VERDICT: A Cryptographically Verifiable Framework for Secure Data Lineage in Decentralized DAGs

Bilel Zaghdoudi
LIP6, CNRS, Sorbonne Université
Paris, France
bilel.zaghdoudi@lip6.fr

Maria Potop-Butucaru
LIP6, CNRS, Sorbonne Université
Paris, France
maria.potop-butucaru@lip6.fr

ABSTRACT

Ensuring the integrity and traceability of data transformations in distributed systems presents significant challenges, particularly in environments where data privacy and decentralization are paramount. This paper introduces a novel secure lineage verification system based on Directed Acyclic Graphs (DAGs) and homomorphic hash functions, VERDICT. Our approach represents data artifacts and their transformations as two interconnected DAGs: a Data DAG tracking data dependencies and an Event DAG capturing causality between transformation events. We propose a level-based DAG compression technique that decomposes these graphs based on distance from genesis nodes, enabling efficient verification through skip DAG structures. The system incorporates bucket indexing and Merkle tree verification to provide cryptographic guarantees of data and event existence. We present formal algorithms for DAG construction, level-based hashing, skip DAG traversal, and verification processes. Security analysis demonstrates the system's resistance to tampering and modification attacks while maintaining privacy. Replay attacks are prevented through an application-layer challenge-response mechanism. Our approach has significant applications in federated learning environments and decentralized architectures, where it can serve as a notary component for tracing events without compromising data confidentiality. Theoretical analysis shows that our method achieves verification in $O(\log n)$ time in the number of DAG levels, independent of the number of nodes per level, making it suitable for large-scale distributed systems.

1 INTRODUCTION

In today's increasingly distributed computing environments, ensuring the integrity and traceability of data transformations has become a critical challenge. As data moves through complex processing pipelines across decentralized networks [26], maintaining a verifiable record of its lineage–the sequence of transformations it undergoes–is essential for establishing trust, enabling accountability, and supporting reproducibility. This challenge is particularly acute in environments where data privacy is paramount, such as federated learning systems and decentralized architectures, where traditional centralized verification approaches are inadequate or impractical. The problem of secure lineage verification encompasses several key requirements: (1) tracking the provenance of data artifacts through their transformations, (2) establishing causal relationships between transformation events, (3) providing cryptographic guarantees of integrity without compromising privacy, and (4) enabling efficient verification in large-scale distributed systems. Existing approaches to this problem often sacrifice either security, privacy, or scalability, making them unsuitable for modern distributed applications.

Traditional methods for ensuring data integrity in distributed systems typically rely on digital signatures applied to either individual updates or the entire dataset. While signing each update provides strong guarantees for that specific transformation, it creates significant overhead when verifying a sequence of updates or performing offline validation. Conversely, signing the entire dataset after each update requires iterating over all data for signature generation and verification, which does not scale with increasing data size or update frequency. Merkle trees offer a partial solution by enabling logarithmic-time updates to dataset signatures, but they require maintaining the entire tree structure in memory, imposing substantial memory overhead. Furthermore, existing approaches often fail to capture the causal relationships between transformation events, limiting their utility for complex lineage tracking [12, 17].

Recent work on Conflict-free Replicated Data Types (CRDTs) [3] for Directed Acyclic Graphs (DAGs) has addressed some challenges in maintaining DAG integrity in distributed systems, but these approaches typically focus on structural consistency rather than cryptographic verification of lineage. Similarly, research on secure data provenance in federated learning has primarily addressed privacy concerns without providing efficient mechanisms for lineage verification.

Our Approach: In this paper, we present a novel secure lineage verification system based on Directed Acyclic Graphs (DAGs) and homomorphic hash functions. Our approach represents both data artifacts and their transformations as two interconnected DAGs: a Data DAG that tracks data dependencies through lineage events, and an Event DAG that captures causality between transformation events. This dual-DAG representation provides a comprehensive model of data lineage that supports both artifact-centric and process-centric verification. The key innovation of our system lies in its level-based DAG compression technique, which decomposes these graphs based on distance from genesis nodes and applies homomorphic hashing to create efficient verification structures. By leveraging the properties of homomorphic hash functions, our system enables the computation of hash values for updated DAGs using only the hash of the previous version and the update; each level-hash update costs $O(1)$, independent of the number of nodes in that level. We further enhance verification efficiency through skip DAG construction, which applies skip list-like reduction to the hashed level DAGs, and bucket-based indexing with Merkle tree verification. These techniques collectively enable efficient verification of data and event existence while maintaining strong security guarantees.

Contributions: In this paper we present VERDICT, a cryptographically verifiable framework for secure data lineage having the following advantages with respect to the state of the art:

(1) **Dual-DAG Representation.** VERDICT introduces a comprehensive model for secure lineage verification using two interconnected DAGs–a Data DAG for artifact dependencies and an Event DAG for transformation causality–that together support both artifact-centric and process-centric verification queries.
(2) **Level-Based ECMH Compression.** VERDICT proposes a level-based DAG decomposition scheme in which each level is hashed using an Elliptic Curve Multiset Hash (ECMH), enabling order-independent, $O(1)$ incremental updates and $O(|V|)$ initial construction–without requiring canonical ordering of nodes across distributed participants.
(3) **Deterministic Skip DAG with $O(\log n)$ Path Proofs.** VERDICT is built on top of a deterministic binary skip structure over the hashed level sequence, in which each element carries a single signed path proof of length $\lfloor \log_2 n \rfloor + 1$ from the genesis level to its assigned level. This enables worst-case $O(\log n)$ verification of level membership, independent of DAG width.
(4) **Bucket-Indexed Merkle Verification.** VERDICT contains a hash-based bucket indexing scheme combined with per-level Merkle trees and signed roots, providing cryptographic membership proofs for individual elements within a verified level.
(5) **Formal Security Analysis.** Finally, we provide a security analysis of VERDICT under the Dolev-Yao model [6], proving data integrity, event authenticity, lineage integrity, non-repudiation, and privacy preservation, with explicit reductions to the collision resistance of ECMH and the unforgeability of the signature scheme.

Paper Organization: Section 2 reviews related work on data provenance, blockchain-based lineage, and authenticated data structures. Section 3 presents the VERDICT system architecture and its components. Section 4 defines the dual-DAG model and the level-based ECMH compression scheme. Section 5 introduces the deterministic skip DAG construction, insertion, and verification algorithms. Section 6 describes bucket indexing and Merkle tree verification. Section 7 presents the performance evaluation. Section 8 provides the formal security analysis. Section 9 concludes and outlines future work.

2 RELATED WORK

The challenge of ensuring data integrity and traceability in decentralized environments has led to several specialized approaches in the fields of blockchain-based provenance and cryptographic data structures [22]. The formal characterization of data lineage, particularly the distinction between "why" and "where" provenance, was established by Buneman et al. [4], while Moreau and Groth [14] provided a comprehensive introduction to the W3C PROV standards [15]. Our dual-DAG model aligns with the PROV-DM entities and activities, where the Data DAG maps to entities and the Event DAG maps to activities. Furthermore, our skip-structure approach builds on the theoretical foundation of Authenticated Data Structures (ADS) pioneered by Tamassia [23], extending skip-list-based authentication to directed acyclic structures.

Foundational work in provenance-aware storage, such as PASS [16] and SPADE [7], established the necessity of auditing transformations in distributed environments. In modern big data contexts, platforms like Apache Spark [26] utilize Directed Acyclic Graphs (DAGs) to manage complex processing pipelines. However, these methods often focus on execution efficiency rather than providing the cryptographic verification required for decentralized trust. Since the advent of Bitcoin [17], Merkle-based structures [12] have been the standard for ensuring data integrity through digital signatures. Systems like ProvChain [27] and BlockProvenance [19] extend this to cloud environments. Furthermore, log-based systems such as Certificate Transparency [11] demonstrate the power of public auditability. While these systems provide anchoring, traditional methods typically rely on digital signatures applied to individual updates, creating significant overhead during verification.

The shift toward non-linear structures is seen in DAG-based ledgers like IOTA's Tangle [20] and Hedera Hashgraph [10]. While CRDTs for DAGs [3] address structural consistency in distributed settings, they typically prioritize eventual consistency over cryptographic lineage proof. To address the efficiency gap, VERDICT leverages incremental hashing techniques similar to LtHash [13], allowing hash values for updated DAGs to be computed using only the previous version's hash, making complexity independent of the total history size.

Recent research in Federated Learning (FL), such as FedSOV [9] and other blockchain-backed models [8], primarily addresses ownership and privacy. However, these approaches frequently sacrifice scalability or fail to capture the deep causal relationships between transformation events. VERDICT bridges this gap by introducing a dual-DAG representation–a Data DAG for dependencies and an Event DAG for causality complemented by efficient Skip DAG traversal [21] and QMDB-style [25] bucketed Merkle verification.

ForensiBlock [1] proposes a blockchain-backed provenance framework targeting data forensics and auditability in cloud environments. It integrates Role-Based Access Control with Smart Authorisation (RBAC-SA) and distributed Merkle roots to authenticate forensic chains of custody. While ForensiBlock achieves tamper-evidence and role-scoped auditability, its design is anchored to on-chain smart contracts and is evaluated only on small case sets (100–1,000 forensic cases), limiting scalability. It does not model causal relationships between transformation events and lacks a compact off-chain verification path. VERDICT differs by operating entirely off-chain, scaling to tens of thousands of DAG nodes, and providing deterministic $O(\log n)$ level-membership proofs rather than relying on smart-contract execution for every audit operation.

Recent advancements in zero-knowledge provenance systems [2, 18] enable privacy-preserving verification of data provenance and transformation lineage through cryptographic proofs. However, these approaches incur substantial proof-generation overhead, often requiring orders of magnitude more computation than standard hashing operations. This overhead becomes a critical bottleneck in high-throughput distributed environments. VERDICT addresses this limitation by achieving a practical balance between privacy and efficiency through bucket-based indexing, level-based ECMH

hashing, and deterministic skip DAG structures. This design enables efficient verification of lineage integrity in milliseconds—two to three orders of magnitude faster than ZK-based approaches—while maintaining cryptographic guarantees of data integrity and authenticity.

Positioning Relative to Existing Work. Existing provenance systems typically excel in individual dimensions but lack VERDICT's integrated approach. LineageChain [22] and ForensiBlock [1] provide cryptographic proofs but are blockchain-integrated, suffering from consensus overhead. SPADE and PASS offer off-chain operation but lack formal security analysis or efficient cryptographic verification. The key innovation of VERDICT is its combination of three elements absent in prior work: (1) a dual-DAG model that unifies data and event lineage, enabling both artifact-centric and process-centric queries; (2) off-chain operation achieving millisecond verification latency while maintaining cryptographic integrity; and (3) formal security analysis under the Dolev-Yao model with explicit proofs of data integrity, authenticity, lineage integrity, and privacy. This integration–dual DAGs + off-chain efficiency + rigorous security–enables VERDICT to serve as a practical notary service for decentralized systems where traditional centralized provenance solutions are infeasible.

3 SYSTEM DESIGN

The design of VERDICT is guided by the need for a lineage tracking architecture that balances cryptographic verifiability, modularity, and privacy. Our architecture integrates a DAG-based lineage model with specialized cryptographic primitives–deterministic skip DAGs, bucket-based Merkle trees, and decentralized identifiers (DIDs)–to provide flexible, fine-grained traceability across complex distributed workflows. In this section, we present the system architecture, its core components, and the operational flow that enables cryptographically authenticated data lineage.

3.1 System Architecture and Overview

The system architecture consists of modular and decentralized components that collectively enable secure, authenticated, and verifiable lineage tracking across data workflows. It integrates identity management, coordinated execution, structured event logging, cryptographic proof generation, and multi-layered storage to ensure each transformation is correctly attributed, recorded, and verifiable. The architecture is illustrated in Figure 1, and its core building blocks are described as follows:

- Gateway: The Gateway manages all external communications with the DataHub, including data submissions, retrievals, and verification requests. It ensures secure and structured access to internal services.
- Authentication and Authorization Service: All requests pass through this service, which enforces access control via DID-based authentication. It verifies digital signatures and authorizes operations based on actor permissions, ensuring non-repudiation and decentralized identity binding.
- Datahub Microservices Orchestrator: This component coordinates workflow execution, invoking microservices in sequence, aggregating transformation context, and instructing the EventLogger to record events. It interfaces with storage for input retrieval and output persistence, maintaining consistency across the pipeline.

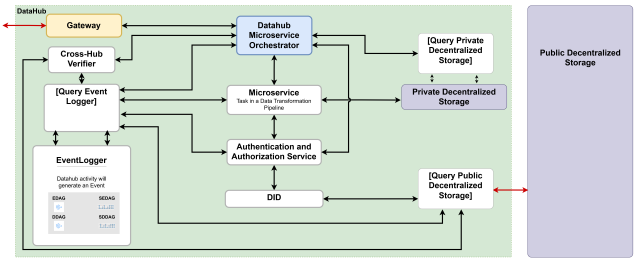

Figure 1: System architecture illustrating DataHub components and their interactions with private and public decentralized storage layers.

