path ensures that only readers with MA-ABE credentials learn the window seed and thus the KEK and AEAD key, while every decrypted record is context-checked through AAD and re-commitment. Together, these paths decouple storage and transport from trust: correctness is verified against the accepted header, and confidentiality is enforced by construction rather than by backend privilege.

Mechanism. After a window is marked *Accepted*, validators have computed the acceptance hash. Anyone can fetch the batch by content identifier and perform a non-decrypting check by parsing the sequence/digest pairs, requiring that there are exactly N_{win} records with strictly increasing sequences and no duplicates, assigning window-local indices by record order, aggregating with the canonical defaults to obtain a root, and requiring equality with the value recorded in the accepted header (Algorithm A.6). This path leaks no plaintexts and needs no keys.

An authorized reader parses the same header and record tuples, decrypts the ABE-wrapped seed with MA-ABE keys, derives the KEK tied to the acceptance hash, unwraps the per-window AEAD key, and then decrypts each record under that key with the canonical AAD from Algorithm A.1. The reader requires that the recovered sequence and timestamp exactly match the public values, recomputes the leaf commitment, and enforces equality to the published leaf digest. Finally, the reader aggregates all leaves at their indices to get a root and requires equality to the header. Any deviation—AAD mismatch, re-commitment mismatch, duplicate or missing indices, or root inequality—causes verification to fail and the batch to be rejected by the client (Algorithm A.7).

Usage. Operators can run the availability check to establish that an attached batch is complete and correctly bound to the accepted header without handling secrets. Authorized readers perform the full retrieval to recover plaintexts for analysis or archival; partial retrieval is safe because every record is independently bound to its context through AAD and its leaf commitment. Because the KEK is derived from the window seed and

the on-chain acceptance hash, wrapped keys and ciphertexts are useless off-context or prior to acceptance. If a decrypted record appears to violate policy, a challenger can present the record and Merkle path tied to the accepted header to a dispute-resolution mechanism, which may run off-chain or via a deployment-specific contract; Algorithm A.10 sketches one such flow. The core protocol does not require revealing plaintext metrics on-chain: in the default configuration, only commitments, ciphertexts, and succinct proofs appear on-chain or in IPFS, and all policy checks and challenges can be implemented without disclosing metric values.

7 Integration with Blockchain

The chain serves as a precise checkpoint for each reporting window. Rather than shipping raw telemetry, producers submit a compact header whose byte layout, domain separation, and step-relation binding are defined once in the canonical helpers (Algorithm A.1). A single succinct proof then certifies that the private computation behind that header is correct. Validators check these two artifacts exactly once per window, comparing public fields byte-for-byte against the previously committed header and running the verifier selected by the decider identifier. This keeps on-chain work small and predictable and, because the base protocol never requires posting plaintext metrics, preserves confidentiality by construction; deployments that choose to expose metrics for challenges do so as an explicit, optional extension. The commit/finalize handshake and its equality checks are specified in Algorithm A.4; the validator-side state machine and acceptance at a finalized height are normative in Algorithm A.8; the full producer→chain→storage path is collected in Algorithm A.5.

The window lifecycle follows a simple sequence. At the end of a window, a producer commits only the header; validators record this intent and perform ordering and sequencing checks, but they accept no storage or keys at this stage. When the succinct proof is ready, the producer finalizes by presenting it together with the public inputs; validators *compute the public-input vector from the committed header* and reject any mismatch against the provided inputs before running the verifier. On acceptance at a finalized height, the chain derives an attestation hash that binds the accepted header to that height. Only then may the producer attach a content address for the off-chain batch and the wrapped keys. This order prevents premature or out-of-context uploads from being referenced and ensures that any decryption material released later is cryptographically tied to the exact header the chain accepted.

Interoperability and evolution are handled explicitly so implementations do not diverge. A decider registry maps a stable identifier to a pinned verification key; producers choose the identifier at registration and cannot silently swap verifiers later (Algorithm A.8). The compiled step relation is guarded by a digest that appears both in the header and inside the decider as two field limbs; changing the compilation changes that digest, which causes old proofs to fail rather than be misinterpreted (Algorithm A.1). Policy parameters are serialized canonically to a digest and written to an append-only policy registry; when policies change, the new digest is appended and future windows refer to it, while historical windows remain auditable against the exact parameters they committed to (Algorithm A.8).

Storage is made predictable and resistant to abuse without exposing plaintexts. The canonical helpers extend the header with a declared records-payload size and a digest of the canonical payload (Algorithm A.1). On commit, validators reserve those bytes against the provider's plan and enforce per-window and per-epoch caps before any data is pinned; on attach, the contract requires the actual attachment to be within the reservation and, where configured, accepts a short validator-quorum receipt attesting the observed size (Algorithm A.8). Batches are kept according to the recorded retention plan and may be pruned after that horizon without affecting the on-chain attestation. Anyone can perform a non-decrypting availability check by rebuilding the Merkle root from public tuples and comparing it to the accepted header (Algorithm A.6), while authorized readers recover and re-commit plaintexts with context-bound associated data (Algorithm A.7).

Participation and accountability complete the process. A rotating subset of validators verifies each proof and can sign the attestation hash derived from the accepted header; these signatures can be aggregated for rewards or for downstream consumers that want a compact confirmation that verification actually took place. If a later check uncovers a violating record, the challenger presents the record and a Merkle path tied to the accepted header; the on-chain challenge procedure evaluates this evidence against the canonical helper and the recorded policy and, if upheld, applies the deployment's penalties (Algorithm A.10). With headers, proofs, storage, and keys all anchored to the same accepted bytes, the system keeps correctness easy to verify,

limits what must be trusted, and leaves plaintexts under the control of authorized readers.

Decider ceremony and governance (normative)

The Groth16 decider uses a structured reference string obtained via a multi-party ceremony (Powers-of-Tau phase 1 followed by circuit-specific phase 2). Ceremony transcripts *must* be published; participants *must* commit to publicly beaconed randomness and destroy any secret trapdoor material ("toxic waste"). The verification key pinned under the decider id in the on-chain `DeciderRegistry` (Algorithm A.8) *must not* change without an explicit governance action; rotation does not retroactively affect historical windows. Any change to the compiled step relation alters `H_step`, which the decider represents as two public field elements; old proofs then fail regardless of the pinned verification key; this guards against undetected circuit changes.

8 Security Analysis

Confidentiality holds because metrics remain encrypted under a per-window AEAD key that is MA-ABE–protected; AAD binds the provider, window, sequence, timestamp, schema id, policy digest, and decider id; and the in-circuit commitment binds the same context together with the plaintext metrics and per-record randomness. As a result, ciphertexts cannot be swapped across windows, schemas, or policies without detection. Integrity and correctness follow from the step circuit and Nova folding, while the Groth16 decider supplies succinct verification. The decider circuit is bound to the fixed step relation via `H_step` (two field limbs) and enforces equality to its compile-time constant; it also enforces the relaxed R1CS check for the finalized accumulator. The step circuit performs a membership-checked Merkle update keyed by the window-local record order index, preventing arbitrary root rewrites. Consistency is guaranteed because commitments appear both in the rolling root and in decryptable records. Freshness and ordering are enforced by monotone sequence indices and window identifiers in public inputs and by BFT finality. Storage gating ensures no on-chain batch attachment occurs unless the proof is accepted.

