PICKLE: Patchable Merkle Multiproofs for Verifiable Utility Intent

Seonghyun Kim
Ericsson Research
kim.seonghyun@ericsson.com

***Abstract*—Intent-based networking frameworks such as 3GPP TS 28.312 introduce utility-driven fulfilment, where producers map high-level intents to quantitative targets via utility formulas over KPIs, but the relationship between the KPIs declared in the intent expectation and the KPIs used in the utility is unconstrained. We address this Utility–Expectation gap with PICKLE (Patchable InCremental multiproof merKLE tree), a generic hash-only provenance layer for such settings. PICKLE commits an application's state vector in an incremental Merkle tree and equips each verifier with a batch proof expressed purely in terms of node positions. A single global sibling map stores each required hash at most once, while per-verifier proofs reference this map without duplicating hashes. Leaf updates patch the global map along the affected paths, leaving proof structure unchanged. As a result, patch communication scales with the number of distinct touched siblings rather than with the number or size of verifier batches while preserving per-verifier isolation. We implement PICKLE and compare it to per-path proofs and per-verifier multiproofs on synthetic multi-verifier workloads. Across varied verifier numbers and tree sizes, PICKLE reduces patch communication cost and update time, while relying only on hash computations and simple table lookups.**

I. Introduction

Intent-Based Networking (IBN) has become one of the emerging autonomous network architectures, promising operators a shift from configuration-centric workflows to declarative and high-level optimizations. In the 3GPP standard, this operational model is formalized through Intent Driven Management Service (MnS) in TS 28.312 [1], where consumers express performance expectations while producers select concrete actions to satisfy them. In this paper, we use the term Intent Management Function (IMF) to treat both consumers and producers uniformly, reflecting that those modules combine ingestion, interpretation, and verification in a single logical entity. The specification introduces utility-based fulfilment, allowing producers to compute a numerical score of success using the `IntentUtilityFormula` and `UtilityResult` information elements. These mechanisms are intended to capture the degree to which a network's behaviour aligns with consumer's expectations, thereby supporting negotiation, adaptation, and autonomous decision-making.

Workshop on Security and Privacy of Next-Generation Networks (FutureG) 2026
23 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-9-7
https://dx.doi.org/10.14722/futureg.2026.23033
www.ndss-symposium.org

TABLE I: Observability of utility computation and intent fulfilment under reporting from intent consumer's perspective.

	Behavior A	**Behavior B**	**Observable**
Throughput (T)	0.90	0.90	✓
Energy (E)	0.50	0.01	✗
Latency (L)	0.50	0.99	✗
IntentUtilityFormula	$T + (1 - E) + (1 - L)$		✓
UtilityResult	1.9	1.9	✓

The standard leaves freedom that the utility intent does *not* need to be restricted to the KPIs explicitly declared in the consumer's expectations. As a result, producers may incorporate unobservable KPIs without violating the specification. These behaviours remain syntactically valid while being semantically divergent, enabling a form of interpretation drift where the producer's understanding of the intent silently diverges from the consumer's. Table I illustrates how this may lead to a potential structural ambiguity, when the expectation includes Throughput KPI only: the same set of observable values may correspond to different behaviors.

Once fulfilment is driven by a numerical score whose parameters are only partially observable and constrained by the consumer, the gap between the declared expectations and the optimized metrics can be exploited as an attack surface. This concern, which can be classified as a security-relevant Adversarial Goodhart [2], aligns with prior findings in alignment and security research, where drift under repeated optimization is shown to accumulate into divergent system states [3], and in adversarial optimization settings where subtle metric misspecification leads to exploitative behaviour that evades monitoring [4], [5]. Prior research on reward tampering shows that when an optimizer both acts through an objective function and controls aspects of its definition, it tends to exploit gaps between the stated goal and the measurable target [6], [7]. Moreover, RFC 9315 explicitly cautions that intent-based automation can be "error amplification due to the combination of a higher degree of automation and the intrinsic higher degree of freedom, ambiguity, and implicit information conveyed by intent statements" [8].

Merkle-tree authenticated data structures can be a fit for bridging the gap in a multi-vendor system because they bind evolving semantics to a compact and integrity-protected digest while enabling selective verification via inclusion proofs. This pattern is established in transparency systems, such as

logging and auditing [9], [10]. In networking, prior works use Merkle commitments to make aggregated measurements tamper-evident [11], [12]. However, these lines of work typically optimize for one-shot verification rather than verification of recurring reports in *closed-loop automation*. In contrast, an IMF participates in a closed-loop control, which repeatedly evaluates intent semantics against periodic fulfilment/utility reports. Verifiers in this setting behave as *long-lived subscribers* that verify a stream of reports over extended horizons.

In this paper, we make the following contributions:

- **Utility–Expectation gap formalization:** We formalize an ambiguity in 3GPP TS 28.312 that the KPIs used in a producer's utility function are not bound to the KPIs declared in the consumer's expectations, enabling potential adversarial utility intent injection while reported fulfilment remains within a normal range.
- **Patch-only and hash-only provenance:** We design a novel hash-only Merkle multiproof scheme achieving patch cost proportional only to the affected siblings, while matching the minimum number of distinct siblings for the combined verifier batches.