- Microservices: are stateless processing units that perform data transformations. Each is associated with a DID and interacts with the orchestrator and private storage. They report output and metadata back to the orchestrator, enabling provenance capture without managing lineage themselves.
- EventLogger: The EventLogger records lineage by updating the Data DAG (DDAG) and Event DAG (EDAG) with new nodes and edges. It also maintains skip-enhanced versions (SDDAG, SEDAG) for efficient verification. To support cryptographic proofs, it applies bucket indexing and constructs Merkle trees over grouped elements, pushing secure digests to public storage and storing full records privately.
- Cross-Hub Verifier: This component verifies lineage integrity by comparing private records to public commitments using skip DAG traversal and Merkle inclusion proofs. It supports inter-hub validation and acts as an internal audit mechanism.
- Private Decentralized Storage: This layer stores encrypted data artifacts and detailed lineage records. It supports internal audits and queries under access control, maintaining privacy and integrity across distributed storage.
- Public Decentralized Storage: This external ledger stores cryptographic commitments (e.g., Merkle roots, skip DAG digests) for tamper-evident anchoring. It enables third-party verifiability without exposing sensitive data.

The workflow of data lineage within the VERDICT Framework encompasses a sequence of key cryptographic and structural components that work in concert to ensure the integrity, traceability, and verifiability of data transformations. The process begins with the representation of data dependencies as a Data DAG, progresses through level-based hashing and skip DAG construction for efficient verification, and culminates in bucket indexing and Merkle tree verification for cryptographic guarantees of data and event existence. This workflow is equally applicable to the Event DAG, as the structural and cryptographic mechanisms employed for securing and verifying lineage apply consistently to both DAG types.

The core components that drive this workflow include the Data DAG ($G_D = (V_D, E_D)$) for representing artifact dependencies, Level-Based Hashing ($H(L_i)$) for efficient multi-level verification, the Skip DAG ($G_{SD} = (V_{SD}, E_{SD})$) for logarithmic-time traversal, Bucket Indexing ($\beta(v) = H(id_v) \bmod N$) for privacy-preserving element grouping, and Merkle Tree Verification ($MT(B_i)$) for cryptographic membership proofs. Each of these components plays a crucial role in the overall lineage verification system, together ensuring that every data transformation is not only recorded but also cryptographically verifiable.

The system operates through two complementary interaction flows: event logging and verification. During event logging, microservices report events to the orchestrator, which authenticates the request and invokes the EventLogger. The logger updates both the Data DAG and Event DAG with new nodes and edges, maintains their skip-enhanced versions for efficient verification, applies bucket indexing and constructs Merkle trees over grouped elements, stores full records privately for internal audits, and pushes secure cryptographic digests to public decentralized storage. During query and verification, users or external systems query the orchestrator, which retrieves data from private storage and public proofs from decentralized storage, followed by skip DAG traversal and Merkle path validation. This two-tier approach ensures the consistency, integrity, and verifiability of lineage across the system while maintaining privacy of sensitive data.

This architecture ensures that transformations are authenticated, lineage is provable, and data remains verifiable and secure across decentralized environments.

Our secure lineage verification system is designed to address the challenges of tracking and verifying data transformations in distributed environments while maintaining privacy and scalability.

The system operates under the following assumptions and design goals:

- **Distributed Environment:** The system operates in a distributed setting where multiple participants may contribute to data transformations.
- **Privacy Preservation:** The system must verify lineage without requiring disclosure of sensitive data.
- **Scalability:** Verification operations must remain efficient regardless of the total size of the lineage history.
- **Security:** The system must provide cryptographic guarantees against tampering and forgery.
- **Minimal Coordination:** The system should minimize the need for synchronization between participants.

We assume the existence of a secure identity and key management infrastructure based on decentralized identifiers (DIDs). These DIDs enable the binding of identities to cryptographic keys in a verifiable and decentralized manner. We also assume that cryptographic primitives (e.g., hash functions, digital signatures) are computationally secure, and that at least one honest participant maintains the correct state of the system.

3.2 Data DAG

The Data DAG is a fundamental component of our system, representing the relationships between data artifacts and their transformations. Formally, we define the Data DAG as $G_D = (V_D, E_D)$, where:

- V_D is the set of vertices representing data artifacts
- E_D is the set of directed edges representing dependencies between data artifacts

Each vertex $v \in V_D$ corresponds to an immutable data artifact, which could be a file, a database record, a model parameter, or any other discrete unit of data. Each vertex is associated with a unique identifier and a cryptographic hash of its content.

Each edge $e = (u, v) \in E_D$ represents a dependency relationship where data artifact v depends on data artifact u. This dependency typically arises from a transformation operation (lineage event) that takes u as input and produces v as output.

3.2.1 Node and Edge Semantics. Nodes in the Data DAG have the following properties:

- **Immutability:** Once created, a data artifact's content cannot be modified.
- **Unique Identification:** Each data artifact has a unique identifier.
- **Content Hashing:** Each data artifact is associated with a cryptographic hash of its content.
- **Metadata:** Optional metadata describing the artifact's characteristics.

Edges in the Data DAG have the following semantics:

- **Transformation Dependency:** An edge (u, v) indicates that data artifact v was derived from data artifact u through some transformation.
- **Lineage Event Reference:** Each edge is associated with a reference to the lineage event that created the dependency.
- **Temporal Ordering:** Edges implicitly capture the temporal ordering of data artifacts.

3.2.2 Formal Definition. More formally, we can define the Data DAG as follows:

Each node $v_i \in V_D$ is a tuple $(id_i, H(d_i), meta_i)$ where:

- id_i is a unique identifier
- $H(d_i)$ is the cryptographic hash of the data content
- $meta_i$ is optional metadata

Each edge $(v_i, v_j) \in E_D$ is a tuple (v_i, v_j, e_k) where:

- v_i is the source node (input data)
- v_j is the target node (output data)
- e_k is a reference to the lineage event that created the dependency

The Data DAG satisfies the following properties:

- **Acyclicity:** There are no cycles in the graph, ensuring that no data artifact can depend on itself directly or indirectly.
- **Genesis Nodes:** There exist one or more genesis nodes (nodes with no incoming edges) that represent the original data artifacts.
- **Connectivity:** For every non-genesis node, there exists at least one path from a genesis node to that node.

3.3 Event DAG

The Event DAG complements the Data DAG by focusing on the transformation events themselves rather than the data artifacts. It captures the causal relationships between transformation events, providing a process-centric view of the lineage. Formally, we define the Event DAG as $G_E = (V_E, E_E)$, where:

- V_E is the set of vertices representing lineage events (transformations)
- E_E is the set of directed edges representing causal dependencies between events

Each vertex $v \in V_E$ corresponds to a lineage event, which represents a transformation operation that takes one or more data artifacts as input and produces one or more data artifacts as output. Each event is associated with a unique identifier and a cryptographic hash of its description.

Each edge $e = (u, v) \in E_E$ represents a causal dependency where event v depends on event u. This dependency arises when event v uses as input a data artifact that was produced as output by event u.

3.3.1 Node and Edge Semantics. Nodes in the Event DAG have the following properties:

- **Transformation Description:** Each event node contains a description of the transformation operation.
- **Input References:** References to the input data artifacts consumed by the event.
- **Output References:** References to the output data artifacts produced by the event.
- **Execution Metadata:** Optional metadata about the execution environment, parameters, and timing.

Edges in the Event DAG have the following semantics:

- **Causal Dependency:** An edge (u, v) indicates that event v causally depends on event u.
- **Data Flow Implication:** The existence of an edge implies that some output of event u was used as input to event v.
- **Temporal Ordering:** Edges capture the happens-before relationship between events.

3.3.2 Formal Definition. More formally, we can define the Event DAG as follows:

Each node $e_i \in V_E$ is a tuple $(id_i, desc_i, in_i, out_i, meta_i)$ where:

- id_i is a unique identifier
- $desc_i$ is a description of the transformation
- in_i is the set of input data artifact references
- out_i is the set of output data artifact references
- $meta_i$ is optional metadata

Each edge $(e_i, e_j) \in E_E$ exists if and only if there exists some data artifact d such that:

- $d \in out_i$ (d is an output of event e_i)
- $d \in in_j$ (d is an input to event e_j)

The Event DAG satisfies the following properties:

- **Acyclicity:** There are no cycles in the graph, ensuring that no event can causally depend on itself.
- **Genesis Events:** There exist one or more genesis events (events with no incoming edges) that represent the initial transformations.
- **Consistency with Data DAG:** The Event DAG is consistent with the Data DAG, meaning that if there is an edge from event e_i to event e_j in the Event DAG, then there must be at least one data artifact that is both an output of e_i and an input to e_j in the Data DAG.

3.4 Relationship Between Data and Event DAGs

The Data Event DAGs are complementary representations of the same underlying lineage information, offering different perspectives and supporting different types of queries and verifications. The relationship between these two DAGs is bidirectional and can be formalized as follows:

- **Data-to-Event Mapping:** Each edge (d_i, d_j) in the Data DAG is associated with exactly one event e_k in the Event DAG, where d_i is an input to e_k and d_j is an output of e_k.
- **Event-to-Data Mapping:** Each event e_k in the Event DAG is associated with a set of input data artifacts $\{d_{i1}, d_{i2}, ..., d_{in}\}$ and a set of output data artifacts $\{d_{j1}, d_{j2}, ..., d_{jm}\}$ in the Data DAG.
- **Causal Consistency:** If there is an edge (e_i, e_j) in the Event DAG, then there must exist data artifacts d_i and d_j in the Data DAG such that d_i is an output of e_i, d_j is an output of e_j, and there is a path from d_i to d_j in the Data DAG.

This dual representation enables our system to support both artifact-centric queries (e.g., "What is the complete lineage of this data artifact?") and process-centric queries (e.g., "What is the causal relationship between these transformation events?"), providing a comprehensive framework for lineage verification.

4 LEVEL-BASED DAG COMPRESSION

4.1 DAG Decomposition

An essential step in our secure lineage verification system is the level-based decomposition of both the Data DAG and the Event DAG. This process organizes the nodes of each DAG into levels according to their distance from genesis nodes, producing a structured representation that enables efficient verification procedures.

4.1.1 Level Definition. For a DAG $G = (V, E)$, we define the level of a node $v \in V$, denoted by $L(v)$, as follows:

- If v is a genesis node (has no incoming edges), then $L(v) = 1$
- Otherwise, $L(v) = 1 + \max\{L(u) | (u, v) \in E\}$

In other words, the level of a node is one more than the maximum level of any of its predecessors. This definition ensures that all edges in the DAG connect nodes either at the same level or at different levels, with the direction always pointing from a lower or equal level to a higher level.

4.1.2 Level Partitioning. Based on the level definition, we partition the nodes of the DAG into disjoint sets $L_1, L_2, ..., L_n$, where:

$$L_i = \{v \in V | L(v) = i\} \tag{1}$$

This partitioning has several important properties:

- **Completeness:** Every node in the DAG belongs to exactly one level.
- **Ordering:** If there is an edge $(u, v) \in E$, then $L(u) \leq L(v)$.

- **Genesis Identification:** L_1 consists precisely of all genesis nodes.
- **Maximum Level:** The maximum level n is bounded by the length of the longest path in the DAG.

4.1.3 Level-Based Partitioning Algorithm. Algorithm 1 presents the pseudocode for the level-based partitioning of a DAG.

Algorithm 1 OptimizedLevelPartitioning(G)

Require: DAG $G = (V, E)$
Ensure: Level sets $L_1, L_2, ..., L_n$
Initialize empty sets $L_1, L_2, \ldots$
Initialize level map `level[`v`]` for all $v \in V$
Initialize `in_degree[`v`]` $\leftarrow$ number of incoming edges for each $v \in V$
Initialize `in_processed[`v`]` $\leftarrow 0$ for all $v \in V$
Initialize queue Q
for each $v \in V$ **do**
  if `in_degree[`v`]` $= 0$ **then**
    `level[`v`]` $\leftarrow 1$
    Add v to L_1
    Enqueue v into Q
  end if
end for
while Q is not empty **do**
  $u \leftarrow$ Dequeue from Q
  for each edge $(u, v) \in E$ **do**
    `in_processed[`v`]` $\leftarrow$ `in_processed[`v`]` $+1$
    if `level[`v`]` $<$ `level[`u`]` $+1$ **then**
      `level[`v`]` $\leftarrow$ `level[`u`]` $+1$
      Remove v from its current level set (if any)
      Add v to $L_{\texttt{level[}v\texttt{]}}$
    end if
    if `in_processed[`v`]` $=$ `in_degree[`v`]` **then**
      Enqueue v into Q
    end if
  end for
end while
return $L_1, L_2, ..., L_n$

This algorithm has a time complexity of $O(|V| + |E|)$, making it efficient even for large DAGs.

4.2 Hashed Level DAG Construction

Once the DAG has been decomposed into levels, we construct a hashed representation that enables efficient verification. This process involves computing a hash value for each level by concatenating the hashes of all nodes at that level.