Omission resistance. The protocol proves that all *included* records satisfied policy; it does not by itself prove that all *generated* records were included. Deployments that require omission detection can bind agent-side signed counters or source-anchored append-only logs into the step circuit; Section 7 provides a standardized hook. In all cases, providers **must** reject duplicate indices $(\texttt{agent\_id}, i)$ within a window to preserve AEAD nonce uniqueness.

9 Conclusion

This design brings strong privacy and verifiable correctness into a domain that traditionally depends on trusted collectors. Agents keep their familiar workflow; providers introduce minimal per-update overhead; validators check one succinct proof per window; authorized readers decrypt only after acceptance; and auditors can recompute commitments without privileged access. The combination of MA-ABE, Nova IVC, and a Groth16 decider achieves strict acceptance at line rate and aligns incentives so that only verified data qualifies for attention or rewards. The result is a trust-minimized, auditable system for decentralized metrics reporting that is rigorous enough for engineers and clear enough for decision-makers who are accountable for performance, privacy, and cost.

References

[1] Melissa Chase. Multi-authority attribute based encryption. In *Theory of Cryptography Conference (TCC)*, 2007.

[2] Melissa Chase and Sherman S. M. Chow. Improving privacy and security in multi-authority attribute-based encryption. In *ACM CCS*, 2009.

[3] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[4] Srinath Setty. Nova: Recursive zero-knowledge arguments for incrementally verifiable computation. *IACR ePrint* 2021/370, 2021.

[5] Srinath Setty, *et al.* Nova: Recursive Proof Composition without a Trusted Setup. In *CRYPTO*, 2022.

[6] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Security*, 2021.

[7] iden3. Sparse Merkle Tree. Technical note, 2018. `https://iden3.io/research/papers/sparse-merkle-tree.pdf`

[8] Vitalik Buterin. Sparse Merkle trees. *ethresear.ch* post, 2018.

[9] M. Bellare and C. Namprempre. The Authenticated Encryption with Associated Data (AEAD) Interface. *RFC 5116*, 2008.

[10] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. *RFC 8439*, 2018.

[11] Jae Kwon. Tendermint: Consensus without mining. *Draft*, 2014.

[12] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. In *PODC*, 2019.

[13] Dan Boneh, Ben Lynn, Hovav Shacham. Short Signatures from the Weil Pairing. In *ASIACRYPT*, 2001.

[14] Dan Boneh, Manu Drijvers, Gregory Neven. BLS Multi-signatures With Public-Key Aggregation. In *CRYPTO*, 2018.

[15] Multiformats. CID: Content Identifier. `https://github.com/multiformats/cid`

[16] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn. BLAKE3: One function, fast everywhere. 2020. `https://github.com/BLAKE3-team/BLAKE3`

Appendices

Normative Algorithms (canonical, implementation-binding)

Implementation contract (normative, brief). Encodings are big-endian with fixed byte widths; field elements use canonical 32-byte encodings; byte-to-field mapping rejects out-of-range integers. Domain separators and Poseidon parameters are fixed as specified; Merkle index bit order, sibling orientation, and default nodes are canonical. SERHDR byte layout and the step-relation hash $\mathrm{H}_{\mathrm{step}}$ are binding; validators compute the *public-input vector* from SERHDR via Algorithm A.1 (two 16-byte limbs for each 32-byte value) and **byte-bind** it to the decider public inputs. AEAD uses 96-bit nonces; the *canonical nonce* is deterministically derived from the record order index per Algorithm A.1 (single-agent: index; multi-agent: agent id $\|$ index). The per-window AEAD key K **MUST** be unique per window; it **SHOULD** be sampled from a CSPRNG. (Deployments *may* derive $K \leftarrow \textsc{DeriveK}(Z, \mathsf{hdrHash})$ instead.) Only the per-window seed Z is ABE-encrypted; a KEK $\mathrm{HKDF}(Z, \mathsf{hdrHash})$ wraps K with AAD $\mathrm{enc}_F(() \,\mathrm{Hash}(\theta)) \,\|\, \mathsf{hdrHash}$. Chain enforces commit→finalize→attach, monotone $(\mathsf{wid}, \mathsf{seq})$ bounds, canonical decider id, and proof validity at a finalized acceptance height used in HdrHash. Within a window, providers **MUST** forbid duplicate $(\mathsf{agent\_id}, i)$ to preserve AEAD nonce uniqueness.

Implementation Notes and Parameters (normative).

Sym.	Meaning	Bound	Notes
k	metrics per vector	$\mathbb{N}_{>0}$	deployment 16–128
n	metric bit width	$n \le 32$	range checks for M_j
τ	timestamp width	$\tau \le 64$	64-bit timestamp
D	window tree depth	$D = \lceil \log_2 N_{\mathrm{win,max}} \rceil$	deployment-fixed
$N_{\mathrm{win,max}}$	max window length	$N_{\mathrm{win,max}} \le 2^D$	
bytes/window	AEAD plaintext cap	$\le 2^{30}$ bytes	AEAD safety envelope
$\mathbb{F}$	SNARK field	Fr(Pallas)	shared by Nova+Groth16
Poseidon	hash params	t=3, α=5, R_F=8, R_P=56	constants via Alg. A.2

Nonce uniqueness bounds (normative). Single-agent profile: **MUST** enforce $N_{\mathrm{win}} < 2^{96}$. Multi-agent profile: **MUST** enforce $N_{\mathrm{win}} \le 2^{64}$ and unique agent_id per agent per window; for any accepted window, at most one ciphertext for any $(\mathsf{agent\_id}, i)$.

Vector schemas. Bind schemaID into both the leaf and AAD. Schema migrations increment schemaID; changing only θ does not change the decider.

Profiles. A 32-bit *profile id* identifies the nonce/AAD/record layout: 1 = single-agent, 2 = multi-agent. It appears in SERHDR (Alg. A.1) and governs whether agent_id is present in leaves, AAD, and records.

Halo2/Plonkish variant. A pure Halo2 path can replace the step circuit with a Plonkish circuit that proves the entire window in a single shot or adopt a Halo2-folding framework to regain streaming amortization. The swap is mechanical for the system: data plane, headers, and verification interface remain unchanged.