II. Related Work

A. Incremental Merkle Trees

Merkle trees [13] are cryptographic and hash-based data structures that commit to a collection of values while supporting logarithmic-size inclusion proofs. Classical constructions are static: modifying a leaf requires recomputing all internal nodes along the path to the root and regenerating proofs for all verifiers. To reduce this cost, several incremental or dynamic variants have been explored. Crosby and Wallach [10] introduced incremental hashing for logarithmic-time updates, which later influenced the incremental append-only transparency logs in RFC 6962 [9]. Cassez and others have formally verified incremental Merkle algorithms in Dafny, confirming their correctness and asymptotic complexity [14]. These works primarily optimize the *tree*, while treating *proofs* as on-demand objects that are recomputed for each query from scratch. They do not analyse the setting where many long-lived proofs must be incrementally patched after each state update and the cost of maintaining such proofs across many updates.

B. Merkle Multiproofs

When a verifier needs to prove membership of many leaves in the same tree, *Merkle multiproofs* allow the prover to deduplicate sibling nodes shared across paths and send a single compact proof. Ramabaja and Avdullahu's *Compact Merkle multiproofs* [15] further optimize sparse multiproofs using position-based encodings that reconstruct internal-node locations from leaf indices, reducing proof size. These designs share our high-level goal of avoiding full proof recomputation, but they are optimized for consensus settings, where multiproofs are one-shot witnesses for individual verifiers.

C. Beyond hash-only Merkle

More general, typically algebraic, vector-commitment schemes extend Merkle trees with stronger asymptotic guarantees. Polynomial-commitment based vector commitments [16] support constant-size commitments and logarithmic or constant-size proofs, often with efficient batch verification. Recent systems such as *Hyperproofs* [17] and *Reckle Trees* [18] further address the *maintenance* problem: they construct updatable batch proofs which deltas under state changes remain small, using recursive SNARK [19] composition and sophisticated proof graphs.

These non-hash-only methods offer powerful functionality, but at a cost. Many of the most efficient constructions rely on elliptic-curve pairings, trusted setup, and higher prover overhead than hash-based alternatives. In latency-sensitive, resource-constrained management functions such as IMF, this additional cryptographic complexity may make them less attractive than lightweight hash-only designs. Our design is conceptually inspired by their treatment of batch proof maintenance, but deliberately stays within hash functions only, no new cryptographic assumptions, and an explicit focus on long-lived multi-verifier workloads.

To our knowledge, no prior work has described a purely hash-based scheme that (i) maintains long-lived multiproofs for many verifiers, (ii) exposes a global sibling cache shared across batches, and (iii) supports patch-only updates whose cost is less dependent on the number of verifiers.

III. Preliminaries

The Merkle tree we consider is binary. Let N be the number of leaves. Let $i \in \{0, \dots, N-1\}$ be the leaf index. Let h denote the height of the Merkle tree and index levels by $\ell \in L = \{0, \dots, h\}$, with leaves at level 0 and the root at level h. Let p denote the tree node position that is a pair (ℓ, j) and let

$$I = \{\, (p) \,\} = \{\, (\ell, j) \mid \ell \in L,\ j \in \{0, \dots, 2^{h-\ell} - 1\} \,\}$$

be the set of all node positions in the tree. For each i we write $\mathsf{sibs}(i)$ for the set of all node positions that appear as siblings on the path from leaf i to the root in the tree, i.e., $\mathsf{sibs}(i) \subset I$ contains exactly one sibling position at each level along that path.

A. State Model and Incremental Commitment

At logical time t, the committed state is a vector:

$$s_t = (s_t[0], s_t[1], \dots, s_t[N-1])$$

where each entry $s_t[i]$ is serialised to bytes via a deterministic encoding function $\mathsf{encode}(\cdot)$ then to digest via a collision-resistant hash function $H(\cdot)$. A Merkle tree over these leaves yields a root R_t that binds the entire vector. Updates are single-leaf modifications of the form:

$$s_{t+1}[i] \leftarrow s'_i, \qquad s_{t+1}[w] = s_t[w] \text{ for all } w \neq i.$$

The Merkle tree we consider is maintained incrementally: after changing the leaf hash at index i, only the nodes on the path from that leaf to the root are recomputed. We denote by:

$$\Delta(i) = \{(\ell, j_\ell) \subset I\}$$

the set of tree positions whose hashes change as a consequence of updating index i. For a complete binary tree, $|\Delta(i)|$ is exactly the path from leaf i to the root and has size $O(\log N)$. This incremental commitment method is shared by all strategies we evaluate. The difference lies in how they represent and maintain the proof state.

B. Multiproof

Let $\mathcal{V}$ be the set of verifiers, and let each verifier $v \in \mathcal{V}$ be associated with a batch

$$B_v \subseteq \{0, \ldots, i, \ldots, N-1\}$$

of leaf indices it wants to verify against the current root R_t. A *multiproof* for B_v is an object π_v for (B_v, s_t, R_t) at time t which, together with the claimed leaf values $\{s_t[i]\}_{i \in B_v}$, enables v to run Merkle membership proof test for every $i \in B_v$, by recomputing the Merkle root from $H(\mathsf{encode}(s_t[i]))$ and the path information contained in π_v yields R_t.