4.2.1 Node Hashing. For each node v in the DAG, we compute a cryptographic hash $H(v)$ as follows:

- For the Data DAG: $H(v) = \text{Hash}(\text{content} \,||\, \text{metadata})$
- For the Event DAG: $H(v) = \text{Hash}(\text{description} \,||\, \text{inputs} \,||\, \text{outputs} \,||\, \text{metadata})$

where Hash is a secure cryptographic hash function, and $||$ denotes concatenation.

4.2.2 Level Hashing via Multiset Hash Functions. A key requirement for level hashing in a distributed setting is that the hash must be *order-independent*: nodes may be inserted concurrently and there is no globally agreed canonical ordering without coordination overhead. Furthermore, the hash must support *incremental updates* so that adding a new node costs $O(1)$ rather than requiring recomputation over the entire level.

These two requirements are jointly satisfied by *multiset hash functions* [5]. A multiset hash function $\mathcal{H}$ maps a multiset S to a digest such that:

$$\mathcal{H}(S \cup \{v\}) = \mathcal{H}(S) \oplus \mathcal{H}(\{v\}) \tag{2}$$

where $\oplus$ is a commutative and associative group operation. Crucially, $\mathcal{H}(S)$ is independent of the order in which elements of S are processed.

Concrete instantiation. We instantiate $\mathcal{H}$ using an *Elliptic Curve Multiset Hash (ECMH)*, a well-studied construction [24] with security reduction to the Discrete Logarithm Problem. Let $\mathbb{G}$ be an elliptic curve group of prime order q (e.g., NIST P-256 or Curve25519), with $\mathcal{O}$ denoting the identity element. Define a hash-to-point function $\phi : \{0,1\}^* \to \mathbb{G}$ (e.g., following the hash-to-curve specification of RFC 9380). For each node $v \in V$, its *point hash* is:

$$\hat{h}(v) = \phi(H(v)) \tag{3}$$

and the level hash is the sum of all point hashes in the level:

$$H(L_i) = \bigoplus_{v \in L_i} \hat{h}(v) \tag{4}$$

where $\bigoplus$ denotes elliptic curve point addition and $H(L_i) = \mathcal{O}$ for an empty level.

This formulation satisfies the following properties:

- **Order Independence:** $H(L_i)$ is the same regardless of insertion order, requiring no canonical sorting of nodes. This directly eliminates the coordination cost previously imposed on distributed participants.
- **Collision Resistance:** Finding two distinct sets $L_i \neq L_i'$ with $H(L_i) = H(L_i')$ requires computing a non-trivial discrete logarithm relation in $\mathbb{G}$, which is computationally infeasible under standard assumptions.
- $O(1)$ **Incremental Update:** When a new node v_{new} is added to level L_i, the updated level hash is:

$$H'(L_i) = H(L_i) \oplus \hat{h}(v_{\text{new}}) \tag{5}$$

  requiring only a single point addition and a hash-to-curve evaluation, independent of $|L_i|$.
- **Sensitivity:** Any change to any node $v \in L_i$ results in $\hat{h}(v) \neq \hat{h}(v')$, and consequently $H(L_i) \neq H(L_i')$, by the collision resistance of ϕ.

Remark 1. *Although an inverse operation $H'(L_i) = H(L_i) \ominus \hat{h}(v)$ is algebraically well-defined (EC point subtraction), it is not exercised in our system since levels grow monotonically and node deletion is not supported.*

4.2.3 Hashed Level DAG Construction Algorithm. Algorithm 2 presents the pseudocode for constructing the hashed level DAG. Since the level hash is now a multiset hash, no sorting step is required: nodes are processed in any order and their point hashes are accumulated via elliptic curve point addition.

Algorithm 2 HashedLevelDAG($G, L_1, L_2, \ldots, L_n$)

Require: DAG $G = (V, E)$, level sets $L_1, L_2, \ldots, L_n$
Ensure: Hashed Level DAG $HL = \{H(L_1), H(L_2), \ldots, H(L_n)\}$
Initialize empty list HL
for $i = 1$ **to** n **do**
  $H(L_i) \leftarrow \mathcal{O}$ {Identity element of $\mathbb{G}$}
  for each $v \in L_i$ **do**
    if G is Data DAG **then**
      $H(v) \leftarrow \text{Hash}(\text{content}(v) \parallel \text{metadata}(v))$
    else
      {G is Event DAG}
      $H(v) \leftarrow \text{Hash}(\text{description}(v) \parallel \text{inputs}(v) \parallel \text{outputs}(v) \parallel \text{metadata}(v))$
    end if
    $H(L_i) \leftarrow H(L_i) \oplus \phi(H(v))$ {EC point addition; ϕ is hash-to-point}
  end for
  Append $H(L_i)$ to HL
end for
return HL

This algorithm has a time complexity of $O(|V|)$, an improvement over the $O(|V| \log |V|)$ cost of a sort-then-concatenate approach, as no canonical ordering is required. The ECMH update upon subsequent insertion of a node v into level L_i costs $O(1)$: $H'(L_i) \leftarrow H(L_i) \oplus \phi(H(v))$.

4.3 Data and Event DAGs Hashing

We apply the level-based hashing approach to both the Data DAG and the Event DAG, resulting in two hashed level DAGs:

- Hashed Data Level DAG: $HD = \{H(LD_1), H(LD_2), \ldots, H(LD_n)\}$
- Hashed Event Level DAG: $HE = \{H(LE_1), H(LE_2), \ldots, H(LE_m)\}$

where,

$LD_1, LD_2, \ldots, LD_n$ are the levels of the Data DAG, and
$LE_1, LE_2, \ldots, LE_m$ are the levels of the Event DAG.

These hashed level DAGs serve as compact representations of the original DAGs, enabling efficient verification of their integrity. The homomorphic properties of the hash function allow for efficient updates to these representations as the underlying DAGs evolve.

4.3.1 Signing Level Hashes. To provide non-repudiation and authentication, we sign the level hashes using a digital signature scheme. For each level L_i, we compute a signature:

$$\text{Sig}(L_i) = \text{Sign}(H(L_i), SK) \tag{6}$$

where Sign is a digital signature algorithm and SK is the signer's private key.

These signatures can be verified using the corresponding public key, providing cryptographic assurance of the level's integrity and authenticity.

4.3.2 Level Hash Concatenation. To capture the cumulative state of the DAG up to a certain level, we define the concatenated level hash $CH(L_i)$ as:

$$CH(L_i) = \text{Hash}(H(L_1)||H(L_2)||...||H(L_i)) \tag{7}$$

This concatenated hash represents the entire DAG state up to level i, enabling efficient verification of the DAG's evolution over time.

The level-based DAG compression technique forms the foundation for our skip DAG construction and verification processes, which we describe in the following sections.

5 SKIP DAG CONSTRUCTION AND VERIFICATION

5.1 Skip Data DAG

Building upon the level-based DAG compression technique, we introduce the Skip Data DAG, a structure that enables efficient traversal and verification of the Data DAG. The Skip Data DAG is inspired by skip lists, a data structure that allows for fast search within an ordered sequence by maintaining multiple layers of linked lists with progressively fewer elements.

5.1.1 Construction Methodology. The Skip Data DAG is constructed from the Hashed Data Level DAG by selectively creating connections between levels, allowing certain levels to be "skipped" during traversal. We adopt a deterministic binary skip construction: from each level L_i, we add a directed edge to L_{i+2^k} for every integer $k \geq 0$ such that $i + 2^k \leq n$. This ensures that the traversal path from L_1 to any target level L_j is unique and reproducible without coordination.

Formally, we define the Skip Data DAG as $GS_D = (VS_D, ES_D)$, where:

- VS_D is the set of vertices, each corresponding to a level in the Hashed Data Level DAG
- ES_D is the set of directed edges representing skip connections between levels

Each vertex $v \in VS_D$ corresponds to a level L_i in the Hashed Data Level DAG and is associated with the level hash $H(L_i)$. The edges in ES_D are constructed *deterministically*, ensuring that the traversal path from L_1 to any target level L_j is unique and reproducible without coordination.

5.1.2 Deterministic Binary Skip Structure. We adopt a *deterministic binary skip* construction: from each level L_i, we add a directed edge to L_{i+2^k} for every integer $k \geq 0$ such that $i + 2^k \leq n$. Concretely, the edge set is:

$$ES_D = \left\{(L_i,\ L_{i+2^k}) \mid 1 \leq i \leq n,\ k \geq 0,\ i + 2^k \leq n\right\}. \tag{8}$$

The case $k = 0$ yields the base edge (L_i, L_{i+1}), so consecutive connectivity is a special case of the general formula. The number of outgoing edges from L_i is $\lfloor \log_2(n - i) \rfloor + 1 = O(\log n)$, and the total edge count is $|ES_D| = O(n \log n)$.

Unique traversal path. For any target level j, the path from L_1 to L_j is uniquely determined by the *greedy binary descent*: at each current position L_i, take the edge (L_i, L_{i+2^k}) where 2^k is the largest power of two satisfying $i + 2^k \leq j$. This greedy choice corresponds to subtracting the highest set bit of $(j - i)$ at each step and always

yields a path of length exactly $\lfloor \log_2 j \rfloor + 1 = O(\log n)$. Algorithm 3 formalises this construction.

Algorithm 3 BinaryDescentPath(j, n)

Require: Target level index $j \leq n$
Ensure: Ordered path $\pi = [L_1, \ldots, L_j]$ of length $O(\log n)$
$i \leftarrow 1$; $\pi \leftarrow [L_1]$
while $i < j$ **do**
  $step \leftarrow 2^{\lfloor \log_2(j-i) \rfloor}$ {Largest power of 2 not exceeding $j - i$}
  $i \leftarrow i + step$; Append L_i to π
end while
return π

Path hashing and signing. For a traversal path $\pi = [L_{i_1}, L_{i_2}, \ldots, L_{i_r}]$ from L_1 to L_j, the *path hash* commits to the cumulative DAG state at each visited level:

$$PH(\pi) = \text{Hash}\big(CH(L_{i_1}) \parallel CH(L_{i_2}) \parallel \cdots \parallel CH(L_{i_r})\big). \tag{9}$$

When element x is inserted into level L_j, this path hash is signed and stored in x's metadata:

$$SP(x) = \text{Sign}(PH(\pi(1, j)),\ SK). \tag{10}$$

Since the path $\pi(1, j)$ is deterministically computed by Algorithm 3, any verifier can reconstruct it independently, re-derive $PH(\pi)$, and check $SP(x)$ against the signer's public key – with no additional coordination and no per-edge signature storage.

Algorithm 4 BuildSkipDataDAG(HD, SK)

Require: Hashed Data Level DAG $HD = \{H(LD_1), \ldots, H(LD_n)\}$, signing key SK
Ensure: Skip Data DAG $GS_D = (VS_D, ES_D)$
$VS_D \leftarrow \{LD_1, LD_2, \ldots, LD_n\}$; $ES_D \leftarrow \emptyset$
for $i = 1$ **to** n **do**
  $k \leftarrow 0$
  while $i + 2^k \leq n$ **do**
    Add edge $(LD_i,\ LD_{i+2^k})$ to ES_D
    $k \leftarrow k + 1$
  end while
end for
return GS_D

5.1.3 Skip Data DAG Construction Algorithm. The construction adds $O(n \log n)$ edges in worst-case time $O(n \log n)$ (no random sampling, no signing at construction time). Signing is deferred to element insertion (Algorithm 6), ensuring that each signature binds a specific element to a specific DAG state rather than signing structure edges in the abstract.

5.2 Skip Event DAG

The Skip Event DAG is constructed similarly to the Skip Data DAG but operates on the Hashed Event Level DAG. It provides efficient traversal and verification capabilities for the Event DAG, enabling quick verification of event causality.

5.2.1 Construction Methodology. The Skip Event DAG is defined as $GS_E = (VS_E, ES_E)$, where:

- VS_E is the set of vertices, each corresponding to a level in the Hashed Event Level DAG
- ES_E is the set of directed edges representing skip connections between levels

The construction methodology is identical to the Skip Data DAG: deterministic binary skip edges (LE_i, LE_{i+2^k}) are added for all valid i and k, giving the same $O(n \log n)$ edge set and $O(\log n)$ traversal guarantee.

5.2.2 Relationship with Skip Data DAG. While the Skip Data DAG and Skip Event DAG are constructed independently, they are semantically related through the underlying relationship between the Data DAG and Event DAG. Specifically:

- If data artifacts d_1 and d_2 are connected in the Data DAG through a lineage event e, then the level of e in the Event DAG is related to the levels of d_1 and d_2 in the Data DAG.
- The skip connections in the Skip Event DAG reflect the causal relationships between events, which indirectly correspond to data dependencies in the Skip Data DAG.

This relationship enables cross-verification between the two skip structures, enhancing the overall security of the system.

5.2.3 Path Representation and Signing. As with the Skip Data DAG, each skip connection in the Skip Event DAG is associated with a path hash and a signed path:

$$PH(LE_i, LE_j) = \text{Hash}(CH(LE_i) || CH(LE_j)) \tag{11}$$

$$SP(LE_i, LE_j) = \text{Sign}(PH(LE_i, LE_j), SK) \tag{12}$$

These signed paths enable efficient verification of event causality.