Algorithm A.1 Canonical helpers (normative; single source of truth)

Define SNARK field and hashes: scalar field $\mathbb{F} = \mathrm{Fr}(\mathrm{Pallas})$; byte hash BLAKE3-256 outputs 32 bytes.
Define Poseidon over $\mathbb{F}$ with $(t{=}3, r{=}2, c{=}1, \alpha{=}5, R_F{=}8, R_P{=}56)$; constants from the pinned generator (Alg. A.2) and seed `POSEIDON_PALLAS_t3_alpha5_RF8_RP56:v1:maabe-streaming`.
Domain separators (ASCII):
$\mathsf{DS}_{\mathrm{leaf}}$:= `"maabe:leaf:v1"`; $\mathsf{DS}_{\mathrm{node}}$:= `"maabe:node:v1"`; $\mathsf{DS}_{\mathrm{step}}$:= `"maabe:step:v1"`;
$\mathsf{DS}_{\mathrm{attest}}$:= `"maabe:attest:v1"`; $\mathsf{DS}_{\mathrm{policy}}$:= `"maabe:policy:v1"`.
Define $\mathrm{I2OSP}(x, L)$: big-endian, fixed-length encoding to exactly L bytes; require $x < 256^L$.
Define $I2F_{\mathrm{be}}(b)$: map a fixed-width big-endian byte string to $\mathbb{F}$ if its integer $< |\mathbb{F}|$ (reject otherwise).
Define $I2F^{\mathrm{var}}_{\mathrm{be}}(\ell)$: map a variable-length ($|\ell| \le 31$) big-endian byte string ℓ to $\mathbb{F}$ as the corresponding integer with *no padding*, requiring $< |\mathbb{F}|$ (reject otherwise).
Define $\mathrm{enc}_4(x) := \mathrm{I2OSP}(x, 4)$; $\mathrm{enc}_8(x) := \mathrm{I2OSP}(x, 8)$.
Define $\mathrm{enc}_F(x)$: canonical 32-byte big-endian encoding of $x \in \mathbb{F}$ (reject non-canonical encodings).
Define $\mathrm{enc}^{\mathrm{bytes}}_L((\,)\,b)$: require $|b| = L$ bytes; **return** b (identity).
Define $\textsc{Split31}(b)$: split b into left-to-right limbs of ≤ 31 bytes (for any value that may be $\ge |\mathbb{F}|$).
Define $\textsc{Split16x2}(b)$ for $|b| = 32$: return $(b[0..15], b[16..31])$ (two 16-byte limbs).
Define $\textsc{PoseidonList}(\mathsf{DS}; X_1, \ldots, X_m)$:
absorb $I2F_{\mathrm{be}}(\mathrm{enc}_4(|\mathsf{DS}|))$ and the limbs of DS via $I2F^{\mathrm{var}}_{\mathrm{be}} \circ \textsc{Split31}$; then $I2F_{\mathrm{be}}(\mathrm{enc}_4(m))$; then each X_i (scalars or limb sequences mapped via $I2F^{\mathrm{var}}_{\mathrm{be}}$).
Define Merkle orientation: write $i = \sum_{\ell=0}^{D-1} b_\ell 2^\ell$ (little-endian bits, b_0 LSB). At level ℓ, if $b_\ell{=}0$ use left child, else right child.
Define node/leaf and defaults: $\textsc{NodeHash}(L, R) := \textsc{PoseidonList}(\mathsf{DS}_{\mathrm{node}}; L, R)$; $Z_0 := \textsc{PoseidonList}(\mathsf{DS}_{\mathrm{leaf}}; 0_{\mathbb{F}})$; $Z_{\ell+1} := \textsc{NodeHash}(Z_\ell, Z_\ell)$ for $\ell{=}0, \ldots, D{-}1$.
Define $\textsc{MerkleRecompute}(i, d, \mathsf{path}[0..D{-}1])$ (normative):
$x \leftarrow d$; **for** $\ell \leftarrow 0$ **to** $D{-}1$ **do**
if $b_\ell = 0$ **then** $x \leftarrow \textsc{NodeHash}(x, \mathsf{path}[\ell])$ **else** $x \leftarrow \textsc{NodeHash}(\mathsf{path}[\ell], x)$
return x
Define $\textsc{MerkleVerify}(\mathsf{root}, i, d, \mathsf{path}) := \big(\textsc{MerkleRecompute}(i, d, \mathsf{path}) = \mathsf{root}\big)$.
Define $\textsc{MerkleAggregate}(\{(i_i, d_i)\}_{i=0}^{N_{\mathrm{win}}-1})$ (normative, sparse input):
require all i_i distinct and $0 \le i_i < 2^D$
build map $\mathcal{M}[i] \leftarrow d$ for provided leaves; missing leaves are default Z_0
compute root by bottom-up folding using NodeHash and defaults Z_ℓ for absent subtrees; **return** the resulting root.
Encoding rule for Poseidon absorption in leaves (profile-aware):
$\mathsf{pid} \to \textsc{Split31}(\mathrm{enc}^{\mathrm{bytes}}_{32}((\,)\,\mathsf{pid}))$; $\mathsf{wid}, \mathsf{seq}, t \to I2F_{\mathrm{be}}(\mathrm{enc}_8(\cdot))$; $\mathsf{schemaID} \to I2F_{\mathrm{be}}(\mathrm{enc}_4(\cdot))$;
$\mathrm{Hash}(\theta)$ is a field element (absorbed directly; see below); each M_j is a scalar in $[0, 2^n)$; r is a 16-byte (128-bit) string (as-is via Split31 if needed);
in the multi-agent profile, also absorb $\mathsf{agent\_id} \to I2F_{\mathrm{be}}(\mathrm{enc}_4(\cdot))$.
Define $\mathrm{Hash}(\theta)$ (policy digest as field):
let $b :=$ canonical byte serialization of θ (deployment-pinned);
let $(\ell_1, \ldots, \ell_s) := \textsc{Split31}(b)$;
return $\textsc{PoseidonList}(\mathsf{DS}_{\mathrm{policy}};\ \ell_1, \ldots, \ell_s)$.
Define $\textsc{LeafCommit}(\mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \mathrm{Hash}(\theta), [\mathsf{agent\_id}], \mathbf{M}, r)$:
single-agent: := $\textsc{PoseidonList}(\mathsf{DS}_{\mathrm{leaf}};\ \mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \mathrm{Hash}(\theta), M_1, \ldots, M_k, r)$
multi-agent: := $\textsc{PoseidonList}(\mathsf{DS}_{\mathrm{leaf}};\ \mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \mathrm{Hash}(\theta), \mathsf{agent\_id}, M_1, \ldots, M_k, r)$
Define AAD bytes (profile-aware):
$\mathrm{AAD}(\cdot) := \mathrm{enc}^{\mathrm{bytes}}_{32}((\,)\,\mathsf{pid}) \parallel \mathrm{enc}_8(\mathsf{wid}) \parallel \mathrm{enc}_8(\mathsf{seq}) \parallel \mathrm{enc}_8(t) \parallel \mathrm{enc}_4(\mathsf{schemaID}) \parallel \mathrm{enc}_F(\mathrm{Hash}(\theta)) \parallel \mathrm{enc}_4(\mathsf{decider\_id})$ [$\parallel \mathrm{enc}_4(\mathsf{agent\_id})$],
where the final $\mathrm{enc}_4(\mathsf{agent\_id})$ term is present *iff* profile id = 2 (multi-agent).
Define canonical sparse R1CS encoding for step binding (normative):
Represent each matrix row as a list of nonzeros $(j, M_{i,j})$ with strictly increasing j.
$\mathrm{enc}^{\mathrm{r1cs}}_{\mathrm{v1}}(\mathbf{A}, \mathbf{B}, \mathbf{C}) := \mathrm{enc}_4(n_c) \parallel \mathrm{enc}_4(n_v) \parallel \textsc{SparseMat}(\mathbf{A}) \parallel \textsc{SparseMat}(\mathbf{B}) \parallel \textsc{SparseMat}(\mathbf{C})$,
where $\textsc{SparseMat}(\mathbf{M}) := \big\|_{i=0}^{n_c-1} \big(\mathrm{enc}_4(\mathrm{nnz}_i) \parallel \big\|_{m=1}^{\mathrm{nnz}_i} \big(\mathrm{enc}_4(j_{i,m}) \parallel \mathrm{enc}_F(M_{i,j_{i,m}})\big)\big)$.
Variable order and constraint order are part of compilation and MUST be fixed by the pinned toolchain; the above encoding assumes the toolchain's canonical ordering and does not materialize dense matrices.
Define step-relation hash (bytes): $\mathrm{H}_{\mathrm{step}} := \mathrm{BLAKE3\text{-}256}\big(\mathsf{DS}_{\mathrm{step}} \parallel \mathrm{enc}^{\mathrm{r1cs}}_{\mathrm{v1}}(\mathbf{A}, \mathbf{B}, \mathbf{C})\big)$.
Define public-input packing (normative; **13** field elements):
$(p_0, p_1) \leftarrow \textsc{Split16x2}(\mathsf{pid})$; $(h_0, h_1) \leftarrow \textsc{Split16x2}(\mathrm{H}_{\mathrm{step}})$
$\textsc{PubInPack}(\mathsf{SERHDR}_{\mathrm{base}})$ returns
$\Big(I2F^{\mathrm{var}}_{\mathrm{be}}(p_0), I2F^{\mathrm{var}}_{\mathrm{be}}(p_1), I2F_{\mathrm{be}}(\mathrm{enc}_8(\mathsf{wid})), I2F_{\mathrm{be}}(\mathrm{enc}_8(\mathsf{seq}_{\min})), I2F_{\mathrm{be}}(\mathrm{enc}_8(\mathsf{seq}_{\max})), R,$
$I2F_{\mathrm{be}}(\mathrm{enc}_8(t_{\min})), I2F_{\mathrm{be}}(\mathrm{enc}_8(t_{\max})), I2F_{\mathrm{be}}(\mathrm{enc}_8(N_{\mathrm{win}})), I2F_{\mathrm{be}}(\mathrm{enc}_4(\mathsf{schemaID})), \mathrm{Hash}(\theta), I2F^{\mathrm{var}}_{\mathrm{be}}(h_0), I2F^{\mathrm{var}}_{\mathrm{be}}(h_1)\Big)$.
Define aggregate commitment (optional public observability):
let $A := (\min_1, \max_1, \sum_1, \mathrm{sumsq}_1, \ldots, \min_k, \max_k, \sum_k, \mathrm{sumsq}_k)$
$A_{\mathrm{root}} := \textsc{PoseidonList}(\mathsf{DS}_{\mathrm{node}}; A)$
Note: A and A_{root} are *not* included in the canonical header or the decider's public inputs.
Define header bytes and attestation hash (*base* then *extended*):
$\mathsf{SERHDR}_{\mathrm{base}} := \mathrm{enc}^{\mathrm{bytes}}_{32}((\,)\,\mathsf{pid}) \parallel \mathrm{enc}_8(\mathsf{wid}) \parallel \mathrm{enc}_8(\mathsf{seq}_{\min}) \parallel \mathrm{enc}_8(\mathsf{seq}_{\max}) \parallel \mathrm{enc}_F(R) \parallel \mathrm{enc}_8(t_{\min}) \parallel \mathrm{enc}_8(t_{\max}) \parallel \mathrm{enc}_8(N_{\mathrm{win}}) \parallel \mathrm{enc}_4(\mathsf{schemaID}) \parallel \mathrm{enc}_F(\mathrm{Hash}(\theta)) \parallel \mathrm{enc}_4(\mathsf{decider\_id}) \parallel \mathrm{enc}_4(\mathsf{profile\_id}) \parallel \mathrm{enc}^{\mathrm{bytes}}_{32}((\,)\,\mathrm{H}_{\mathrm{step}})$,
given $B_{\mathrm{rec}} := \textsc{BatchPack}(\cdot)$, let $L := |B_{\mathrm{rec}}|$ and $H_{\mathrm{batch}} := \mathrm{BLAKE3\text{-}256}(B_{\mathrm{rec}})$; **set** $\mathsf{SERHDR} := \mathsf{SERHDR}_{\mathrm{base}} \parallel \mathrm{enc}_8(L) \parallel \mathrm{enc}^{\mathrm{bytes}}_{32}((\,)\,H_{\mathrm{batch}})$
Convention (normative): All references to SERHDR in this specification denote the *extended* bytes defined above, and $\textsc{HdrHash}(\mathsf{SERHDR}, \cdot, \cdot)$ *MUST* be computed over these extended bytes.
Define $\textsc{HdrHash}(\mathsf{SERHDR}, \mathsf{chainID}, h_{\mathrm{accept}}) := \mathrm{BLAKE3\text{-}256}(\mathsf{DS}_{\mathrm{attest}} \parallel \mathrm{enc}_4(\mathsf{chainID}) \parallel \mathrm{enc}_8(h_{\mathrm{accept}}) \parallel \mathsf{SERHDR})$.