Verification succeeds iff, for every $i \in B_v$, the verifier can derive a hash chain

$$\big(h_0^{(i)}, h_1^{(i)}, \ldots, h_\ell^{(i)}, \ldots, h_h^{(i)}\big)$$

satisfying

$$h_0^{(i)} = H\big(\mathsf{encode}(s_t[i])\big)$$

and, for every level ℓ in the binary tree there exists a sibling hash $b_\ell^{(i)}$ and a direction bit $d_\ell^{(i)} \in \{0,1\}$, both determined by π_v, such that

$$h_{\ell+1}^{(i)} = \begin{cases} H\big(h_\ell^{(i)} \,\|\, b_\ell^{(i)}\big) & \text{if } d_\ell^{(i)} = 0, \\ H\big(b_\ell^{(i)} \,\|\, h_\ell^{(i)}\big) & \text{if } d_\ell^{(i)} = 1. \end{cases}$$

Formally,

$$\mathsf{verify\_multiproof}\big(B_v, \{s_t[i]\}_{i \in B_v}, \pi_v, R_t\big) = 1$$

iff

$$h_h^{(i)} = R_t \quad \text{for all } i \in B_v.$$

C. Proof State

We use the term *proof state*, denoted $\mathcal{S}$, for the per-verifier data required to verify membership against the current root by recomputing each π_v. The concept is defined differently such as *Merkle proof* [20] or *Witness* [21]. We define the proof state at time t as a collection

$$\mathcal{S}_t = \{x_1, x_2, \ldots\}$$

where each proof state entry $x \in \mathcal{S}_t$ stores the hash of a tree node h_x together with its node position. We model

$$x = (\mathrm{pos}(x), h_x),$$

where $\mathrm{pos}(x)$ for the node position of x.

Only proof state entries whose positions lie in $\Delta(i)$ become outdated and must be refreshed upon an update at leaf i. We therefore define the per-update patch cost at leaf i as

$$m(i) \;=\; \big|\{\, x \in \mathcal{S}_t \mid \mathrm{pos}(x) \in \Delta(i) \,\}\big|.$$

Intuitively, $\Delta(i)$ measures how much work done by the incremental commitment, while $m(i)$ measures the additional work induced by the chosen proof representation as well as the chosen proof scheme.

For the rest of the paper, it is convenient to conceptually separate *which node positions* appear in the proof state from *which hash values* they currently carry. We refer to the former as the *proof structure* (a fixed set of node positions and per-verifier path layouts) and to the latter as the *proof contents* (the stored hash value h_x at those positions). Baseline schemes in the paper make different choices about how much of this structure is shared and how the contents are maintained under updates, but all of them fit into this common model.

D. Global Sibling Set

The proof structure induced by a collection of verifier batches is captured by the global set of sibling positions

$$P \;=\; \bigcup_{v \in V} \bigcup_{i \in B_v} \mathsf{sibs}(i) \;\subseteq\; I.$$

From the perspective of classical multiproofs, P coincides with the set of internal node positions that would appear for the union of all queried leaves $U = \bigcup_{v \in V} B_v$. Any proof scheme that verifies all leaves in U via Merkle membership test must verify exactly the same siblings in P. In that sense, P captures the information-theoretic minimum set of siblings for this global workload.

E. Proof Maintenance Schemes

Different proof schemes induce different shapes for $\mathcal{S}_t$. The two selected schemes below differ in how they store and update these proofs:

- **Per-path incremental proof.** The Merkle root is updated in $O(\log N)$, but proofs are still represented as independent paths. After each update, the prover scans all paths and refreshes any sibling whose position lies in $\Delta(i)$. Shared siblings are updated once *per proof* in which they appear.
- **Per-verifier multiproof.** For each batch, the prover builds a dictionary of siblings keyed by their tree positions and deduplicates overlaps within that batch. This compresses proofs but does not, by itself, provide a structured patch operation; maintaining long-lived proofs still requires recomputing or re-deriving the dictionary after each update.

In a case of frequent updates and many overlapping batches, these designs either pay repeatedly for duplicated work (per-path) or lack a notion of frequent updates to long-lived proofs (per-verifier).

IV. Problem Formulation

We consider a multi-vendor and multi-IMF intent pipeline following 3GPP TS 28.312 where an intent is negotiated and then repeatedly evaluated in a closed loop. Utility reports are produced over time, and one or more verifiers (e.g., consumer-side monitors, auditors, or downstream IMFs) validate that reported utility is consistent with the negotiated utility semantics. The standard separates what the consumer declares in the intent expectations from what the producer actually optimises in its utility computation. The expectation domain $E = \{e_1, \ldots, e_u\}$ captures the KPIs on which the consumer declares acceptance , while the utility domain $K = \{k_1, \ldots, k_n\}$ is the KPI set referenced by the `IntentUtilityFormula`. During fulfilment, the producer evaluates a numeric utility

$$U_{\text{benign}} = f(k_1, \ldots, k_n; w_1, \ldots, w_n)$$

where $W = \{w_1, \ldots, w_n\}$ are the internal weights. The standard does not require $K \subseteq E$ nor any semantic relation between E and K. The producer may optimise over latent KPIs in $K \setminus E$ while still returning syntactically valid utility reports and fulfilment summaries.

A. Utility-Expectation Gap

This freedom may induce a gap: different fulfilment behaviours can produce identical numeric utilities, and the consumer has no evidence of which KPI set and weights were used when a given report was produced. In a multi-IMF pipeline, the situation worsens: each IMF may re-negotiate utility formulas with downstream modules, so the utility observed at the top-level may have traversed a sequence of opaque transformations before it is used for optimization.

B. Threat Model and Adversarial Capability

We assume an *injection adversary* who can compromise or influence one or more intermediate components in the pipeline (ingress, decomposition/translation, propagation, delivery, or reporting), and can therefore modify or substitute pipeline artifacts (e.g., utility formula parameters or utility reports).