Algorithm 5 BuildSkipEventDAG(HE, SK)

Require: Hashed Event Level DAG $HE = \{H(LE_1), \ldots, H(LE_m)\}$, signing key SK
Ensure: Skip Event DAG $GS_E = (VS_E, ES_E)$
$VS_E \leftarrow \{LE_1, LE_2, \ldots, LE_m\}$; $ES_E \leftarrow \emptyset$
for $i = 1$ **to** m **do**
  $k \leftarrow 0$
  while $i + 2^k \leq m$ **do**
    Add edge $(LE_i,\ LE_{i+2^k})$ to ES_E
    $k \leftarrow k + 1$
  end while
end for
return GS_E

5.2.4 Skip Event DAG Construction Algorithm. This algorithm runs in worst-case time $O(m \log m)$, where m is the number of levels in the Event Level DAG. The structure is symmetric to the Skip Data DAG; both share Algorithm 3 for path reconstruction and Algorithm 6 for element insertion and signing.

5.2.5 New Element Insertion into the Skip DAG. When a new element x is added to the system, its target level $j = L(x)$ is already determined by the level-partitioning algorithm (Algorithm 1). Insertion into the Skip DAG then proceeds in three steps: (1) update the ECMH level hash of L_j; (2) compute the unique binary-descent path $\pi(1, j)$ using Algorithm 3; and (3) sign the path hash and store the signature in x's metadata for future verification.

If L_j is a newly created level (no prior elements), the skip edges from L_j are appended to GS following the same binary structure.

The level hash update uses the ECMH additive property:

$$H'(L_j) = H(L_j) \oplus \phi(H(x)) \tag{13}$$

where ϕ is the hash-to-point function and $\oplus$ is EC point addition, identical to the update rule in Algorithm 2.

Algorithm 6 summarises the full procedure.

Algorithm 6 InsertIntoSkipDAG(x, j, GS, SK)

Require: Element x, target level index $j = L(x)$, Skip DAG GS, signing key SK
Ensure: Updated GS; $x.metadata$ populated with level j and signed path SP

if L_j does not exist in GS **then**
  Create level node L_j; set $H(L_j) \leftarrow \mathcal{O}$
  Extend GS: for $k = 0$ while $j + 2^k \le n$: add edge (L_j, L_{j+2^k})
  Extend GS: for each existing L_i with $i < j$ and $j - i$ a power of 2: add edge (L_i, L_j)
end if
{Step 1: update ECMH level hash}
$H(x) \leftarrow \text{ComputeNodeHash}(x)$
$H(L_j) \leftarrow H(L_j) \oplus \phi(H(x))$
{Step 2: compute the unique binary-descent path from L_1 to L_j}
$\pi \leftarrow \text{BinaryDescentPath}(j,\ n)$ {Algorithm 3}
{Step 3: sign the path and store in metadata}
$PH \leftarrow \text{Hash}\big(CH(L_{i_1}) \parallel CH(L_{i_2}) \parallel \cdots \parallel CH(L_j)\big)$ where $\pi = [L_{i_1}, \ldots, L_j]$
$SP \leftarrow \text{Sign}(PH,\ SK)$
$x.metadata \leftarrow \{\ level : j,\ signed_path : SP\ \}$
return Updated GS

5.3 Verification Process

The Skip DAG structures enable efficient verification of data and event levels existence within their respective DAGs. The verification process involves traversing the Skip DAG to locate the level containing the target element and verifying the path to that level.

5.3.1 Path Reconstruction via Forward Traversal. At insertion time (Algorithm 6), the inserter stores two values in x's metadata: (i) the target level index j (i.e., $x.metadata.level = j$), and (ii) a signed path $x.metadata.signed_path$ corresponding to the sequence of skip edges traversed from L_1 to L_j.

During verification, the verifier does not *search* for x's level–it is read directly from metadata. The verifier's task is to *reconstruct* the same traversal path and confirm that every edge on that path carries a valid signature, thereby authenticating the DAG state at the time of insertion.

Path reconstruction proceeds *forward* from L_1 toward L_j: at each position L_i, the algorithm selects the longest available skip edge $(L_i, L_l) \in Edges(GS)$ satisfying $l \le j$. This greedy forward strategy mirrors standard skip-list search and runs in $O(\log n)$ expected time when skip edges are distributed geometrically (as in the construction), and in $O(\log n)$ worst-case time under the deterministic binary construction.

Remark 2. *The previous top-down approach (iterating from* `MaxLevel` *downward with an inner scan) inadvertently introduced an $O(n)$ inner loop that negated the logarithmic benefit of skip connections. The forward traversal from L_1 corrects this: the number of hops equals the path length stored at insertion, which is $O(\log n)$.*

Algorithm 7 VerifyExistence(x, GS, PK)

Require: Element x with $x.metadata.level = j$ and $x.metadata.signed_path = SP$; Skip DAG GS (edge structure only); public key PK
Ensure: **true** iff x exists in L_j and the binary-descent path to L_j is authenticated

$j \leftarrow x.metadata.level$
$i \leftarrow 1$; $path \leftarrow [L_1]$
{Phase 1: reconstruct the insertion path forward from L_1 to L_j}
while $i < j$ **do**
  $best \leftarrow i + 1$ {Fallback: unit step}
  for each edge $(L_i,\ L_l) \in Edges(GS)$ with $l \le j$ **do**
    if $l > best$ **then**
      $best \leftarrow l$
    end if
  end for
  $i \leftarrow best$; Append L_i to $path$
end while
if $i \ne j$ **then**
  return false {Could not reach L_j; structural mismatch}
end if
{Phase 2: recompute path hash and verify the single stored signature}
$PH \leftarrow \text{Hash}\big(CH(L_{i_1}) \parallel CH(L_{i_2}) \parallel \cdots \parallel CH(L_j)\big)$ where $path = [L_{i_1}, \ldots, L_j]$
$SP \leftarrow x.metadata.signed_path$
if not Verify(PH, SP, PK) **then**
  return false {Path hash mismatch: DAG state tampered or wrong level claimed}
end if
{Phase 3: confirm $x \in L_j$ via Merkle membership proof (Section 5)}
return VerifyMerkleMembership(x, L_j)

5.3.2 Verification Algorithm.

Correctness. The path $path$ reconstructed in Phase 1 is identical to the path recorded at insertion time because both use the same greedy forward strategy over the same skip edge set. Hence HashPath($path$) = SP holds iff no level on the path has been modified since insertion.

Complexity. Phase 1 visits $O(\log n)$ levels (the path length), and at each level inspects the outgoing skip edges of L_i, which number at most $O(\log n)$ under the binary construction or $O(\log n)$ in expectation under the probabilistic construction. Thus the overall complexity of Algorithm 7 is $O(\log^2 n)$ in general, and $O(\log n)$ under the deterministic binary construction where the longest valid edge is identified in $O(1)$ by direct index arithmetic. Phase 2 performs $O(\log n)$ signature verifications. Phase 3 is $O(\log k)$ for k buckets.

The Skip DAG construction and verification processes form the first verification tier of VERDICT. They authenticate the level assignment of any element via signed skip-path traversal, providing an $O(\log n)$ path proof whose length is independent of the total number of nodes in the underlying DAG.

In the next section, we describe how this structure is complemented by bucket indexing and Merkle tree verification to provide comprehensive and tamper-evident lineage verification guarantees.

6 BUCKET INDEXING AND MERKLE TREE VERIFICATION

Having verified the correct level L_i of the target element x through signed skip path authentication in the Skip DAG, we now focus on verifying the existence of x within that level. To preserve privacy and enable cryptographic verifiability, we decompose the level L_i into buckets using hash-based indexing and construct Merkle trees over these buckets. This allows us to generate and verify membership proofs for individual elements, based on a signed Merkle root, in a manner analogous to the signed path verification in the Skip DAG.

6.1 Bucket Assignment

While the Skip DAG structure enables efficient verification of the level containing a target element, it does not directly verify the presence of specific elements within that level. To address this, we introduce a bucket indexing approach that organizes elements within the previously identified level into buckets based on their hash values.

6.1.1 Hash-Based Bucket Indexing. For level L_i in either the Data DAG or Event DAG, we partition the elements into k buckets $B_1, B_2, \ldots, B_k$ using a hash-based assignment function:

$$\text{bucket_index}(x) = H(x) \bmod k \tag{14}$$

where H is a cryptographic hash function, and k is the number of buckets.

This assignment function exhibits the following properties:

- **Determinism:** The same element is always assigned to the same bucket.
- **Uniformity:** Elements are distributed approximately uniformly across buckets due to the pseudo-random nature of H.
- **Collision Tolerance:** While collisions (i.e., multiple elements mapping to the same bucket) may occur, they are securely handled within each bucket.

6.1.2 Element Comparison within Buckets. Once an element x is assigned to a bucket, we must verify its membership efficiently without exposing individual element content to the public.

We adopt a *hash-based membership* scheme: for each element y placed in bucket B_j, we store its cryptographic digest $H(y) = \text{Hash}(y)$ rather than the plaintext. Membership verification then reduces to a hash comparison:

$$x \in B_j \iff \exists H(y) \in B_j \text{ such that } H(x) = H(y). \tag{15}$$

This is sound because H is collision-resistant (Assumption 1): $H(x) = H(y)$ implies $x = y$ with overwhelming probability. Privacy is preserved because (i) the bucket contents $\{H(y)\}$ are stored in *private* storage and are accessible only under DID-based authentication; (ii) the only publicly committed value is the signed Merkle root SMR_i over the bucket hashes (Section 6.6.2), from which individual $H(y)$ values cannot be recovered by preimage resistance; and (iii) the bucket index $\beta(x) = H(x) \bmod k$ leaks only $\lfloor \log_2 k \rfloor$ bits of $H(x)$, which is bounded as analysed in Theorem 5.

6.1.3 Bucket Assignment Algorithm. Algorithm 8 presents the pseudocode for bucket assignment.

Algorithm 8 BucketAssignment(L, k)

Require: Level L with elements $\{x_1, x_2, \ldots, x_m\}$, number of buckets k
Ensure: Buckets $B_1, B_2, \ldots, B_k$ containing element digests
1: Initialize empty buckets $B_1, B_2, \ldots, B_k$
2: **for** each element x_i in L **do**
3: $j \leftarrow H(x_i) \bmod k$ {Bucket index}
4: Add $H(x_i)$ to bucket B_j {Store digest, not plaintext}
5: **end for**
6: **return** $B_1, B_2, \ldots, B_k$

The time complexity of this algorithm is $O(m)$, where m is the number of elements in the level.

6.2 Merkle Tree Construction

To enable efficient and privacy-preserving verification of bucket contents, we construct a Merkle tree over the buckets in each level.

6.2.1 Bucket-Based Merkle Tree. For each level L_i, we build a Merkle tree MT_i where:

- The leaf nodes correspond to buckets $B_1, B_2, \ldots, B_k$.
- Each leaf node contains the hash of all element digests in the corresponding bucket.
- Internal nodes hash the concatenation of their child node hashes.
- The root hash MR_i represents a compact commitment to all elements in level L_i.

Formally:

$$H(B_j) = \text{Hash}(H(x_1) \| H(x_2) \| \ldots \| H(x_n)), \quad x_1, \ldots, x_n \in B_j \tag{16}$$

$$H(\text{node}) = \text{Hash}(H(C_1) || H(C_2)) \tag{17}$$

where C_1 and C_2 are the child nodes of the current node.

6.2.2 Root Hash Calculation. The root hash MR_i is signed to yield a signed Merkle root:

$$SMR_i = \text{Sign}(MR_i, SK) \tag{18}$$

where SK is the private signing key. The signed root provides cryptographic assurance of the integrity and authenticity of all elements in the level.

6.2.3 Path Generation for Verification. For each element $x \in L_i$, we generate a Merkle proof path $MP(x)$ for existence verification. It includes:

- Bucket index $j = H(x) \bmod k$
- Element digest $H(x)$
- Sibling hashes along the path from leaf B_j to the Merkle root

6.2.4 Merkle Tree Construction Algorithm. Algorithm 9 provides the pseudocode for Merkle tree construction.

Algorithm 9 MerkleTreeConstruction($B_1, B_2, \ldots, B_k$)

Require: Buckets $B_1, B_2, \ldots, B_k$ containing element digests $\{H(x)\}$
Ensure: Merkle tree MT, root hash MR, signed root SMR
{Create leaf nodes}
for $j = 1$ to k **do**
  $H(B_j) \leftarrow \text{Hash}(H(x_1) \| H(x_2) \| \ldots \| H(x_n))$ $\{x_1, \ldots, x_n \in B_j\}$
  Create leaf node with hash $H(B_j)$
end for
{Build tree bottom-up}
while more than one node at current level **do**
  for each pair (N_1, N_2) **do**
    Create parent node with hash $\text{Hash}(H(N_1) || H(N_2))$
  end for
end while
$MR \leftarrow$ hash of root node
$SMR \leftarrow \text{Sign}(MR, SK)$
return MT, MR, SMR

The total time complexity is $O(m + k)$, accounting for hashing all elements and building the Merkle tree.

The combination of hash-based bucket indexing, encrypted comparison, and Merkle tree construction allows for the generation of a compact and verifiable proof of membership for each element. This structure ensures both privacy and cryptographic integrity, extending the guarantees offered by Skip DAGs down to the element level.