Define $\textsc{BatchPack}(\{(d_i, C_i, \mathsf{seq}_i, t_i[, \mathsf{agent\_id}_i])\}_{i=1}^{N_{\mathrm{win}}})$ **(canonical records payload)**:
require strictly increasing $\mathsf{seq}_1 < \cdots < \mathsf{seq}_{N_{\mathrm{win}}}$ and $N_{\mathrm{win}} > 0$
$B \leftarrow \mathrm{enc}_8(N_{\mathrm{win}})$
for $i \leftarrow 1$ **to** N_{win} **do**
$B \leftarrow B \parallel \mathrm{enc}_F(d_i) \parallel \mathrm{enc}_8(|C_i|) \parallel \mathrm{enc}^{\mathrm{bytes}}_{|C_i|}((\,)\,C_i) \parallel \mathrm{enc}_8(\mathsf{seq}_i) \parallel \mathrm{enc}_8(t_i)$
if $\mathsf{profile\_id}{=}2$ **then** $B \leftarrow B \parallel \mathrm{enc}_4(\mathsf{agent\_id}_i)$
return B ▷ This is the *records payload* B_{rec}; the header is *not* included.
Define $\textsc{BatchDigest}(B) := \mathrm{BLAKE3\text{-}256}(B)$ ▷ digest over the *records payload*.
Define AEAD suite: deployments fix one suite per system or per schemaID; default is ChaCha20-Poly1305; key length equals suite's key size.
Define KEK derivation (canonical):
salt = BLAKE3-256(`"maabe-streaming:win-key"`); info = `"win-key"` $\parallel$ BLAKE3-256(hdrHash);

