ARTIFACT EVALUATED
USENIX
AVAILABLE

VCAUSE: Efficient and Verifiable Causality Analysis for Cloud-based Endpoint Auditing

Qiyang Song[1,2], Qihang Zhou[1,2,*], Xiaoqi Jia[1,2,*], Zhenyu Song[1,2]
Wenbo Jiang[3], Heqing Huang[5], Yong Liu[4], Dan Meng[1,2]
[1] *Institute of Information Engineering, Chinese Academy of Sciences*
[2] *School of Cyber Security, University of Chinese Academy of Sciences*
[3] *University of Electronic Science and Technology of China*
[4] *Qi An Xin Technology Group Inc.*, [5] *Independent Researcher*
{songqiyang, zhouqihang, jiaxiaoqi, songzhengyu}@iie.ac.cn
wenbo_jiang@uestc.edu.cn, {heqing.state,mengdan255}@gmail.com, liuyong03@qianxin.com

Abstract

In cloud-based endpoint auditing, security administrators often rely on the cloud to perform causality analysis over log-derived versioned provenance graphs to investigate suspicious attack behaviors. However, the cloud may be distrusted or compromised by attackers, potentially manipulating the final causality analysis results. Consequently, administrators may not accurately understand attack behaviors and fail to implement effective countermeasures. This risk underscores the need for a defense scheme to ensure the integrity of causality analysis. While existing tamper-evident logging schemes and trusted execution environments show promise for this task, they are not specifically designed to support causality analysis and thus face inherent security and efficiency limitations.

This paper presents VCAUSE, an efficient and verifiable causality analysis system for cloud-based endpoint auditing. VCAUSE integrates two authenticated data structures: a graph accumulator and a verifiable provenance graph. The data structures enable validation of two critical steps in causality analysis: (i) querying a point-of-interest node on a versioned provenance graph, and (ii) identifying its causally related components. Formal security analysis and experimental evaluation show that VCAUSE can achieve secure and verifiable causality analysis with only $< 1\%$ computational overhead on endpoints and 3.36% on the cloud.

1 Introduction

Cloud-based endpoint auditing [16, 19, 26, 27, 30, 59] has become a vital security infrastructure for enterprises. It collects system event logs from enterprise endpoints and leverages cloud computing to perform large-scale analysis, particularly in investigating advanced persistent threats (APTs) [36]. A key log analysis technique is causality analysis [20, 31, 42], typically performed on (versioned) provenance graphs [40] derived from system logs. It correlates causality dependencies within these graphs, enabling security administrators to trace the information flow of suspicious entities and assess their impact. Maintaining the integrity of causality analysis is critical, as it empowers administrators to fully understand attack behaviors and deploy effective countermeasures.

However, the integrity of causality analysis cannot be fully ensured in a cloud-based auditing infrastructure. Typically, attackers who have infiltrated the infrastructure may alter stored logs to erase traces of their activities, rendering the associated causality analysis incomplete. This threat is well documented—recent reports show that 72% of security analysts have encountered log tampering [15, 24]. Beyond external attacks, the cloud itself may be untrustworthy [12, 63] in preserving the integrity of logs and associated causality analysis. For instance, when managing large volumes of daily endpoint logs (*e.g.,* 50 GB from 100 endpoints [21]), the cloud may silently discard a portion of the logs due to misconfiguration or self-interest in conserving resources. In summary, these threats lead to incomplete causality analysis, hindering accurate and comprehensive attack investigation.

In practice, while fully preventing this integrity issue may be infeasible, it is vital to develop a practical defense that allows third parties to verify the integrity of causality analysis in the cloud. To our knowledge, no existing solutions are specifically designed for this purpose. Although tamper-evident logging schemes [1, 38, 44, 50] and trusted execution environments (TEEs) [51] show promise in offering partial support, they still face the following challenges.

Challenge #1. Existing tamper-evident endpoint logging schemes [1, 44, 50] can secure log collection at endpoints and support validation of all produced logs. However, they do not inherently support causality analysis validation in the cloud. Extending these schemes for such validation typically poses significant efficiency challenges: the administrators should retrieve and verify the *entire set* of endpoint logs from the cloud, reconstruct the provenance graph, and re-run the causality analysis to confirm the results. While deploying TEEs in the cloud can offload this burden from administrators, providing efficient causality analysis still remains challenging, as processing massive logs in enclaves often incurs substantial

*Corresponding authors.

enclave-to-nonenclave context switch overheads.