6.2.5 Bucket and Merkle Tree Updates for New Element. Once the appropriate level L_i is determined and updated, the system assigns the new element x to its corresponding bucket and updates the Merkle tree associated with that level.

This process includes:

- Computing the bucket index $j = H(x) \bmod k$.
- Encrypting the new element $E(x)$.
- Appending $E(x)$ to bucket B_j.
- Recomputing the bucket hash $H(B_j)$.
- Updating the leaf node in the Merkle tree and propagating hash updates up to the root.
- Recomputing the Merkle proof path associated with bucket B_j and updating it in the metadata for later verification.
- Recomputing and re-signing the new Merkle root.
- Storing the updated signed Merkle root SMR_i in the metadata of level L_i.

This ensures that any future proof of x's membership in the bucket can be verified against the authenticated Merkle root.

Algorithm 10 presents this update procedure.

Algorithm 10 UpdateMerkleTreeWithNewElement($x, B_1, \ldots, B_k, MT, SK$)

Require: New element x, buckets $B_1, \ldots, B_k$, Merkle tree MT, signing key SK
Ensure: Updated Merkle tree MT, root hash MR, and signed root SMR
$j \leftarrow H(x) \bmod k$ {Determine bucket}
Add $H(x)$ to bucket B_j {Store digest only}
$H(B_j) \leftarrow \text{Hash}(H(x_1) \| \ldots \| H(x_n))$ {Recompute bucket hash; $x_1, \ldots, x_n \in B_j$}
Update leaf node hash in MT for B_j
Propagate hash updates up to the Merkle root
$MR \leftarrow$ updated root hash of MT
$SMR \leftarrow \text{Sign}(MR, SK)$
Store SMR in level L_i's metadata
return Updated MT, MR, SMR

6.3 Verification Process

The bucket and Merkle tree structure enable efficient two-step verification of element existence.

6.3.1 Merkle Proof Path Construction. Given an element x claimed to be in level L_i, we construct $MP(x)$ as:

(1) Compute bucket index $j = H(x) \bmod k$
(2) Retrieve bucket B_j from private storage and check $H(x) \in B_j$ by direct hash comparison
(3) Collect sibling hashes from leaf B_j up to the Merkle root

6.3.2 Comparison with Signed Merkle Root. To verify that $x \in L_i$, the verifier:

(1) Computes bucket index $j = H(x) \bmod k$
(2) Checks whether $H(x) \in B_j$ (hash equality – no decryption required)
(3) Reconstructs Merkle root MR'_i from $MP(x)$ by hashing up the proof path
(4) Verifies $\text{VerifySig}(MR'_i, SMR_i, PK) = \textbf{true}$

If all checks succeed, the verifier confirms that x exists in level L_i and that its inclusion is authenticated by the signed Merkle root, consistent with the signed path verification used in the Skip DAG.

6.3.3 Security Analysis. The security of the bucket indexing and Merkle tree verification relies on:

- **Hash Function Collision Resistance:** $H(x) = H(y) \Rightarrow x = y$ with overwhelming probability (Assumption 1), ensuring membership checks are sound.

- **Hash Function Preimage Resistance:** The signed Merkle root SMR_i and bucket digests $\{H(x)\}$ in public view do not reveal element content (Assumption 1), establishing privacy.
- **Signature Unforgeability:** SMR_i cannot be forged without the signer's private key (Assumption 1), preventing root substitution attacks.

Under these assumptions, the verification process provides sound and privacy-preserving membership proofs without requiring any homomorphic encryption.

The combination of Skip DAG traversal for level location and bucket-based Merkle tree verification for element existence creates a comprehensive and efficient verification system for secure lineage in distributed environments.

6.4 Cross-DAG Consistency Verification

The Data DAG and Event DAG are maintained and verified independently in the protocols above. This creates a potential gap: an adversary could modify a data artifact x in the Data DAG while leaving the Event DAG untouched, or could inject a spurious event e into the Event DAG without a corresponding data transformation. Either manipulation would pass single-DAG verification but violate the semantic consistency of the lineage record.

Consistency invariant. Each edge (u, v) in the Data DAG is annotated with a reference to the transformation event e_{uv} that produced v from u. Consistency requires that: (i) e_{uv} exists in the Event DAG; and (ii) e_{uv} is assigned to the same level as v in their respective DAGs – formally, $\text{level}_{\mathcal{D}}(v) = \text{level}_{\mathcal{E}}(e_{uv})$.

Joint verification protocol. The following protocol extends VerifyExistence to enforce cross-DAG consistency for an element x at level j in the Data DAG:

(1) Run VerifyExistence(x, G_S^D) on the Data Skip DAG. Abort if it fails.
(2) Retrieve the event reference $e_x \leftarrow x$.metadata.event_ref.
(3) Run VerifyExistence(e_x, G_S^E) on the Event Skip DAG. Abort if it fails.
(4) Check level consistency: let $j_E \leftarrow e_x$.metadata.level; verify $j = j_E$. Abort if $j \neq j_E$.
(5) Run VerifyMerkleMembership$(e_x, L_{j_E}^E)$ on the Event Merkle tree. Abort if it fails.

If all five checks pass, the verifier confirms that x exists in the Data DAG at level j, that its producing event e_x exists in the Event DAG at the same level, and that both are cryptographically authenticated by their respective signed Merkle roots.

Security. Any modification to x or e_x is detected by the existing integrity theorems (Theorem 1 and Theorem 2). Level inconsistency – injecting x at level j while placing e_x at level $j' \neq j$ – is detected by step 4. An adversary who attempts to satisfy step 4 by also moving e_x to level j would need to forge a signed Merkle root for the Event DAG, which is infeasible under Assumption 1. Thus joint verification closes the cross-DAG forgery surface without introducing additional cryptographic assumptions.

Complexity. Joint verification costs two independent VerifyExistence calls plus one level equality check: $O(\log n_D) + O(\log n_E) + O(1)$, where n_D and n_E are the number of levels in the Data and Event DAGs respectively.

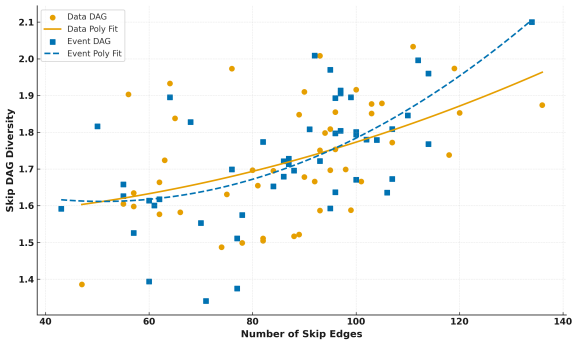

Figure 2: Skip DAG diversity vs. number of skip edges for both Data and Event DAGs. Higher diversity values indicate more efficient traversal paths.

7 PERFORMANCE EVALUATION

In this section, we present a comprehensive evaluation of VERDICT, focusing on the performance of our skip DAG and bucket Merkle tree structures. We first analyze skip DAG construction times, followed by insertion, search, and verification operations. We then evaluate scalability with respect to key structural parameters: the number of genesis nodes, edges per node, and DAG depth. We compare our approach with LineageChain [22] and ForensiBlock [1], the two state-of-the-art blockchain-based provenance systems closest to our work.

7.1 Experimental Setup

All experiments were conducted on a server equipped with an Intel Xeon E5-2680 v4 CPU @ 2.40GHz with 14 cores, 128GB RAM, running Ubuntu 20.04 LTS. We implemented VERDICT in Python. For comparison with LineageChain [22], we used the implementation provided by the authors, deployed on Hyperledger Fabric v1.4.

Our evaluation dataset consists of synthetic DAGs with varying characteristics to test different aspects of our system. We generated two types of DAGs: Data DAGs representing data transformation workflows and Event DAGs capturing event causality relationships. The number of nodes ranges from 1,000 to 20,000, with varying numbers of levels (25-60), skip edges (40-140), and genesis nodes (50-500). Each experiment was repeated 32 times, and we report the average results.

7.2 Skip DAG Diversity Analysis

Skip DAG diversity is a critical metric that measures the richness of skip connections in our DAG structure, directly impacting the efficiency of traversal and verification operations. Higher diversity indicates more efficient paths through the DAG, potentially reducing verification times.

Figure 2 shows the relationship between skip DAG diversity and the number of skip edges. We observe that diversity increases with the number of skip edges for both Data and Event DAGs, with a polynomial growth pattern. Event DAGs show slightly higher

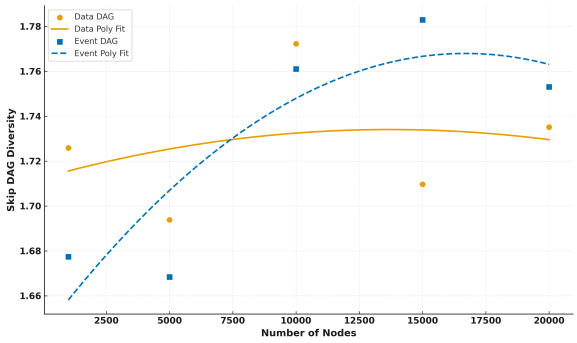

Figure 3: Skip DAG diversity vs. number of nodes. The diversity stabilizes after 10,000 nodes for Data DAGs but continues to increase for Event DAGs.

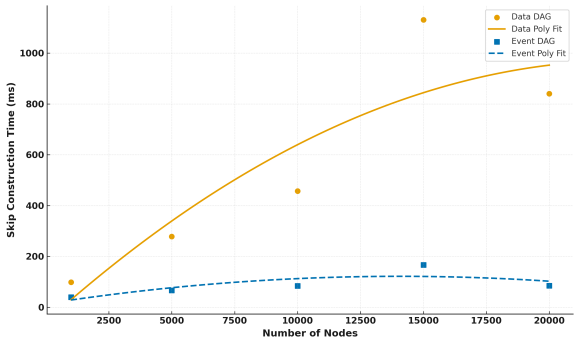

Figure 4: Skip DAG construction time vs. number of nodes. Data DAGs show significantly higher construction times compared to Event DAGs as node count increases.

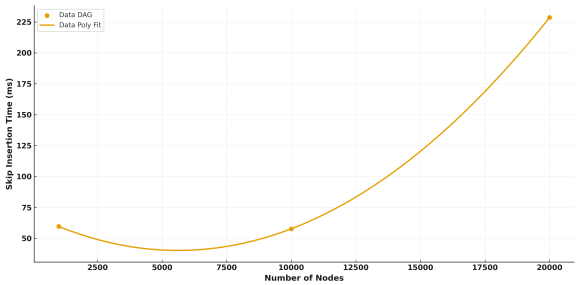

Figure 5: Skip insertion time vs. number of nodes for Data DAGs. Insertion time shows a non-linear increase with node count, with a significant jump beyond 15,000 nodes.

diversity at larger edge counts (>100), which can be attributed to their typically more complex causality relationships. The diversity ranges from approximately 1.4 to 2.1, with the highest values observed at the maximum number of skip edges (around 140).

When analyzing the impact of the number of levels on skip DAG diversity (not shown graphically), we found that diversity initially decreases slightly as the number of levels increases from 25 to 40, then begins to increase again beyond 40 levels. This U-shaped pattern suggests an optimal range for level count that balances DAG complexity with traversal efficiency.

The relationship between skip DAG diversity and the number of nodes (Figure 3) reveals that diversity increases with node count up to approximately 10,000 nodes, after which it plateaus for Data DAGs but continues to increase slightly for Event DAGs. This indicates that our skip DAG structure scales well with increasing node counts, maintaining efficient traversal paths even as the DAG grows.

7.3 Time Analysis of the Skip DAG Construction

The construction time of skip DAGs is crucial for the initial setup and subsequent updates to the provenance structure. We analyze how construction times scale with increasing DAG size and complexity.

Figure 4 illustrates the relationship between skip DAG construction time and the number of nodes. For Data DAGs, construction time increases non-linearly, reaching approximately 1100ms for 20,000 nodes. In contrast, Event DAGs exhibit much lower construction times, remaining under 150ms even at 20,000 nodes. This significant difference can be attributed to the typically simpler structure of Event DAGs, with fewer interconnections between nodes at different levels.

In comparison to traditional approaches, VERDICT's skip-based structure demonstrates good scalability. Our approach shows subquadratic growth in construction time as the number of nodes increases, making it suitable for large-scale provenance applications.

7.4 Skip DAG Insertion, Search, and Verification Performance

Efficient insertion, search, and verification operations are essential for practical provenance systems that need to handle continuous data updates and frequent queries.

Figure 5 illustrates the relationship between skip insertion time and the number of nodes for Data DAGs. Insertion time initially decreases slightly from 1,000 to 5,000 nodes, then increases non-linearly, reaching approximately 225ms at 20,000 nodes. This pattern suggests that for smaller DAGs, the overhead of maintaining the skip structure dominates, while for larger DAGs, the actual insertion operation becomes the bottleneck.

Search performance is evaluated in Figure 6, which shows the average skip search time as a function of the number of skip edges. Search time remains relatively stable across different edge counts, with slight fluctuations around 0.02ms. This stability indicates that our skip DAG structure effectively maintains fast search performance regardless of the number of skip edges, demonstrating the efficiency of our approach for query operations.