Algorithm A.2 Deterministic Poseidon parameter generator (normative)

Require: field $\mathbb{F} = \mathrm{Fr}(\text{Pallas})$, width t=3, full rounds R_F=8, partial rounds R_P=56, S-box exponent α=5
Require: seed string S = `POSEIDON_PALLAS_t3_alpha5_RF8_RP56:v1:maabe-streaming`
PRF/XOF: define $\mathsf{XOF}(\texttt{label}) := \mathrm{BLAKE3\_XOF}(S \parallel \texttt{label})$
Sample to field (rejection sampling): from XOF read 64 bytes at a time as a big-endian integer x; if $x < |\mathbb{F}|$ output x as an element of $\mathbb{F}$; else continue
Round constants: let $N_c = t \cdot (R_F + R_P)$; define $\mathsf{RC}[0..N_c-1]$ by drawing N_c field elements from $\mathsf{XOF}(\texttt{"poseidon:rc:v1"})$
MDS (Vandermonde over distinct nonzero points):
    draw distinct nonzero $s_0, s_1, s_2 \in \mathbb{F}$ from $\mathsf{XOF}(\texttt{"poseidon:mds:v1"})$
    set $\mathsf{MDS}_{i,j} := s_i^j$ for $i, j \in \{0, 1, 2\}$ ▷ this 3×3 Vandermonde is invertible iff s_i distinct
Return (MDS, RC)

MA-ABE authorities and revocation (normative). Only the per-window seed Z is ABE-encrypted. Deployments *MUST* sample fresh, unpredictable Z per window and define a fixed attribute schema including an *epoch* (e.g., window or time-bucket identifier) and the set of issuing authorities. Readers obtain epoch-scoped decryption keys; revocation is realized by withholding future-epoch keys. Collusion resistance follows the underlying MA-ABE construction; attributes and authority identifiers are part of the ABE ciphertext's label set.

Algorithm A.3 StreamingUpdate (single source of truth; definitions inline)

Require: public constants to step-circuit: $(\mathsf{pid}, \mathsf{wid}, \mathsf{schemaID}, \mathrm{Hash}(\theta), \mathrm{H_{step}})$
Require: private witness: canonical policy serialization θ_{bytes}
Require: profile id $\in \{1, 2\}$ (single-agent or multi-agent)
Require: AEAD suite fixed; **unique** per-window K; 96-bit nonce rule (canonical, deterministic):
    single-agent: $\mathrm{I2OSP}(i, 12)$; *multi-agent:* $\mathrm{I2OSP}(\mathsf{agent\_id}, 4) \parallel \mathrm{I2OSP}(i, 8)$ with unique agent_id and $N_{\mathrm{win}} \le 2^{64}$
Initialization (per window): $S_0 = (Z_D, \mathbf{u}_{\max}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathsf{seq}_{\min}, T_{\max}, 0, 0)$ where $\mathbf{u}_{\max}=(2^n-1, \ldots, 2^n-1)$ and $T_{\max}=2^\tau-1$
Inputs per record: $\mathbf{M} \in [0, 2^n)^k$, $r \in \{0,1\}^{128}$, $t \in [0, 2^\tau)$; $\mathsf{seq} \in \mathbb{N}$; optional $\mathsf{agent\_id} \in [0, 2^{32})$ when multi-agent
        Merkle witness $(d_{\mathrm{old}}, \mathsf{path})$ for the *record order index* i (where the first record has $i = 0$ and each step increments i by one)
$d \leftarrow \textsc{LeafCommit}(\mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \mathrm{Hash}(\theta), [\mathsf{agent\_id}], \mathbf{M}, r)$
StepCheck (in-circuit):
    enforce $0 \le M_j < 2^n$ and $0 \le t < 2^\tau$; if multi-agent, enforce $0 \le \mathsf{agent\_id} < 2^{32}$
    compute $\mathrm{Hash}(\theta_{\mathrm{bytes}}) \leftarrow \textsc{PoseidonList}(\mathsf{DS}_{\mathrm{policy}};\ \textsc{Split31}(\theta_{\mathrm{bytes}}))$
    enforce $\mathrm{Hash}(\theta_{\mathrm{bytes}}) = \mathrm{Hash}(\theta)$
    enforce $\mathsf{seq} > \mathsf{seq}_{\mathrm{prev}}$ (strictly increasing, allowing gaps); enforce $i = i_{\mathrm{prev}} + 1$ (record order index)
    verify $\textsc{MerkleVerify}(\mathsf{root}, i, d_{\mathrm{old}}, \mathsf{path})$
$\mathsf{root}' \leftarrow \textsc{MerkleRecompute}(i, d, \mathsf{path})$ ▷ parents via NodeHash; leaves are exactly d
require $\textsc{PolicyPredicate}(\mathbf{M}, \theta_{\mathrm{bytes}}) = \mathsf{true}$; update timestamp bounds and step counters ▷ aggregates, if maintained, are optional and not part of the canonical header
AEAD encryption (normative, out-of-circuit):
    $$\text{nonce} \leftarrow \begin{cases} \mathrm{I2OSP}(i, 12) & \text{if profile} = \text{single-agent} \\ \mathrm{I2OSP}(\mathsf{agent\_id}, 4) \parallel \mathrm{I2OSP}(i, 8) & \text{if profile} = \text{multi-agent} \end{cases}$$
    $C \leftarrow \mathsf{AEAD.Enc}_K((\mathbf{M}, r, \mathsf{seq}, t);\ \mathrm{AAD}(\cdot),\ \text{nonce})$
append $(d, C, \mathsf{seq}, t[, \mathsf{agent\_id}])$ to the off-chain batch buffer; fold $(S \to S')$ into Nova accumulator

Algorithm A.4 Commit and Finalize (normative header bytes and bindings)

Commit: publish header bytes SERHDR (extended; as in Alg. A.1) for (pid, wid)
Require: $N_{\mathrm{win}} > 0$; no batch reference prior to acceptance
Finalize: let $\mathsf{SERHDR}_{\mathrm{base}}$ be parsed from SERHDR
Public-input binding (normative): compute $\mathsf{pubVec}^{\mathrm{expected}} \leftarrow \textsc{PubInPack}(\mathsf{SERHDR}_{\mathrm{base}})$ (Alg. A.1)
verify Groth16 decider proof under the canonical $\mathsf{vk}_{\mathrm{decider}}[\mathsf{decider\_id}]$ with provided public inputs pubVec
Require: $\mathsf{pubVec} = \mathsf{pubVec}^{\mathrm{expected}}$ ▷ exact element-wise equality
on acceptance at **finalized** height h_{accept} on chain chainID, compute $\mathsf{hdrHash} \leftarrow \textsc{HdrHash}(\mathsf{SERHDR}, \mathsf{chainID}, h_{\mathrm{accept}})$
off-chain (provider): later, when attaching, compute $\mathrm{KEK} \leftarrow \textsc{DeriveKEK}(Z, \mathsf{hdrHash})$