Let K_0 denote the KPI domain of the utility formula agreed at negotiation time, and let K_t denote the KPI domain actually used during fulfilment at step t. TS 28.312 does not constrain their relation. We write the *domain deviation*

$$\Delta_{\text{dom}}(t) = K_t \setminus K_0.$$

Whenever $\Delta_{\text{dom}}(t) \neq \varnothing$, the producer is optimizing over additional KPIs that the consumer never authorised as part of the utility formulas. The consumer still observes only a numerical utility value U and a subset of KPIs E, but the effect of the latent KPIs in $\Delta_{\text{dom}}(t)$ is not observable.

An adversary can therefore perform *utility injection* by adding latent KPIs $L_t = \{\ell_1, \ldots, \ell_m\}$ with attacker-chosen weights q:

$$U_{\text{attack}} = U_{\text{base}} + f(\ell_1, \ldots, \ell_m; q_1, \ldots, q_m)$$

to cause a systemic pressure. As long as U_{attack} preserves expected scores on the observable expectations in E, the consumer cannot distinguish benign from manipulated fulfilment trajectories.

The adversary cannot break the collision resistance of the hash function $H(\cdot)$ used by the Merkle commitment, and cannot produce a valid proof for a root it does not possess.

C. Requirements and Limitations of Merkle-Based Provenance

In order to mitigate the utility–expectation gap in a TS 28.312-style intent framework requires turning the question "which utility formulas were used?" into something verifiable over time. A practical hash-based mechanism for IBN must satisfy the following requirements:

(R1) **Selective disclosure**: Each verifier v should obtain only the proof material needed to verify its own batch B_v (i.e., the positions P_v and corresponding hash labels), without requiring a single shared multiproof object for the union $S = \bigcup_v B_v$. We emphasize that this is a separation requirement on proof material. Hiding the existence of other verifiers' monitored positions (query privacy) depends on how the shared sibling map is disseminated. Merkle inclusion proofs naturally support such disclosure: everyone agrees on R_t, but each verifier only learns the leaves and siblings relevant to its own checks.

(R2) **Local maintenance under frequent updates**: Utility functions and KPI values may change at the timescale of the optimisation loop. A usable mechanism must handle frequent updates without rebuilding global state or re-sending full proofs. Incremental Merkle trees already allow $O(\log N)$ root updates for a single leaf change; for many long-lived, overlapping verifiers we additionally want proof-maintenance cost to scale with the number of *distinct* affected siblings, not with the total size or count of verifier batches.

(R3) **Lightweight deployability**: The scheme should be implementable using only standard hash functions without heavy machinery, because provenance checks run inside resource-constrained management functions.

Existing Merkle-based schemes only partially meet these requirements:

(i) **Per-verifier paths**: Maintaining independent Merkle paths for each leaf in each batch and with an incremental tree, gives $O(\log N)$ root updates. However, each proof is patched separately: when a leaf changes, every proof touching the corresponding path must be updated, so patch cost scales with the *sum* of all proof sizes across verifiers, violating R2.

(ii) **Per-verifier multiproofs**: Compact Merkle multiproofs deduplicate overlapping siblings *within* a single batch B_v, shrinking individual proofs while preserving per-verifier isolation (R1). In a multi-verifier setting, however, the total proof-side state and patch cost still scale with $\sum_v |G_v|$, where G_v is the sibling set for B_v, because overlapping siblings are stored and updated multiple times across verifiers. R2 remains unmet.

(iii) **Global multiproofs**: Conceptually, one could build a single compact multiproof for the union $S = \bigcup_v B_v$, obtaining a global sibling set that matches the information-theoretic

minimum number of distinct siblings and ideal patch cost for S. Yet exposing such a shared proof object reveals which positions other verifiers monitor, contradicting R1.

(iv) **Beyond hash-only Merkle**: SNARK-based Merkle variants can satisfy R1–R2, but rely on heavy algebraic machinery and complex implementations. This violates R3.

D. Formal Problem Statement

Across multiple IMFs, the negotiated utility function (U_0, K_0, W_0) evolves along a chain of transformations

$$(U_0, K_0, W_0) \to (U_1, K_1, W_1) \to \cdots \to (U_i, K_i, W_i),$$

where each IMF may introduce additional KPIs $K_i = K_{i-1} \cup L_i$ with $L_i \neq \varnothing$ and apply local weighting and aggregation rules. At fulfilment time, the top-level consumer observes only a numerical utility value U and a subset of KPIs E.

Goal. Design a *hash-only* mechanism that (i) preserves per-verifier views, (ii) supports high-frequency updates with patch cost proportional to the number of distinct affected siblings and less dependent on the number of verifiers, and (iii) supports lightweight and hash-only deployability across a intent processing pipeline involving multiple IMFs.

V. Proposed Method

We introduce *PICKLE* (Patchable InCremental multiproof merKLE tree), a purely hash-based construction for maintaining Merkle batch proofs under frequent updates and long-lived verifiers. PICKLE addresses the issues in Section IV by making the proof state explicitly patchable. We view a proof state as consisting of a proof *structure* (the set of node positions and per-verifier path layouts) together with the hash *contents* currently stored at those positions.

At a high level, PICKLE fixes the structure induced by the verifier batches (the global sibling set P and the per-verifier paths through it) and lets only the contents evolve over time. In particular, PICKLE maintains a single global map from node positions in P to hash labels, while each verifier holds only a position-only proof that indexes into this map. When the underlying state vector changes, the verifier proofs remain structurally unchanged and only the corresponding entries in the global map are patched. Importantly, our core isolation goal (R1) is *Selective disclosure*: each verifier holds a fixed position-only witness π_v and validates updates locally by reading the required hashes from G_t. Verifiers do not share proofs with one another and do not need to trust other verifiers' state. PICKLE defines the proof state $\mathcal{S}_t$ as two components:

1) a single global sibling map S_t that stores one hash value for each $(\ell, j) \in P$ (the proof contents) at logical time t, and
2) a family of position-only proofs $\{\pi_v\}_{v \in V}$ that list which positions in P each verifier uses (the proof structure).