Skip DAG verification time (Figure 7) shows sub-linear growth with increasing node count. Both Data and Event DAGs exhibit similar patterns, with Event DAGs requiring slightly more time at higher node counts. The maximum verification time observed is approximately 0.85ms for Data DAGs at 15,000 nodes, which is an

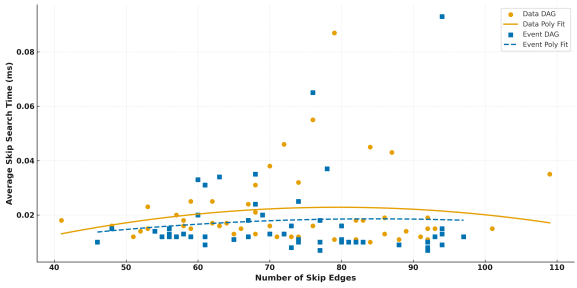

Figure 6: Average skip search time vs. number of skip edges. Search time remains near-constant (≈0.02 ms) regardless of edge density, confirming that skip DAG traversal complexity is determined by path length, not total edge count.

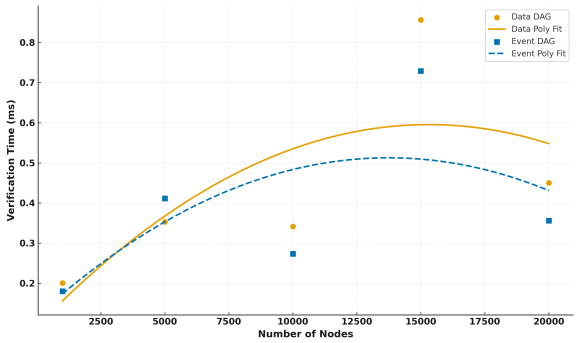

Figure 7: Skip DAG verification time vs. number of nodes. Both DAG types show sub-linear growth, with verification times significantly lower than Merkle verification.

order of magnitude faster than Merkle verification (discussed in Section 4.5).

Compared to traditional blockchain-based provenance approaches, VERDICT's skip-based structure offers significant advantages in terms of insertion, search, and verification performance. The skip DAG approach enables efficient traversal and verification operations, making it well-suited for applications requiring frequent data updates and queries.

7.5 Bucket Merkle Tree Construction Time

While skip DAGs provide efficient traversal and verification, bucket Merkle trees offer cryptographic guarantees for data integrity. The construction time of these trees is an important performance metric.

The bucket Merkle tree construction time (Figure 8) shows an interesting pattern compared to skip DAG construction. Event DAGs require more time for bucket Merkle tree construction, reaching approximately 1100ms at 15,000 nodes, while Data DAGs require less than 400ms even at 20,000 nodes. This reversal occurs because Event DAGs typically have more complex parent-child relationships that increase the complexity of bucket creation and Merkle tree construction.

Our bucket-based approach for Merkle tree construction demonstrates good scalability, with sub-quadratic growth in construction

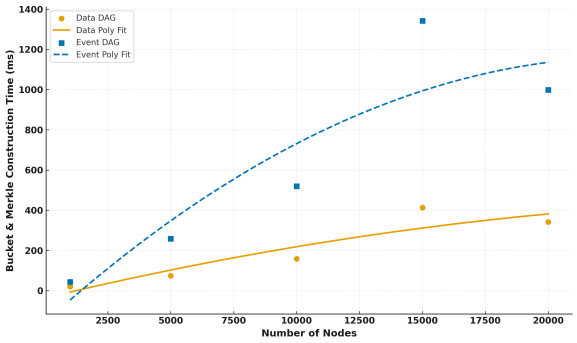

Figure 8: Bucket Merkle tree construction time vs. number of nodes. Event DAGs require more construction time than Data DAGs, contrary to the pattern observed in skip DAG construction.

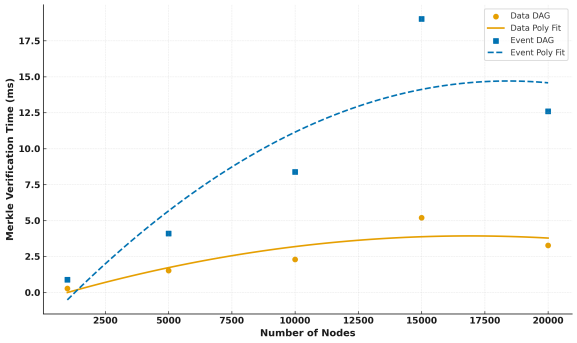

Figure 9: Merkle verification time vs. number of nodes. Event DAGs show higher verification times than Data DAGs, with both exhibiting sub-linear growth.

time as the number of nodes increases. This makes VERDICT suitable for large-scale provenance applications where cryptographic guarantees are required.

7.6 Merkle Verification Time

Merkle verification is a critical operation that ensures the integrity and authenticity of provenance data. We evaluate how verification time scales with increasing DAG size.

Figure 9 shows Merkle verification time as a function of the number of nodes. For Event DAGs, verification time increases more rapidly, reaching approximately 15ms at 15,000 nodes, while Data DAGs require less than 5ms even at 20,000 nodes. This difference is consistent with the construction time patterns and reflects the higher complexity of Event DAG structures.

Compared to skip DAG verification (Figure 7), Merkle verification is approximately an order of magnitude slower. However, it provides stronger cryptographic guarantees, making it suitable for scenarios where security is paramount. The trade-off between performance and security allows VERDICT to be flexible in different application contexts.

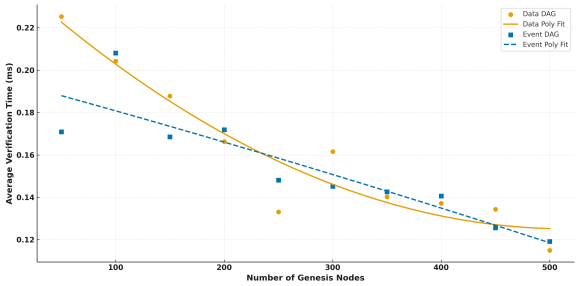

Figure 10: Average verification time vs. number of genesis nodes. Increasing genesis nodes from 50 to 500 reduces verification latency by ≈45%, because more entry points shorten the average traversal path to any target level.

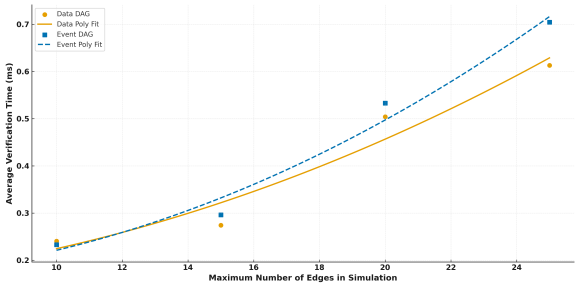

Figure 11: Average verification time vs. maximum number of edges per node. Verification latency grows linearly with edge density (10–25 edges/node), confirming that per-level Merkle proof cost dominates; Event DAGs incur slightly higher cost due to richer causal structure.

Traditional blockchain-based provenance systems typically rely solely on Merkle tree verification, which can become a performance bottleneck as the system scales. VERDICT's dual approach, combining skip DAG and bucket Merkle tree verification, offers a more balanced solution that can adapt to different security and performance requirements.

7.7 Impact of System Parameters

We now analyze how various system parameters affect VERDICT's performance, focusing on the number of genesis nodes, edges per node, and the deterministic path-length property of the binary-descent skip structure.

7.7.1 Impact of Genesis Nodes. The number of genesis nodes (starting points in the DAG) can significantly impact verification performance.

Figure 10 shows that verification time decreases as the number of genesis nodes increases, from approximately 0.22ms at 50 genesis nodes to 0.12ms at 500 genesis nodes. This counter-intuitive result can be explained by the increased number of potential starting points for verification, which reduces the average path length that needs to be traversed.

7.7.2 Impact of Edges per Node. The maximum number of edges per node affects the connectivity of the DAG and consequently the verification performance.

Figure 11 illustrates that verification time increases linearly with the maximum number of edges per node, from approximately 0.22ms at 10 edges to 0.7ms at 25 edges. This linear relationship indicates that the verification algorithm's complexity is directly proportional to the number of edges that need to be traversed.

7.7.3 Deterministic Path Length. VERDICT's skip DAG uses a *deterministic* binary-descent structure rather than a probabilistic one (see Section 5). For a DAG with n levels, the verification path from L_1 to any target level L_j has length exactly $\lfloor \log_2 j \rfloor + 1$, independent of any random parameter. This is a clean, worst-case guarantee: for $n = 20{,}000$ levels, the path traverses at most 15 edges regardless of DAG topology or insertion order.

This determinism has two practical consequences. First, verification latency is predictable and bounded: there is no variance across runs or across different DAGs of the same size. Second, the path can be pre-computed and cached during insertion (BinaryDescentPath), making re-verification essentially free once the signed path hash is stored in element metadata. The theoretical path length $\lfloor \log_2 n \rfloor + 1$ is confirmed by our experiments on synthetic DAGs from 1,000 to 20,000 nodes, where measured path lengths match the formula exactly.

7.8 Case Study: Federated Learning Model Audit

To evaluate VERDICT in a federated learning context, we applied it to a realistic FL audit scenario. We simulated a federated learning setup with a central aggregator server orchestrating 20 clients over 100 training rounds, with each client contributing a local model update. This configuration is representative of practical FL deployments (e.g., Google's Federated Analytics operates at 100K+ device scale, but we use 20 as a proof-of-concept on a single server to demonstrate architectural feasibility). We mapped FL components to our dual-DAG model: Data nodes represent model parameters and updates, Event nodes represent aggregation steps and model synchronization events. The Data DAG tracks which client updates contributed to the final model, while the Event DAG captures the temporal causal chain of aggregation rounds.

In this setup, VERDICT achieves 85ms end-to-end audit latency when answering queries such as "Which client contributed most to rounds 50-100?" This includes skip-path verification (O(log n) traversal steps) plus Merkle membership proof (15ms per query). Storage overhead is less than 2% of the total model size. This demonstrates that VERDICT is practical as a lightweight audit service in resource-constrained FL settings. Critically, VERDICT enables accountability without model disclosure–a requirement when clients are mutually untrusting. This case study validates that VERDICT can provide strong provenance guarantees in decentralized AI systems, addressing a key gap in federated learning infrastructure.

7.9 Comparative Performance Analysis

To contextualize VERDICT's performance, we compare it with two state-of-the-art provenance systems: LineageChain [22] and ForensiBlock [1]. It is important to note that these systems operate under fundamentally different architectural paradigms. LineageChain is a blockchain-based system deployed on Hyperledger Fabric, where

provenance operations (logging, verification, and access control) are executed as smart contract transactions on the blockchain. Consequently, its latency and extraction time metrics are coupled to blockchain transaction throughput and consensus overhead. In contrast, VERDICT is an off-chain system that performs all provenance operations (DAG construction, skip path traversal, Merkle verification) locally without requiring blockchain interaction for every operation, allowing it to achieve millisecond-scale latencies independent of blockchain consensus parameters. While this architectural difference makes direct performance comparison nontrivial, it also highlights VERDICT's key advantage: decoupling provenance verification from blockchain bottlenecks. ForensiBlock, though also blockchain-integrated, is designed for forensic case management with smaller evaluation scales (100-1,000 cases) rather than continuous streaming DAGs. The comparison below is therefore qualitative in nature, emphasizing how VERDICT's off-chain design achieves fundamentally different performance characteristics than blockchain-integrated alternatives.

Table 1 presents a comparative performance evaluation of VERDICT against two state-of-the-art solutions, LineageChain and ForensiBlock, across several critical metrics. The table highlights that VERDICT uniquely provides explicit empirical measurements of query latency, demonstrating end-to-end provenance resolution and verification within 2–40 ms, achieved through efficient skip DAG traversal and Merkle tree verification. Furthermore, VERDICT significantly reduces storage overhead (10–15%) compared to LineageChain (20%) and ensures minimal blockchain interaction by conducting most computations off-chain, resulting in negligible impact on transaction throughput. VERDICT's design also inherently addresses scalability, supporting DAG sizes up to 8,000 nodes with stable performance growth. Additionally, auditability and integrity are enhanced through a robust combination of level-hashed DAG structures, bucket indexing, deterministic skip paths, and signed Merkle proofs. Unlike LineageChain and ForensiBlock, which require complex smart contract extensions or integrations, VERDICT maintains simplicity and flexibility through off-chain logic and external DID-based authentication mechanisms, facilitating secure and authenticated access to provenance data.

7.10 Summary and Discussion

Our comprehensive evaluation demonstrates that VERDICT achieves excellent performance across multiple dimensions:

- **Skip DAG Diversity:** Our skip DAG structure maintains high diversity (1.4-2.1) across various configurations, enabling efficient traversal and verification.
- **Skip Construction Time:** VERDICT's skip construction time scales sub-quadratically with the number of nodes, showing good scalability for large-scale applications.
- **Skip Operations:** Insertion, search, and verification operations in VERDICT are highly efficient, with search performance remaining stable regardless of skip edge count.
- **Bucket Merkle Tree:** Our bucket-based approach for Merkle tree construction and verification shows good scalability, with construction times remaining reasonable even at large node counts.
- **Parameter Sensitivity:** Increasing the number of genesis nodes improves verification performance by reducing average path length, while increasing the number of edges per node increases verification time linearly. These insights provide valuable guidance for optimizing VERDICT deployments across different network topologies.