Algorithm A.5 End-to-End Metric Lifecycle (normative; provider → chain → IPFS → authorized reader)

Require: canonical helpers from Alg. A.1; streaming step from Alg. A.3; commit/finalize from Alg. A.4; attachment from Alg. A.9
Require: provider controls Server A (producer) and holds per-window K; authorized reader (Client A) holds MA-ABE keys; chain exposes a canonical decider registry

1: **Per-record processing at the provider (Server A):**
ingest metric vector $\mathbf{M}$ with timestamp t and sequence seq; sample $r \leftarrow \{0,1\}^{128}$
optional $\mathsf{agent\_id}$ if multi-agent profile
$d \leftarrow \textsc{LeafCommit}(\mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \mathrm{Hash}(\theta), [\mathsf{agent\_id}], \mathbf{M}, r)$
run Alg. A.3 step: enforce $\textsc{PolicyPredicate}(\mathbf{M}, \theta_{\text{bytes}})$, verify/update Merkle root at record order index $i \in \{0, \ldots, N_{\text{win}} - 1\}$, update bounds, fold into Nova accumulator

2: **Per-record encryption at the provider (out of circuit):**
nonce ← canonical rule from Alg. A.3
$C \leftarrow \mathsf{AEAD.Enc}_K((\mathbf{M}, r, \mathsf{seq}, t);\ \mathrm{AAD}(\cdot),\ \text{nonce})$
append $(d, C, \mathsf{seq}, t[, \mathsf{agent\_id}])$ to the local off-chain records buffer

3: **Records payload packaging (pre-commit; canonical and deterministic):**
$B_{\text{rec}} \leftarrow \textsc{BatchPack}\Big(\{(d_i, C_i, \mathsf{seq}_i, t_i[, \mathsf{agent\_id}_i])\}_{i=1}^{N_{\text{win}}}\Big)$
$L \leftarrow |B_{\text{rec}}|;\quad H_{\text{batch}} \leftarrow \textsc{BatchDigest}(B_{\text{rec}})$

4: **End-of-window commit (intent only; no storage or keys):**
compute header bytes SERHDR per Alg. A.1 with final R, bounds $(\mathsf{seq}_{\min}, \mathsf{seq}_{\max}, t_{\min}, t_{\max}, N_{\text{win}})$, and bindings $(\mathsf{schemaID}, \mathrm{Hash}(\theta), \mathrm{H}_{\text{step}}, \mathsf{decider\_id}, \mathsf{profile\_id})$
append $\mathrm{enc}_8(L) \,\|\, \mathrm{enc}_{32}^{\text{bytes}}(()\, H_{\text{batch}})$ to SERHDR as specified in Alg. A.1
broadcast `CommitWindow`$(\mathsf{pid}, \mathsf{wid}, \mathsf{SERHDR})$

5: **Finalization with one succinct proof (validators learn no plaintext):**
produce Groth16 decider proof π_{win} for the Nova-folded instance; compute $\mathsf{pubVec}^{\text{expected}} \leftarrow \textsc{PubInPack}(\mathsf{SERHDR}_{\text{base}})$
call `FinalizeWindow`; validators require $\mathsf{pubVec} = \mathsf{pubVec}^{\text{expected}}$ and verify π_{win} under the canonical $\mathsf{vk}_{\text{decider}}[\mathsf{decider\_id}]$
on acceptance at **finalized** height h_{accept} on chain $\mathsf{chainID}$, compute $\mathsf{hdrHash} \leftarrow \textsc{HdrHash}(\mathsf{SERHDR}, \mathsf{chainID}, h_{\text{accept}})$

6: **Post-acceptance attachment (bind key material to the accepted header):**
set $\mathsf{batch\_bytes} \leftarrow B_{\text{rec}}$ ▷ the IPFS object is exactly the canonical records payload
$\mathsf{CID} \leftarrow \mathsf{IPFS.Put}(\mathsf{batch\_bytes})$
$\mathrm{KEK} \leftarrow \textsc{DeriveKEK}(Z, \mathsf{hdrHash});\quad \mathsf{CT}_K \leftarrow \mathsf{AEAD.Enc}_{\mathrm{KEK}}(K;\ \text{aad} = \mathrm{enc}_F(\mathrm{Hash}(\theta)) \,\|\, \mathsf{hdrHash},\ \text{nonce} = \text{nonce}_K)$
broadcast `AttachBatch`$(\mathsf{pid}, \mathsf{wid}, \mathsf{CID}, \mathsf{CT}_{\mathrm{ABE}}(Z), \mathsf{CT}_K\,[, \sigma_{\text{size}}])$ ▷ σ_{size} is optional size receipt; see Alg. A.8

7: **Authorized retrieval and end-to-end verification (Client A):**
download batch by CID; parse header from chain or embedded manifest; obtain $(\mathsf{pid}, \mathsf{wid}, \mathsf{schemaID}, \mathrm{Hash}(\theta), \mathsf{decider\_id}, \mathsf{profile\_id}, \mathsf{seq}_{\min}, \mathsf{seq}_{\max}, N_{\text{win}}, L, H_{\text{batch}})$
verify $\textsc{BatchDigest}(\mathsf{batch\_bytes}) = H_{\text{batch}}$ and $|\mathsf{batch\_bytes}| = L$; then proceed as in Alg. A.7

Algorithm A.6 AvailabilityCheck (non-decrypting; index completeness and root)

Require: batch provides tuples $(\mathsf{seq}_i, t_i, d_i, C_i[, \mathsf{agent\_id}_i])$ for $i = 1..N_{\text{win}}$
Require: strictly increasing seq_i with $\mathsf{seq}_1 = \mathsf{seq}_{\min}$ and $\mathsf{seq}_{N_{\text{win}}} = \mathsf{seq}_{\max}$
1: $\forall i:\ i_i \leftarrow i - 1$ ▷ record order index is the tuple order
2: $R' \leftarrow \textsc{MerkleAggregate}(\{(i_i, d_i)\})$ ▷ defaults Z_ℓ from Alg. A.1
Require: $R' = R$ ▷ matches accepted header
3: **return** `OK`

Algorithm A.7 Off-Chain Retrieval and Decryption (Authorized Users)