Per-leaf updates are handled by patching the map S_t along the affected Merkle path; the position-only proofs π_v remain unchanged as long as the verifier batches stay fixed. Unlike most existing multiproof work, which focuses on one-shot proof size for a static tree, we treat *the cost of maintaining a large population of such proofs under updates* as a first-class metric.

A. Design Goals

PICKLE is designed for settings with (i) a shared state vector committed by a Merkle root, (ii) many long-lived verifiers that each care about overlapping subsets of leaves, and (iii) frequent single-leaf updates. In this regime, naive strategies either duplicate proof material across verifiers or incur patch cost that scales with the total proof size. Our design is guided by the following goals.

Multi-verifier scalability: The construction should support many verifiers with possibly overlapping batches. Shared siblings must be stored and patched once, not duplicated across proofs. In particular, the total proof-side state and per-update cost should depend on the *union* of all sibling sets, i.e., the global structure P, not on $\sum_v |B_v|$.

Patch-efficient updates: Once initial proofs have been issued, changes to the state should be reflected as small patches to existing proofs rather than by regenerating and re-sending full multiproofs after every update. The patch size should be bounded by the number of changed siblings that are actually referenced by some verifier.

Hash-only simplicity: The construction should rely only on a collision-resistant hash function and standard Merkle-tree operations, without trusted setup, pairings, or specialised algebraic assumptions. This keeps PICKLE deployable in existing hash-only codebases and hardware.

B. Root Update

Root update remains Incremental Merkle tree update: for an update at leaf index i, it recomputes the hash values along the path from leaf i to the root.

C. Per-verifier Position-Only Proof States

PICKLE first defines the *global sibling map* at time t as the restriction of h_t to the global sibling set P:

$$G_t : P \to \{0,1\}^\lambda, \qquad G_t(\ell, j) \;=\; h_t(\ell, j) \;\text{ for all } (\ell, j) \in P.$$

It means there is exactly one entry in G_t per sibling position, regardless of how many verifiers depend on it.

For each verifier v with batch B_v, we first compute the verifier-local sibling set

$$P_v \;=\; \bigcup_{i \in B_v} \mathsf{sibs}(i) \;\subseteq\; P.$$

This step uses only the leaf indices i and the index arithmetic $\mathsf{sibs}(\cdot)$; no hash computations are required.

We then define the position-only proof as a sorted list of positions:

$$\pi_v \;=\; \big[\, p_1, \ldots, p_L \,\big] \;=\; \big[\, (\ell_1, j_1), \ldots, (\ell_L, j_L) \,\big],$$

with $\{p_1, \ldots, p_L\} = P_v$. The proof π_v thus encodes only the *structure* of the verifier's multiproof (which node positions it will use); all hash values live in the global map G_t.

TABLE II: Complexity of the proof maintenance schemes per verifier.

Scheme	Root Update	Proof Maintenance	Per-update COMM	Default Proof Size	Verifier Cost
Per-path	update path $O(\log n)$	rebuild $O(k \log n)$ per batch	$O(k \log n)$	$O(k \log n)$	$O(k \log n)$
Per-verifier	update path $O(\log n)$	rebuild $O(s)$ per multiproof	$O(s)$	$O(s)$	$O(k \log n)$
PICKLE	update path $O(\log n)$	patch $O(m)$	$O(m)$	$O(m)$	$O(k \log n)$

When a verifier is remote and cannot query G_t directly, the prover can serialize the pair

$$\big(\, \pi_v,\ \{\, G_t(p) \mid p \in P_v \,\}\,\big)$$

as a conventional Merkle multiproof at issuance time. Internally, however, the prover maintains only the shared map G_t and the fixed position-only proofs $\{\pi_v\}_{v \in V}$, and patches G_t as the state evolves.

D. Localized Proof Patching

We write

$$\Delta(i) \subseteq I$$

for the set of node positions whose labels change under this update. In the simple case where every node on the path is recomputed, $\Delta(i)$ coincides with $\{\mathsf{pos}_\ell(i) \mid 0 \le \ell \le h\}$.

By definition of the abstract patch cost, for an update at leaf i we count how many proof-state entries lie on the updated path:

$$m(i) \;=\; \big|\{\, x \in S_t \mid \mathrm{pos}(x) \in \Delta(i)\,\}\big|.$$

In PICKLE, the only node positions that have associated proof-state entries are those in the global sibling set P, because G_t is defined exactly on P. Consequently, the positions that both (i) change under the update at i and (ii) appear in the proof state are exactly $\Delta(i) \cap P$, and the expression simplifies to

$$m(i) \;=\; \big|\Delta(i) \cap P\big|.$$

The patch cost counts exactly those changed nodes on the Merkle path for leaf i that have been selected as siblings by at least one verifier. Concrete patching in PICKLE proceeds as follows.

1) Apply the incremental Merkle update at leaf index i to obtain the new tree labelling h_{t+1} and root R_{t+1}, computing $\Delta(i)$ along the way. This costs $O(\log N)$ hash evaluations.
2) For every position $p = (\ell, j) \in \Delta(i)$ such that $p \in P$, overwrite the corresponding entry in the global sibling map:

$$G_{t+1}(p) \;=\; G_{t+1}(\ell, j) \leftarrow h_{t+1}(\ell, j).$$

   Positions on the path that are not in P (i.e., never used as siblings for any batch) are ignored, as they are not part of the proof state.
3) Leave all position-only proofs $\{\pi_v\}_{v \in V}$ unchanged as long as the verifier batches $\{B_v\}_{v \in V}$ remain fixed.