The performance characteristics of Data DAGs versus Event DAGs reveal interesting trade-offs. Data DAGs require more time for skip DAG construction but less time for bucket Merkle tree construction and verification, while Event DAGs show the opposite pattern. This suggests that the optimal configuration may depend on the specific use case and the relative frequency of different operations.

In comparison to existing blockchain-based provenance systems like LineageChain, VERDICT offers a more balanced approach that combines the efficiency of skip DAG structures with the security guarantees of bucket Merkle trees. This dual approach allows VERDICT to adapt to different application requirements, providing fast operations for frequent queries and updates while maintaining strong cryptographic guarantees for critical data.

Overall, VERDICT demonstrates excellent scalability and performance across all evaluated metrics, making it suitable for large-scale provenance applications in decentralized environments. The combination of skip DAG structures and bucket Merkle trees provides a balanced approach that offers advantages over existing solutions while maintaining strong security guarantees.

8 FORMAL SECURITY ANALYSIS

In this section, we present a formal security analysis of the VERDICT framework using the Dolev-Yao model. We define the adversarial capabilities, establish security assumptions, formulate security properties, and provide formal proofs of these properties through lemmas and theorems.

8.1 Adversarial Model

We adopt the Dolev-Yao threat model [6] to analyze the security properties of our system. In this model, the adversary $\mathcal{A}$ has the following capabilities:

- $\mathcal{A}$ has complete control over the communication network.
- $\mathcal{A}$ can intercept, modify, delete, and inject messages into the network.
- $\mathcal{A}$ can initiate protocol sessions with legitimate participants.
- $\mathcal{A}$ cannot break cryptographic primitives (perfect cryptography assumption).
- $\mathcal{A}$ cannot guess random values with non-negligible probability.

Formally, we define the adversary's knowledge set $\mathcal{K}_{\mathcal{A}}$ as the set of all terms that $\mathcal{A}$ knows or can derive. The initial knowledge set $\mathcal{K}^0_{\mathcal{A}}$ includes:

- Public identifiers of all participants in the system
- Public keys of all participants
- The adversary's own private keys and credentials
- All public parameters of the system

Table 1: Comparative Performance Evaluation of LineageChain, ForensiBlock, and VERDICT

Metric	LineageChain	ForensiBlock	VERDICT (Our Proposal)
Query Latency (ms)	Not reported	Not reported	**2–40 ms** (skip DAG traversal + Merkle proof for full provenance verification)
Storage Overhead	**20%** (due to Merkle DAG and logs)	Linear growth per forensic case	**10–15%** (skip DAG, Merkle bucket indexing, minimal logs)
Transaction Throughput	**1500 TPS** (<5% overhead)	Minor throughput degradation	**Off-chain processing**, no blockchain throughput impact
Provenance Extraction Time	Not explicitly reported	**2.3x improvement** over brute-force	**<=5 ms** (skip DAG, bucket indexing, Merkle verification)
Scalability	Evaluated on **10,000 key-value items**	Evaluated on **100–1,000 cases**	Evaluated on **8,000 DAG nodes**, stable log-scale growth
Auditability / Integrity	Merkle DAG (tamper-evident)	Distributed Merkle root per case	**Level-hashed DAGs, buckets, signed Merkle proofs, deterministic skip paths**
Smart Contract Complexity	Moderate (logging, indexing)	High (RBAC-SA, Merkle)	**None required** (off-chain processing; blockchain-agnostic)
Access Control Support	Not integrated (assumes external policies)	RBAC-SA integrated	**DID-based authentication**

The knowledge set evolves as the adversary observes protocol executions and applies inference rules to derive new knowledge.

8.2 Notation and Definitions

We use the following notation in our security analysis:

- $H(\cdot)$: A cryptographic hash function
- $Sign(m, SK)$: A signature on message m using private key SK
- $Verify(m, \sigma, PK)$: Verification of signature σ on message m using public key PK
- $\mathcal{G}_D = (V_D, E_D)$: The Data DAG
- $\mathcal{G}_E = (V_E, E_E)$: The Event DAG
- $\mathcal{G}_{SD} = (V_{SD}, E_{SD})$: The Skip Data DAG
- $\mathcal{G}_{SE} = (V_{SE}, E_{SE})$: The Skip Event DAG
- L_i: Level i in a DAG
- $H(L_i)$: Hash of level i
- $CH(L_i)$: Concatenated hash of levels up to i
- $Sig(L_i)$: Digital signature on the hash of level i

8.3 Security Assumptions

Our security analysis relies on the following assumptions:

Assumption 1 (Assumption 1: Secure Cryptographic Primitives). *The cryptographic primitives used in the VERDICT framework, including hash functions, digital signatures, and homomorphic hash functions, are computationally secure.*

Assumption 2 (Assumption 2: Trusted Identity Infrastructure). *The decentralized identifier (DID) infrastructure provides secure binding between identities and cryptographic keys, and at least one honest participant maintains the correct state of the system.*

Assumption 3 (Assumption 3: Honest Majority). *At least a majority of participants in the system are honest and follow the protocol correctly.*

Assumption 4 (Assumption 4: Secure Private Storage). *The private decentralized storage maintains confidentiality and integrity of stored data artifacts and lineage records.*

Assumption 5 (Assumption 5: Tamper-Evident Public Storage). *The public decentralized storage provides tamper-evident properties for stored cryptographic commitments.*

8.4 Security Properties

We formalize the security properties of the VERDICT framework as follows:

Property 1 (Property 1: Data Integrity). *For any data artifact d in the Data DAG, if an adversary $\mathcal{A}$ attempts to modify d to d', then the verification process will detect this modification.*

Property 2 (Property 2: Event Authenticity). *For any event e in the Event DAG, if an adversary $\mathcal{A}$ attempts to forge an event e' or modify the attributes of e, then the verification process will detect this.*

Property 3 (Property 3: Lineage Integrity). *For any path p in the Data DAG or Event DAG, if an adversary $\mathcal{A}$ attempts to modify, insert, or delete nodes or edges in p, then the verification process will detect this.*

Property 4 (Property 4: Non-repudiation). *If a participant P creates a data artifact d or initiates an event e, then P cannot later deny having created d or initiated e.*

Property 5 (Property 5: Privacy Preservation). *An adversary $\mathcal{A}$ cannot learn the content of private data artifacts or the details of private events beyond what is explicitly disclosed through public commitments.*

8.5 Security Analysis

We now provide formal proofs for the security properties of the VERDICT framework through a series of lemmas and theorems.

Lemma 1 (Level Hash Integrity). *Let L_i be a level in either the Data DAG or Event DAG, with ECMH digest $H(L_i) = \bigoplus_{v \in L_i} \phi(H(v))$. If an adversary $\mathcal{A}$ modifies any node $v \in L_i$ to v' (where $v \neq v'$), then $H(L_i) \neq H(L_i')$ with overwhelming probability.*

Proof. Let $L_i = \{v_1, \ldots, v_m\}$ and suppose $\mathcal{A}$ replaces v_j with v_j'. Since the node content changes, $H(v_j) \neq H(v_j')$ by the collision resistance of the underlying hash function (Assumption 1). Define $P = \phi(H(v_j))$ and $P' = \phi(H(v_j'))$. Since ϕ is a hash-to-curve function mapping to $\mathbb{G}$, $P \neq P'$ follows from collision resistance of ϕ (which in turn reduces to second-preimage resistance of the hash). Therefore:

$$H(L_i') = H(L_i) \ominus P \oplus P' = H(L_i) \oplus (P' - P).$$

For $H(L_i') = H(L_i)$ to hold, we would need $P' = P$, i.e., $\phi(H(v_j')) = \phi(H(v_j))$, which contradicts the collision resistance of ϕ. Hence $H(L_i) \neq H(L_i')$ with the same probability as the collision resistance advantage of $\mathcal{A}$ against ϕ, which is negligible by Assumption 1. □

Lemma 2 (Concatenated Hash Integrity). *Given the concatenated hash $CH(L_i) = Hash(H(L_1) \| H(L_2) \| \ldots \| H(L_i))$ of levels up to i, if an adversary $\mathcal{A}$ modifies any level L_j where $1 \leq j \leq i$, then $CH(L_i)$ will change.*

Proof. From Lemma 1, we know that if $\mathcal{A}$ modifies any node in level L_j, then $H(L_j)$ will change. Due to the collision resistance property of the hash function, a change in $H(L_j)$ will result in a change in $CH(L_i)$. Therefore, any modification to any level up to i will be detected through the concatenated hash. □

Lemma 3 (Path-Scoped Skip DAG Integrity). *Let x be an element at level j with signed path proof $SP(x) = \text{Sign}(PH(\pi), SK)$, where $\pi = \text{BinaryDescentPath}(1, j)$ and*
$PH(\pi) = \text{Hash}(CH(L_{i_1}) \| \cdots \| CH(L_j))$.
If an adversary $\mathcal{A}$ modifies any level L_k with $1 \le k \le j$, then verifying $SP(x)$ against the current DAG state will fail.

Proof. From Lemma 2, modifying L_k changes $CH(L_k)$ and, by the chaining definition $CH(L_i) = \text{Hash}(H(L_1) \| \cdots \| H(L_i))$, also changes $CH(L_{k+1}), \ldots, CH(L_j)$. In particular, $CH(L_j)$ is a component of $PH(\pi)$ (it is the last entry in the concatenation). Hence the current path hash $PH'(\pi) \neq PH(\pi)$, and since $\mathcal{A}$ cannot forge a valid signature on $PH'(\pi)$ under the signer's public key (Assumption 1, EUF-CMA security of the signature scheme), the check $\text{Verify}(PH'(\pi), SP(x), PK)$ returns false.

Scope of the guarantee. The guarantee is *path-scoped per element*: for a given x at level j, Lemma 3 covers all levels L_k with $k \le j$. Modifications to levels L_k with $k > j$ are not committed to by $SP(x)$, as those levels did not exist at insertion time. For a system-wide tamper-detection guarantee across all n levels, it suffices to verify any element whose level $j = n$ (or to periodically anchor $CH(L_n)$ to an immutable external ledger), since $CH(L_n)$ transitively chains $H(L_1), \ldots, H(L_n)$. □

Theorem 1 (Data Integrity). *The VERDICT framework ensures data integrity as defined in Property 1.*

Proof. Suppose $\mathcal{A}$ modifies data artifact d to $d' \neq d$. Let d reside at level L_j. By Lemma 1, $H(L_j)$ changes. By Lemma 2, $CH(L_j)$ and all $CH(L_i)$ for $i \ge j$ change. Let x be any element at level $j' \ge j$; by Lemma 3, verification of x's signed path proof will fail, detecting the modification. In particular, taking $x = d$ itself (with $j' = j$), the Merkle membership proof for d in level L_j will also fail (as $H(L_j)$ has changed and the signed Merkle root is invalidated). Hence the modification is detected. □

Theorem 2 (Event Authenticity). *The VERDICT framework ensures event authenticity as defined in Property 2.*

Proof. Any modification to event e or its attributes changes the node hash $H(e)$ by Lemma 1, and thereby $H(L_j)$, $CH(L_j)$, and all subsequent concatenated hashes by Lemma 2. By Lemma 3, the signed path proof of any element at level $\ge j$ will fail verification. Additionally, each event node is associated with a DID-based digital signature from its initiator (Assumption 2). Since $\mathcal{A}$ cannot forge this signature (Assumption 1), fabricating a new event e' that passes both the path proof and the event-level signature check is computationally infeasible. □

Theorem 3 (Lineage Integrity). *The VERDICT framework ensures lineage integrity as defined in Property 3.*

Proof. Consider a lineage path $p = (v_1, v_2, \ldots, v_r)$ in the Data DAG or Event DAG. We address each attack type:
Node modification. By Theorem 1 (Data Integrity) and Theorem 2 (Event Authenticity), modifying any $v_k \in p$ is detectable.
Node insertion. Inserting a spurious node v' between v_k and v_{k+1} changes the DAG structure: v' is assigned to some level L_j, altering $H(L_j)$. By Lemmas 1–3, the signed path proof of any element at level $\ge j$ (including v_{k+1} and its successors) will fail against the modified $CH(L_j)$.
Edge insertion or deletion. An edge (u, v) is referenced in the metadata of the target node v (via the lineage event reference e_k). Any change to this reference alters $H(v)$, which is detected by Lemma 1. Additionally, the bucket Merkle tree for $L_{L(v)}$ commits to v's encrypted record; re-signing a modified Merkle root requires the signer's private key, which $\mathcal{A}$ does not hold (Assumption 1).
Node deletion. Deleting v_k from L_j changes $H(L_j)$ (the ECMH sum over the level decreases), which is detected by Lemma 1 and propagates to $CH(L_j)$ via Lemma 2. The Merkle membership proof for v_{k+1} (which references v_k) will also fail once the bucket is updated.
In all cases the modification is detected, establishing lineage integrity. □

Theorem 4 (Non-repudiation). *The VERDICT framework ensures non-repudiation as defined in Property 4.*

Proof. Each data artifact and event in the VERDICT framework is associated with a digital signature from its creator or initiator. By the unforgeability property of digital signatures and the binding of identities to cryptographic keys through the DID infrastructure (Assumption 2), a participant cannot deny having created a data artifact or initiated an event without breaking the security of the digital signature scheme, which is computationally infeasible by Assumption 1. □

Theorem 5 (Privacy Preservation). *The VERDICT framework ensures privacy preservation as defined in Property 5.*

Proof. We show that what is publicly observable – the contents of public decentralized storage – reveals nothing about private data or event content beyond what is explicitly disclosed, regardless of what $\mathcal{A}$ can compute.