1: **function** $\textsc{OffChainRetrievalAndDecryption}(\mathsf{CID}, \{\mathsf{SK}_{\mathrm{ABE},i}\}, \mathsf{CT}_{\mathrm{ABE}}(Z), \mathsf{CT}_K, R, \mathsf{hdrHash})$
2: **download** batch; parse header $(\mathsf{pid}, \mathsf{wid}, \mathsf{schemaID}, \mathrm{Hash}(\theta), \mathsf{decider\_id}, \mathsf{profile\_id}, \mathsf{seq}_{\min}, \mathsf{seq}_{\max}, N_{\text{win}})$
3: **parse** records: tuples $(d_i, C_i, \mathsf{seq}_i, t_i[, \mathsf{agent\_id}_i])$ for $i = 1..N_{\text{win}}$
4: $Z \leftarrow \mathsf{ABE.Decrypt}(\mathsf{CT}_{\mathrm{ABE}}(Z);\ \{\mathsf{SK}_{\mathrm{ABE},i}\})$
5: $\mathrm{KEK} \leftarrow \textsc{DeriveKEK}(Z, \mathsf{hdrHash})$
6: $K \leftarrow \mathrm{AEAD.Dec}_{\mathrm{KEK}}(\mathsf{CT}_K;\ \text{aad} = \mathrm{enc}_F(\mathrm{Hash}(\theta)) \,\|\, \mathsf{hdrHash},\ \text{nonce} = \text{nonce}_K)$
7: $\forall i:\ i_i \leftarrow i - 1$
8: $\forall i:\ \text{nonce}_i \leftarrow \begin{cases} \mathrm{I2OSP}(i_i, 12) & \text{if profile} = \mathsf{single\text{-}agent} \\ \mathrm{I2OSP}(\mathsf{agent\_id}_i, 4) \,\|\, \mathrm{I2OSP}(i_i, 8) & \text{if profile} = \mathsf{multi\text{-}agent} \end{cases}$
9: $\forall i:\ (\mathbf{M}_i, r_i, \mathsf{seq}'_i, t'_i) \leftarrow \mathsf{AEAD.Dec}_K(C_i;\ \mathrm{AAD}(\cdot),\ \text{nonce}_i)$
Require: $\mathsf{seq}'_i = \mathsf{seq}_i,\ t'_i = t_i$
10: $\forall i:\ d'_i \leftarrow \textsc{LeafCommit}(\mathsf{pid}, \mathsf{wid}, \mathsf{seq}_i, t_i, \mathsf{schemaID}, \mathrm{Hash}(\theta), [\mathsf{agent\_id}_i], \mathbf{M}_i, r_i)$
Require: $d'_i = d_i$
11: $R' \leftarrow \textsc{MerkleAggregate}\big(\{(i_i,\ d'_i)\}\big)$;
Require: $R' = R$
12: **return** `OK`
13: **end function**

Algorithm A.8 On-chain state and lifecycle (normative; with size-safe attachment)

State: DeciderRegistry: map decider_id $\mapsto$ $\mathsf{vk}_{\text{decider}}$
State: ProviderState[pid] record with fields:
    decider_id, policyParams, policyHash, bond, lastWindowID, lastSeq, lastRoot
State: PendingCommit[(pid, wid)] $\in \{\perp$ or SERHDR$\}$ with timestamp/height
State: Accepted[(pid, wid)] $\in \{\perp$ or (SERHDR, hdrHash, h_{accept})$\}$ with timestamp/height
State: PolicyRegistry: map Hash(θ) $\mapsto \theta$ (canonical serialization), append-only

State (added): StoragePlan[pid] record with fields:
    window_bytes_cap, epoch_bytes_cap, epoch_id, epoch_bytes_used
State (added): Reservation[(pid, wid)] $\in \{\perp$ or declared_size$\}$

function RegisterProvider(pid, decider_id, policyParams, bond)
Require: ProviderState[pid] is empty
Require: decider_id $\in$ DeciderRegistry
    $\theta \leftarrow$ canonical serialization derived from policyParams ▷ deployment-specific mapping
    policyHash $\leftarrow$ Hash(θ)
    ProviderState[pid] $\leftarrow$ (decider_id, policyParams, policyHash, bond, 0, 0, 0)
    PolicyRegistry[policyHash] $\leftarrow \theta$
end function
function UpdatePolicy(pid, policyParams)
Require: authorized caller for pid
    $\theta \leftarrow$ canonical serialization from new policyParams
    ProviderState[pid].policyParams $\leftarrow$ policyParams
    ProviderState[pid].policyHash $\leftarrow$ Hash(θ)
    PolicyRegistry[Hash(θ)] $\leftarrow \theta$ ▷ append-only, versioned by digest
end function

function CommitWindow(pid, wid, SERHDR)
    parse SERHDR exactly as in Alg. A.1 into (pid$'$, wid$'$, $\mathsf{seq}_{\min}$, $\mathsf{seq}_{\max}$, R, $t_{\min}$, $t_{\max}$, N_{win}, schemaID, Hash(θ), decider_id, profile_id, H_{step}, declared_size, H_{batch}
Require: pid$'$=pid and wid$'$=wid
Require: decider_id = ProviderState[pid].decider_id
Require: $N_{\text{win}} > 0$
Require: $\mathsf{seq}_{\min}$ > ProviderState[pid].lastSeq ▷ monotone, gaps allowed
Require: wid = ProviderState[pid].lastWindowID + 1
Require: PendingCommit[(pid, wid)]$= \perp$ ▷ no duplicate commits
Require: SequenceRolloverGuard(pid, N_{win})

    // Size reservation and plan enforcement
Require: `declared_size` $\le$ StoragePlan[pid].`window_bytes_cap`
    if current epoch $\neq$ StoragePlan[pid].`epoch_id` **then**
        StoragePlan[pid].`epoch_bytes_used` $\leftarrow 0$
        update StoragePlan[pid].`epoch_id`
Require: StoragePlan[pid].`epoch_bytes_used` + `declared_size` $\le$ StoragePlan[pid].`epoch_bytes_cap`
    StoragePlan[pid].`epoch_bytes_used` $\leftarrow$ StoragePlan[pid].`epoch_bytes_used` + `declared_size`
    Reservation[(pid,wid)] $\leftarrow$ `declared_size` ▷ reserve bytes for this window