By the discussion above, the per-update patch cost is

$$C_{\mathsf{patch}}(i) \;=\; \big|\Delta(i) \cap P\big| \;=\; m(i) \;\le\; |\Delta(i)| \;=\; O(\log N),$$

with the constant determined by how many of the nodes on the updated path have actually been selected as siblings in the global workload.

Intuitively, the commitment side always touches all nodes on the path $\Delta(i)$ from leaf i to the root, whereas the proof side only touches those nodes that both lie on that path and appear in the global sibling set P. Crucially, this patch cost does not grow with the number of verifiers $|V|$ or the sum of batch sizes $\sum_v |B_v|$.

E. Verification

Verification remains the standard Merkle membership test: given a batch B_v, leaf values $\{s_t[i]\}_{i \in B_v}$, a fixed position-only proof π_v, and the relevant sibling hashes read from the latest map S_t, the verifier reconstructs the root and accepts if and only if the result equals the R_t.

F. Complexity

Let n the number of leaves in the Merkle tree. Let k the batch size $|B_v|$ for one verifier. Let s the number of internal sibling nodes that would appear for S, i.e., $|P|$. For a single batch, $s \le k \log n$. For multiple batches, s will be smaller if paths overlap. Let m the worst per-update patch cost,

$$m \;=\; \max_i m(i) \;\le\; \max_i \Delta(i) \cap P \;\le\; \max_i \Delta(i),$$

thus $m \le \log n$ and $m \le s$. Using these symbols, the complexity of PICKLE and the baselines are summarised in Table II.

Proof Maintenance: PICKLE updates only positions in $\Delta(i)$, thus we denote as $O(m)$, which is $O(m) \le O(\log n)$. Likewise, we denote as $O(s)$ for the per-verifier scheme, which is $O(s) \le O(\log n)$.

Per-update Communication: Every time a leaf in the Merkle tree gets updated, it has to refresh the proofs that verifiers hold, sending hashes to each verifier. PICKLE only sends updated entries in the global table, at most m per update, while the per-path scheme resends all $k \log n$ siblings and the per-verifier resends the whole multiproof of size s.

Stored Proof Size by Default: The space overhead to store the proof by default, that is tied up over updates. PICKLE stores the global sibling map S_t, which has one entry per sibling used by any verifier, while the per-path scheme stores k independent paths, each of them $\log n$ and the per-verifier stores one deduplicated sibling set per batch, that is $O(s)$.

VI. Intent Driven MnS Adaptation

Figure 1 shows a set of IMFs $F = F_1, \ldots, F_n$ that process and refine intents. On each interface $F_i \rightarrow F_{i+1}$, the left module acts as consumer for that edge and the right module

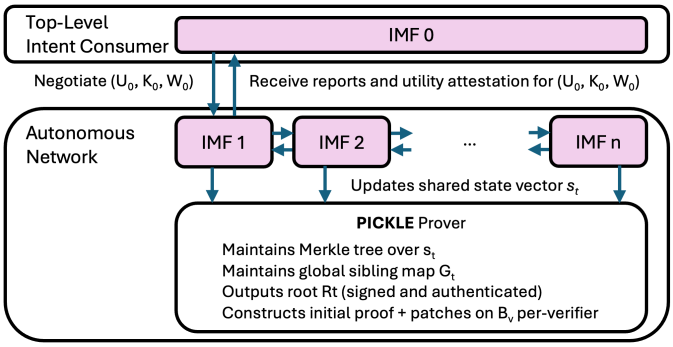

Fig. 1: Intent Driven MnS using PICKLE

as producer of the upstream configuration. All modules share a global state vector s_t, which contains utility configuration fragments and KPI bindings. A single PICKLE prover in this domain maintains a Merkle tree over s_t and exports the root R_t as a state commitment. Throughout F, we model a set of long-lived verifiers $V = v_1, v_2, \ldots$ which examine utility intents throughout its lifecycle in the closed-loop.

For each report on $F_{i+1} \rightarrow F_i$ and verifier $v \in V$, the producer sends a *utility attestation*

$$\mathsf{Attest}_{v,t} = \big(U_t, K_t, W_t, y_t, \pi_v(t), R_t\big),$$

where U_t is the utility intent, K_t the KPI binding set, W_t the internal weights, y_t the KPI value vector, $\pi_v(t)$ the PICKLE proof state for v, and R_t the Merkle root over s_t. Each verifier maintains its own leaf set B_v and long-lived proof state $\pi_v(t)$. Upon receiving $\mathsf{Attest}_{v,t}$, before checking that (U_t, K_t, W_t) are consistent with the negotiated baseline (U_0, K_0) according to its local policy, v firstly checks that (U_t, K_t, W_t) are included in s_t under R_t using $\pi_v(t)$ to detect an intent injection attack.

VII. Experimental Evaluation

Our goals are to answer the following questions:

Q1 Under many verifiers over shared state, does PICKLE reduce proof maintenance cost compared to the baselines?
Q2 How does PICKLE scale with the tree size N?

A. Experiment Setup

We evaluate the two proof maintenance schemes from Section III-E with our PICKLE design, using an implementation of a binary Merkle tree. All strategies share the same code for tree layout and hashing to avoid implementation bias. We use a full binary tree over N leaves, store 32-byte hashes at all nodes, and use the compression function with SHA-256. Internal nodes are kept in a flat array representation, and leaves are initialised with independent random values. Unless stated otherwise, we use a tree with 4096 leaves, apply 2000 single-leaf updates, and each verifier monitor a fixed batch B_v of size 256.