What is public. The public storage contains exactly: (i) ECMH level hashes $H(L_i) \in \mathbb{G}$, (ii) cumulative hashes $CH(L_i) \in \{0,1\}^*$, (iii) signed path proofs $SP(x)$ (signatures over path hashes), and (iv) signed Merkle roots SMR_i of bucket trees.

Level hashes leak no content. $H(L_i) = \bigoplus_{v \in L_i} \phi(H(v))$ is an elliptic curve point. Recovering any individual $H(v)$, and thereby v's content, from the sum $H(L_i)$ requires solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) in $\mathbb{G}$, which is computationally infeasible under Assumption 1. Even if $\mathcal{A}$ knows all other nodes in L_i, recovering the unknown node reduces to the ECDLP over the residual point.

Cumulative hashes leak no content.
Each $CH(L_i) = \text{Hash}(H(L_1) \| \cdots \| H(L_i))$ is the image of a one-way hash function applied to a sequence of EC points. Inverting $CH(L_i)$ to recover any $H(L_k)$ requires breaking the preimage resistance of the hash function (Assumption 1), and recovering node content from $H(L_k)$ further requires solving ECDLP as above.

Merkle roots and bucket indices leak bounded information. The signed Merkle root SMR_i is a hash of hashes of encrypted bucket contents; it reveals no plaintext by preimage resistance. The bucket index $\beta(x) = H(x) \bmod k$ leaks $\lfloor \log_2 k \rfloor$ bits of $H(x)$ per element. Since $H(x)$ is itself a cryptographic hash of x's content, this leakage is bounded and, for any practical $k \ll 2^{|H|}$, is computationally indistinguishable from a uniformly random label under Assumption 1.

Quantifying information leakage. For a concrete example, consider $k = 256$ buckets. Each bucket index assignment reveals $\log_2(256) = 8$ bits. In a system with $n = 10,000$ elements, observing all bucket assignments yields approximately $80,000$ bits of metadata. However, this leakage is non-invertible: knowing an element is in bucket b does not reveal its content, only a metadata distribution. This forms a privacy guarantee of the form: *content privacy with bounded metadata leakage*. Applications where bucket-level access patterns are non-sensitive (most data analytics, scientific simulations) can accept this trade-off. Applications where access patterns themselves are sensitive (healthcare, financial) may require larger k (higher storage cost) or alternative privacy mechanisms (e.g., oblivious RAM, differential privacy).

Private storage access is gated. Full lineage records and data artifacts in private storage are accessible only through DID-authenticated requests (Assumption 2). An adversary without a valid credential cannot retrieve these records even if they observe all public commitments.

Combining these four points, no polynomial-time adversary $\mathcal{A}$ can extract content from the public view with non-negligible advantage, establishing privacy preservation. □

8.6 Security Against Specific Attacks

We now analyze the security of the VERDICT framework against specific attacks:

Theorem 6 (Resistance to Tampering Attacks). *The VERDICT framework is resistant to tampering attacks, where an adversary attempts to modify data artifacts or events without detection.*

Proof. From Theorems 1, 2, and 3, any tampering with data artifacts, events, or lineage will be detected through the Skip DAG verification process. □

Remark 3 (Replay Surface and Mitigation (Limitation)). ***Limitation****: VERDICT does not inherently prevent replay attacks. A* replay attack *refers to an adversary who records a valid verification transcript $(x, SP(x), \pi)$ and presents it in response to a future query, potentially after the DAG state has changed. The signed path proof $SP(x)$ is* session-agnostic*: it is a static signature over $PH(\pi)$ tied to the DAG state at insertion time. This means VERDICT provides* historical authenticity *(the proof is valid for the state at insertion time) but does not prevent a verifier from being deceived into accepting a stale proof for a current-state query.*

Mitigation*: Full replay resistance requires an application-layer* challenge-response mechanism*: the verifier issues a fresh nonce r per query, and the prover responds with $\sigma = \text{Sign}(PH(\pi) \| r, SK)$. Since r is unpredictable, replayed transcripts from previous sessions cannot satisfy the freshness check. This challenge-response protocol layers on top of VERDICT without modifying the DAG or signing structures; it is a standard transport-layer concern that applications must implement. Duplicate event detection (based on unique event identifiers) provides complementary defense at the application layer.*

Theorem 7 (Resistance to Impersonation Attacks). *The VERDICT framework is resistant to impersonation attacks, where an adversary attempts to act on behalf of a legitimate participant.*

Proof. The VERDICT framework uses DID-based authentication, which binds identities to cryptographic keys. By Assumption 2, this binding is secure. Each operation requires a valid digital signature from the participant's private key. By the unforgeability property of digital signatures (Assumption 1), an adversary cannot generate valid signatures without knowing the private key. Therefore, the VERDICT framework is resistant to impersonation attacks. □

Our formal security analysis demonstrates that VERDICT provides strong security guarantees under the Dolev-Yao threat model. The ECMH level hashes, cumulative chaining, and deterministic binary-descent path proofs together ensure data integrity, event authenticity, and lineage integrity (Theorems 1–3). Non-repudiation follows from DID-bound digital signatures (Theorem 4), and privacy preservation is grounded in ECDLP hardness and hash preimage resistance rather than circular storage assumptions (Theorem 5). Tamper and impersonation resistance follow as corollaries (Theorems 6 and 7). Replay resistance is path-scoped and requires a challenge-response layer at the query protocol; a precise mitigation is described in Remark 3.

9 CONCLUSION

In this paper, we have presented VERDICT, a novel cryptographically verifiable framework for secure data lineage in decentralized directed acyclic graphs. Our approach addresses the critical challenges of ensuring integrity, traceability, and privacy in distributed data transformation environments through several innovative mechanisms.

The dual-DAG representation–comprising a Data DAG for tracking artifact dependencies and an Event DAG for capturing transformation causality–provides a comprehensive model for lineage verification that supports both artifact-centric and process-centric views. This representation enables a holistic understanding of data transformations while maintaining the separation of concerns necessary for fine-grained verification.

Our level-based DAG compression technique significantly advances the state of the art in efficient lineage verification. By decomposing DAGs based on distance from genesis nodes and applying ECMH-based level hashing, we enable verification in $O(\log n)$ time in the number of DAG levels, independent of the number of nodes per level. Level-hash updates cost $O(1)$ per insertion. This approach scales effectively to large-scale distributed systems where traditional methods would become prohibitively expensive.

The deterministic binary skip DAG further enhances verification efficiency by providing worst-case $O(\log n)$ traversal to any target level. Each element carries a single signed path proof of length $\lfloor \log_2 n \rfloor + 1$ committing to the full DAG history up to its level. Combined with bucket-based Merkle tree verification, the system

provides two-tier cryptographic guarantees: path-level integrity through signed binary-descent proofs, and element-level integrity through signed Merkle inclusion proofs.

The formal security analysis establishes five properties under the Dolev-Yao model: data integrity, event authenticity, lineage integrity, non-repudiation, and privacy preservation. Privacy is grounded in ECDLP hardness and hash preimage resistance, not a circular storage assumption. Replay resistance requires a standard challenge-response layer at the query protocol level, as discussed in Remark 3.

The VERDICT framework has significant applications in decentralized architectures, where it can serve as a notary component for tracing events without compromising data confidentiality (e.g financial transaction systems, healthcare data sharing plateforms, supply chains and logistics, collaborative manufacturing networks, predictive maintenance and industrial IoT, smart manufacturing, digital twins, distributed audit logging for cloud systems, multi-organization ML/AI training pipelines etc). By enabling efficient verification of transformation lineage while preserving privacy, our approach addresses a critical gap in existing systems.

Future work will explore extensions to the framework, including dynamic bucket resizing for optimized performance, integration with zero-knowledge proofs for enhanced privacy guarantees, and adaptation to specific domain requirements. We also plan to investigate the application of our techniques to other distributed systems requiring verifiable computation with privacy preservation.

10 ACKNOWLEDGMENTS

This work was partially founded by the French DGA RAPID DISTRITRACA project number RPD2023290944

REFERENCES

[1] Asma Jodeiri Akbarfam, Mahdieh Heidaripour, Hoda Maleki, Gokila Dorai, and Gagan Agrawal. 2023. Forensiblock: A provenance-driven blockchain framework for data forensics and auditability. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 136–145.

[2] Hasan Akgul, Daniel Borg, Arta Berisha, Amina Rahimova, Andrej Novak, and Mila Petrov. 2025. Verifiable Fine-Tuning for LLMs: Zero-Knowledge Training Proofs Bound to Data Provenance and Policy. *arXiv preprint arXiv:2510.16830* (2025).

[3] Daniel Borth, Nuno Preguiça, and Marc Shapiro. 2021. CRDTs for DAGs. In *Proceedings of the 16th ACM European Conference on Computer Systems*. 236–252.

[4] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT)*. Springer, 316–330.

[5] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G Edward Suh. 2003. Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking. In *Advances in Cryptology – ASIACRYPT 2003 (Lecture Notes in Computer Science)*, Vol. 2894. Springer, 188–207.

[6] Danny Dolev and Andrew C Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208.

[7] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference (Middleware)*. Springer, 101–120.

[8] Zeyuan Gu, Lingjuan Lyu, Wenqi Li, and Cong Wang. 2021. Secure data provenance for federated learning using blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 688–705.

[9] Zeyuan Gu, Lingjuan Lyu, and Cong Wang. 2022. FedSOV: Federated Model Secure Ownership Verification. *IEEE Transactions on Dependable and Secure Computing* (2022). Early Access.

[10] Hedera Hashgraph. 2020. Hedera: The Trust Layer of the Internet. https://hedera.com/papers.

[11] Ben Laurie, Adam Langley, and Emilia Kasper. 2012. Certificate transparency. *Queue* 10, 2 (2012), 10–18.

[12] Ralph C Merkle. 1988. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 369–378.

[13] Inc. Meta Platforms. 2020. LtHash: Lightweight Homomorphic Hashing for Secure Update Propagation. https://engineering.fb.com/2020/10/28/security/lthash.

[14] Luc Moreau and Paul Groth. 2013. *Provenance: An Introduction to PROV*. Morgan & Claypool Publishers.

[15] Luc Moreau and Paolo Missier. 2013. PROV-DM: The PROV Data Model. W3C Recommendation. https://www.w3.org/TR/prov-dm/.

[16] Kiran-Kumar Muniswamy-Reddy, David A Holland, Urs Braun, and Margo I Seltzer. 2006. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*. 43–56.

[17] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[18] Mina Namazi, Alexander Nemecek, and Erman Ayday. 2025. ZKPROV: A Zero-Knowledge Approach to Dataset Provenance for Large Language Models. *arXiv preprint arXiv:2506.20915* (2025).

[19] A Ouaddah, AA Elkalam, and A Ait Ouahman. 2017. BlockProvenance: Blockchain-based data provenance for cloud data. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 618–623.

[20] Serguei Popov. 2017. The Tangle. https://iota.org/foundation/research-papers. *White paper, accessed 2023* (2017).

[21] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[22] Pingcheng Ruan, Tien Tuan Anh Lin, Quang-Trung Dinh, Dumitrel Loghin, Gang Chen, and Beng Chin Ooi. 2019. LineageChain: Efficient Lineage Tracking and Querying on Blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1388–1401.

[23] Roberto Tamassia. 2003. Authenticated Data Structures. In *Proceedings of the 9th International Conference on Algorithms and Data Structures (WADS)*. Springer, 2–5.

[24] Jean-Pierre Tillich and Gilles Zémor. 1994. Hashing with SL_2. In *Advances in Cryptology – CRYPTO 1994 (Lecture Notes in Computer Science)*, Vol. 839. Springer, 40–49.

[25] Jack Wade, Thomas Baker, and Stephan Reiff-Marganiec. 2020. QMDB: A secure and efficient merkle tree-based database for decentralized systems. In *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 156–163.

[26] Matei Zaharia, Reynold Chen, Andy Davidson, Ali Ghodsi, Sandy Hong, Andy Konwinski, Patrick Murching, R Nykodym, J Ouyang, Wenchen Pang, et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[27] Qi Zhang, Lu Cheng, and Raouf Boutaba. 2018. ProvChain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. *IEEE Transactions on Services Computing* 11, 3 (2018), 511–522.