    PendingCommit[(pid, wid)] $\leftarrow$ SERHDR with current height/time
end function
function FinalizeWindow(pid, wid, π_{win}, pubVec)
Require: PendingCommit[(pid, wid)] exists as SERHDR
    parse SERHDR into $\mathsf{SERHDR}_{\text{base}}$; compute $\mathsf{pubVec}^{\text{expected}} \leftarrow$ PubInPack($\mathsf{SERHDR}_{\text{base}}$)
    decider_id $\leftarrow$ value from SERHDR; vk $\leftarrow$ DeciderRegistry[decider_id]
Require: Hash(θ) = ProviderState[pid].policyHash
Require: pubVec = $\mathsf{pubVec}^{\text{expected}}$ **and** Groth16.Verify(vk, pubVec, π_{win}) = `true`
    $h_{\text{accept}} \leftarrow$ current height; chainID $\leftarrow$ network id
    hdrHash $\leftarrow$ HdrHash(SERHDR, chainID, h_{accept}) ▷ Alg. A.1
    Accepted[(pid, wid)] $\leftarrow$ (SERHDR, hdrHash, h_{accept})
    ProviderState[pid].lastSeq $\leftarrow \mathsf{seq}_{\max}$; ProviderState[pid].lastWindowID $\leftarrow$ wid; ProviderState[pid].lastRoot $\leftarrow R$
    PendingCommit[(pid, wid)] $\leftarrow \perp$
end function
function AttachBatch(pid, wid, CID, $\mathsf{CT}_{\text{ABE}}(Z)$, CT_K, $\sigma^{\text{opt}}_{\text{size}}$)
Require: Accepted(pid, wid) exists
Require: Reservation(pid, wid) exists as declared_size
    if $\sigma^{\text{opt}}_{\text{size}} \neq \perp$ **then** ▷ optional attested metering
        require VerifySizeReceipt$\sigma^{\text{opt}}_{\text{size}}$; pid, wid, CID, declared_size, H_{batch}=`true`
    store (CID, $\mathsf{CT}_{\text{ABE}}(Z)$, CT_K) on-chain for (pid, wid)
    Reservation(pid, wid) $\leftarrow \perp$ ▷ release reservation
end function
function PruneExpiredCommit(pid, wid)
Require: PendingCommit[(pid, wid)] exists and finalize deadline Δ_{final} exceeded
    PendingCommit[(pid, wid)] $\leftarrow \perp$
    if Reservation(pid, wid) $\neq \perp$ **then** Reservation(pid, wid) $\leftarrow \perp$ ▷ release reserved bytes
end function
function SequenceRolloverGuard(pid, N_{win})
    $L \leftarrow$ ProviderState[pid].lastSeq
Require: $L \le 2^{64} - N_{\text{win}}$ ▷ reject further commits if about to wrap
end function

function VerifySizeReceipt(σ_{size}; pid, wid, CID, declared_size, H_{batch})
    Input: a threshold/aggregate signature from a pinning quorum over message
    $m :=$ (`"maabe:size:v1"`, pid, wid, CID, declared_size, H_{batch})
    return BLS.VerifyAggregate(pk_set, m, σ_{size}) ▷ deployment-pinned quorum pk_set
end function

Algorithm A.9 Attachment (wrapper; reference target for Alg. A.5)

Require: accepted $(\mathsf{pid}, \mathsf{wid})$ with SERHDR (extended) and hdrHash on-chain
Require: canonical records payload B_{rec}, its CID, and wrapped artifacts $\mathsf{CT}_{\text{ABE}}(Z), \mathsf{CT}_K$
1: **Provider:** ensure B_{rec} matches L, H_{batch} in SERHDR
2: **Call** `AttachBatch`$(\mathsf{pid}, \mathsf{wid}, \mathsf{CID}, \mathsf{CT}_{\text{ABE}}(Z), \mathsf{CT}_K[, \sigma_{\text{size}}])$ (Algorithm A.8)

Algorithm A.10 Challenge (optional, deployment-specific; evidence of policy violation)

1: **Context:** This procedure is intended to run off-chain; only a verdict or a succinct proof derived from it need be posted on-chain.
Require: accepted $(\mathsf{pid}, \mathsf{wid})$ with SERHDR (extended), R, $\text{Hash}(\theta)$, and profile_id
Require: challenger-supplied record $(\mathbf{M}, r, \mathsf{seq}, t[, \mathsf{agent\_id}])$ and Merkle path path for the record order index i (where i is the record's ordinal position within the committed payload)
2: parse schemaID and $\mathsf{seq}_{\min}$ from SERHDR (Alg. A.1)
3: $d \leftarrow \textsc{LeafCommit}(\mathsf{pid}, \mathsf{wid}, \mathsf{seq}, t, \mathsf{schemaID}, \text{Hash}(\theta), [\mathsf{agent\_id}], \mathbf{M}, r)$ (Alg. A.1)
4: **Verify** $\textsc{MerkleVerify}(R, i, d, \mathsf{path}) = \texttt{true}$ under canonical orientation/defaults
5: **Re-evaluate** $\textsc{PolicyPredicate}(\mathbf{M}, \theta_{\text{bytes}})$ off-chain using canonical θ from PolicyRegistry; or, where available, verify a dedicated on-chain predicate proof bound to SERHDR
6: **If** predicate fails **then** apply deployment-defined penalties (slash provider's bond for pid); otherwise reject challenge

Interoperability and minimal test vectors (normative)

Endianness. All fixed-width integer encodings are big-endian via `I2OSP`. `enc_F(x)` is the canonical 32-byte big-endian encoding of an element of the SNARK field $\mathbb{F}$. Implementations using libraries that expose field elements in little-endian *must* convert to the specified big-endian form before hashing or serialization.

Public-input packing sanity vectors.

- For any 32-byte value b, $\textsc{Split16x2}(b)$ returns $(b[0..15], b[16..31])$; then $I2F_{\text{be}}^{\text{var}}(b[0..15])$ and $I2F_{\text{be}}^{\text{var}}(b[16..31])$ are valid field elements because $2^{128} < |\mathbb{F}|$.
- If $\mathsf{pid} = \texttt{0x11} \times 32$, then
$$(\mathsf{pid}^{(0)}, \mathsf{pid}^{(1)}) = \big(I2F_{\text{be}}^{\text{var}}(\texttt{0x11} \times 16),\ I2F_{\text{be}}^{\text{var}}(\texttt{0x11} \times 16)\big).$$
- If $\text{H}_{\text{step}} = \texttt{00} \times 3101$, then $(\text{H}_{\text{step}}^{(0)}, \text{H}_{\text{step}}^{(1)}) = \big(I2F_{\text{be}}^{\text{var}}(\texttt{00} \times 16),\ I2F_{\text{be}}^{\text{var}}(\texttt{00} \times 1501)\big)$.

Encoding sanity vectors.

- `enc_8(0x0000000000001234)` = `00 00 00 00 00 00 12 34`.
- `enc_4(0x0000002A)` = `00 00 00 2A`.
- `enc_F(1)` = `00` $\times$ 31 `||` `01`.
- If `pid` is 32 bytes of `0x11`, `wid`= 1, `seq`= 9, `t`= 0, `schemaID`= 2, `Hash(theta)` encodes to `00`$\times$`3101`, and `decider_id`= 7, then the **single-agent** AAD bytes are (concatenation in this order):
  - `pid` (32 bytes) ||
  - `enc_8(wid)` = `00 00 00 00 00 00 00 01` ||
  - `enc_8(seq)` = `00 00 00 00 00 00 00 09` ||
  - `enc_8(t)` = `00 00 00 00 00 00 00 00` ||
  - `enc_4(schemaID)` = `00 00 00 02` ||
  - `enc_F(Hash(theta))` = `00`$\times$`3101` ||
  - `enc_4(decider_id)` = `00 00 00 07`.
- With the same parameters and `agent_id`= $0x0000002A$, the **multi-agent** AAD appends `enc_4(agent_id)` = `00 00 00 2A`.
- Nonce examples (profile-aware):
  - **single-agent:** record order index i=9 $\Rightarrow$ nonce = `00`$\times$`11 09`.
  - **multi-agent:** `agent_id`=`0x0000002A`, i=9 $\Rightarrow$ nonce = `00 00 00 2A || 00 00 00 00 00 00 00 09`.

Copyright and Licensing Statement