TABLE III: Scaling with the number of verifiers V.

Scheme	V	$\log_2 m$	COMM [MB]	Time [s]
Per-path	1	11.585	15.622	0.845
	8	14.585	124.969	6.573
	32	16.585	499.877	26.784
Per-verifier	1	10.363	0.505	0.033
	8	13.387	4.039	0.063
	32	15.396	16.247	0.167
PICKLE	1	10.363	0.505	0.033
	8	12.271	0.668	0.033
	32	12.902	0.724	0.034

B. Cost Metrics

We report three classes of cost: *proof state size* ($\log_2 m$), *communication* (COMM [MB]), and *runtime* (Time [s]). To reduce noise from the hardware environment, each configuration is repeated over 10 times and we report the median across runs.

a) Proof State Size: Each scheme maintains proof-side state that must be stored either at the prover or jointly between prover and verifiers. We model state size as the number of distinct stored sibling hashes multiplied by the hash length. Concretely, we let m be the number of unique sibling nodes present in the scheme's proof data structure after initialization, and report $\log_2 m$ as a measure of structural complexity.

b) Communication: We measure the total *patch* traffic required to keep verification consistent over the U state updates, counting one sibling hash as 32 bytes and reporting MB. For the baselines, proofs are refreshed in a *push* model: the prover must transmit updated sibling hashes to each verifier that needs them, so the total update traffic is the aggregate $C = \sum_{v \in V} C^{(v)}_{\text{update}}$. For PICKLE, verifiers have read access to the shared sibling map G_t. An update consists of *publish* the modified entries of G_t once, with cost $C = \sum_{t=1}^{U} |\Delta(i_t) \cap P| \times 32$ bytes, which is not multiplied by $|V|$.

c) Runtime: For each leaf update we measure the wall-clock time.

C. Scaling with the Number of Verifiers

Table III shows how each proof-maintenance scheme scales with the number of verifiers V that consume proofs from the same Merkle tree.

The per-path scheme exhibits the expected linear scaling in V. When V increases from 1 to 32, $\log_2 m$ grows from 11.6 to 15.6 and patch communication grows from roughly 15.6 MB to 249.9 MB. Runtime follows the same trend, increasing from 0.83 s to 13.08 s, reflecting the fact that each additional verifier carries its own full set of patched siblings.

The per-verifier scheme mitigates this cost but still scales essentially linearly in V. As V grows from 1 to 32, $\log_2 m$ increases from 10.4 to 14.6, and patch communication rises from 0.51 MB to 14.27 MB. This represents a substantial improvement over the per-verifier, as for $V = 32$, the per-verifier scheme uses about $17\times$ less patch bandwidth than the

TABLE IV: Scaling with the number of leaves N.

Scheme	N	$\log_2 m$	COMM [MB]	Time [s]
Per-path	2048	16.459	499.756	24.373
	4096	16.585	499.877	26.857
	8192	16.700	499.939	29.248
Per-verifier	2048	15.123	16.218	0.154
	4096	15.396	16.247	0.168
	8192	15.632	16.263	0.171
PICKLE	2048	11.991	0.671	0.031
	4096	12.902	0.724	0.034
	8192	13.652	0.763	0.036

baseline, but the cost still grows roughly proportionally to the number of verifiers.

For $V = 1$, PICKLE matches the per-verifier scheme almost exactly: $\log_2 m = 10.4$ and patch communication of $0.51\,\text{MB}$, which is expected. However, as V increases to 32, PICKLE performs differently. Patch communication grows only modestly from $0.51\,\text{MB}$ to $0.70\,\text{MB}$, while $\log_2 m$ increases from 10.4 to 12.7. In other words, increasing the verifier population by a factor of 32 leads to only a $1.4\times$ increase in patch bandwidth, compared to roughly $16\times$ for the per-path scheme and $28\times$ for the per-verifier scheme. Runtime for PICKLE remains almost flat across all V (around $0.03\,\text{s}$), indicating that the overhead of managing the shared multiproof and pointer-based proofs is small.

D. Scaling in Tree Size

Table IV shows how the per-path scheme, the per-verifier scheme, and PICKLE perform as the tree grows. We fix the number of verifiers to $V = 32$ and vary the tree size over $N \in \{2048, 4096, 8192\}$. Note that the stored state grows as $m = k \log_2 N$, but reporting $\log_2 m = \log_2 k + \log_2(\log_2 N)$ compresses this dependence, making the increase across N visually small.

For the per-path scheme, $\log_2 m$ increases slightly from 16.459 to 16.700 as N grows, while total patch communication remains essentially flat (499.756–499.939 MB). Runtime increases from 24.373 s to 29.248 s with tree size.

For the per-verifier scheme, $\log_2 m$ also grows mildly with N, from 15.123 to 15.632, while patch communication is effectively constant at 16.218–16.263 MB. Runtime remains low and increases only slightly from 0.154 s to 0.171 s.

PICKLE exhibits the same locality behavior but at a much lower scale. Its effective proof size increases from $\log_2 m = 11.991$ to 13.652, with patch communication staying near-constant at 0.671–0.763 MB. Runtime increases marginally from 0.031 s to 0.036 s. Thus, PICKLE shifts the operating point down substantially in both communication and runtime compared to the baselines.

VIII. Discussion

An important point is that PICKLE *does not claim to beat baseline schemes on the theoretical minimum number of distinct siblings*. For a fixed set of queried leaves $S = \bigcup_{v \in V} B_v$, any Merkle-compatible multiproof must touch the same minimal set of sibling positions P, and no hash-only scheme can use fewer distinct hashes while still supporting standard Merkle verification. Our contribution is therefore not a further reduction in this lower bound, but a new way of *realising* it in a setting with many long-lived and mutually distrustful verifiers: PICKLE attains the same minimum on distinct siblings as an ideal compact multiproof for the union of all verifier queries, while (i) preserving per-verifier isolation via position-only proofs, and (ii) supporting patch-only maintenance whose cost depends on the number of affected siblings rather than on the number or size of verifiers' batches.

Nonetheless, the combination of specification analysis and performance benchmarking suggests that provenance-aware utility semantics represent a practical path toward interpretable and auditably secure utility intent fulfilment.

A. Limitations

PICKLE currently assumes verifiers receive all incremental patches. If a verifier misses patches and becomes desynchronized, a practical recovery path is periodic checkpointing with re-anchoring to the latest root. The global map's footprint is memory-dominant and may introduce write contention bottlenecks at high update rates, which batched epochs can be a mitigation. While integrity holds, access patterns to shared proof material may leak which leaves are monitored. Proxy aggregation can reduce such metadata leakage.

IX. Conclusion

The freedom and opacity of intent processing pipelines may introduce ambiguity and novel attack surfaces, particularly when automated components reinterpret declarative goals without transparency.

To mitigate this potential security implication, we proposed PICKLE, a lightweight and hash-transparent provenance mechanism that cryptographically binds the negotiated utility semantics to the fulfilment-time computation. PICKLE uses an incremental Merkle-tree construction to commit to the full KPI and weight structure and maintains a cache of Merkle multipaths updated via a patch-only approach. Our implementation demonstrates that PICKLE achieves significant speedups over per-path and per-verifier schemes, and compatibility with the low-latency requirements of intent processing pipelines. By making utility intent auditable, PICKLE transforms fulfilment into one with verifiable mechanism.

Future work includes specializing the proof scheme to utility formulas. One option is to aggregate proofs across related utility intents and shared sub-expressions, which would further improves scalability. The patch-only mechanism may also be extended to sparse Merkle trees.

Acknowledgment

This work is funded by Sweden's Innovation Agency under grant agreement No. 2025-02987 (SUSTAINET) and the European Union's Horizon Europe Research and Innovation Programme under grant agreement No. 101135576 (INTEND).

REFERENCES

[1] 3GPP, "Management and orchestration; Intent driven management services for mobile networks, TS 28.312, version 19.1.0," 3rd Generation Partnership Project (3GPP), Tech. Rep.

[2] D. Manheim and S. Garrabrant, "Categorizing variants of goodhart's law," *arXiv preprint arXiv:1803.04585*, 2018.

[3] R. Ngo, L. Chan, and S. Mindermann, "The alignment problem from a deep learning perspective," 2025. [Online]. Available: https://arxiv.org/abs/2209.00626

[4] J. Leike, D. Krueger, T. Everitt, M. Martic, V. Maini, and S. Legg, "Scalable agent alignment via reward modeling: a research direction," 2018. [Online]. Available: https://arxiv.org/abs/1811.07871

[5] R. Kumar, J. Uesato, R. Ngo, T. Everitt, V. Krakovna, and S. Legg, "Realab: An embedded perspective on tampering," 2020. [Online]. Available: https://arxiv.org/abs/2011.08820

[6] T. Everitt, M. Hutter, R. Kumar, and V. Krakovna, "Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective," *Synthese*, vol. 198, no. Suppl 27, pp. 6435–6467, 2021.

[7] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016.

[8] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Rfc 9315: Intent-based networking-concepts and definitions," 2022.

[9] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency (rfc 6962)," *IETF RCFs*, 2013.

[10] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging." in *USENIX security symposium*, 2009, pp. 317–334.

[11] B. E. Ujcich, A. Miller, A. Bates, and W. H. Sanders, "Towards an accountable software-defined networking architecture," in *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–5.

[12] J. An, Z. Zhu, I. Miers, and Z. Liu, "Towards verifiable network telemetry without special purpose hardware," in *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, 2025, pp. 411–418.

[13] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[14] F. Cassez, "Verification of the incremental merkle tree algorithm with dafny," 2021. [Online]. Available: https://arxiv.org/abs/2105.06009

[15] L. Ramabaja and A. Avdullahu, "Compact merkle multiproofs," *arXiv preprint arXiv:2002.07648*, 2020.

[16] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[17] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang, "Hyperproofs: Aggregating and maintaining proofs in vector commitments," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3001–3018.

[18] C. Papamanthou, S. Srinivasan, N. Gailly, I. Hishon-Rezaizadeh, A. Salumets, and S. Golemac, "Reckle trees: Updatable merkle batch proofs with applications," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1538–1551.

[19] S. Bowe, J. Grigg, and D. Hopwood, "Recursive proof composition without a trusted setup," *Cryptology ePrint Archive*, 2019.

[20] O. Kuznetsov, A. Rusnak, A. Yezhov, D. Kanonik, K. Kuznetsova, and O. Domin, "Efficient and universal merkle tree inclusion proofs via or aggregation," *Cryptography*, vol. 8, no. 3, p. 28, 2024.

[21] M. Christ and J. Bonneau, "Limits on revocable proof systems, with implications for stateless blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2023, pp. 54–71.