Challenge #2. To improve efficiency, certificate transparency [38] can be employed for flexible validation of individual logs. With this approach, administrators can selectively retrieve and verify only the subset of logs relevant to a given causality analysis query, and then examine the corresponding results. However, this approach introduces a *completeness problem*: administrators cannot ascertain whether the retrieved logs—or the resulting causality analysis results—are complete with respect to the original query.

In this paper, we present VCAUSE, an efficient and verifiable causality analysis system for cloud-based endpoint auditing. Unlike existing tamper-evident logging schemes, VCAUSE enables direct validation of causality analysis on versioned provenance graphs. It employs two new authenticated data structures: a *graph accumulator* and a *verifiable versioned provenance graph*, which provide cryptographic proofs for two analysis steps: (i) querying a point-of-interest node, *i.e.,* a version of a suspicious entity at a specific time; and (ii) identifying the node's causally related components.

To support proofs for any node in versioned provenance graphs, our accumulator builds on indexed Merkle trees [41, 57] for node storage and extends them to meet causality analysis requirements. Notably, a standard Merkle tree supports only single-keyword queries, whereas causality analysis requires two: *a system entity ID* and *a timestamp*. To address this, the accumulator organizes Merkle trees hierarchically. Locally, it employs multiple indexed Merkle trees, each storing version nodes for a single system entity and indexing them by creation timestamps, enabling timestamp-based proofs. Globally, it uses a Merkle tree to aggregate these local trees and indexes them by entity IDs, enabling entity-based proofs. By combining proofs from both levels, the accumulator provides complete proof for any node query.

Our verifiable versioned provenance graph provides cryptographic proofs for each node's causally related components—namely, the nodes and edges along its incoming and outgoing paths. This is achieved via a recursive hash mechanism that computes two digests for each node: an *incoming path digest* and an *outgoing path digest*, which capture the structure of the node's paths and jointly serve as proofs of its causally related components. Notably, as the graph should continuously incorporate new nodes to encode real-time system events, the outgoing path structures of many nodes may often change, necessitating frequent updates to their outgoing digests. To mitigate this overhead, we combine graph segmentation strategies with the recursive hash mechanism to construct update-efficient outgoing path digests (§ 5.4).

To enable *comprehensive validation* for causality analysis results, VCAUSE integrates two authenticated data structures in a coordinated manner: the verifiable versioned provenance graph encodes a continuous stream of system event logs, while the graph accumulator stores graph nodes, maintains the global graph digest, and tracks node updates. Consequently, by retrieving node proofs from the accumulator, we can verify the integrity of the initially queried node in causality analysis. Using the node's internal incoming and outgoing path digests, we can then validate its causally related components.

We formalize the security of VCAUSE and prove its resilience against an adaptive adversary that can modify any part of the causality analysis results. To demonstrate efficiency, we implement a VCAUSE prototype with 3,500 lines of C++ code and evaluate it on large-scale public log datasets [17,46]. Our results show that VCAUSE processes 25 million logs in 2 minutes and generates a proof for a 100,000-node causality analysis in 49 ms. We further assess runtime overhead in real-world settings by integrating VCAUSE with realistic endpoint loggers and co-deploying it with three common benchmarks under high workloads. Overall, VCAUSE can incur $< 1\%$ overhead on endpoints and 3.36% on the cloud.

Contributions. We make the following key contributions:

- We propose a verifiable causality analysis system for cloud-based endpoint auditing, enabling third-party verifiers to efficiently validate causality analysis results.
- We propose a versioned provenance graph that can encode causality relations in system events while providing proofs for each node's causally related components.
- We propose a graph accumulator based on hierarchical indexed Merkle trees, enabling proof generation for graph node queries. To support dynamic node updates, we further extend it with a dynamic indexed Merkle structure.
- We implement a VCAUSE prototype and evaluate its security and performance, showing that VCAUSE provides secure, verifiable causality analysis with low overhead.

2 Background and Related Work

2.1 Causality Analysis on Provenance Graphs

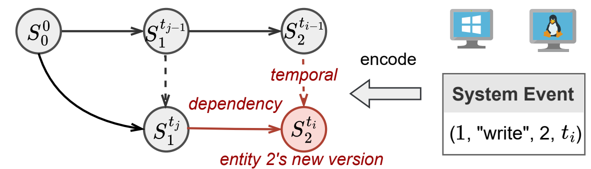


Figure 1: A versioned provenance graph

Provenance Graphs record system events and related causal relationships, where nodes represent system entities (*e.g.,* processes, files, sockets) and edges represent dependencies among entities (*e.g.,* read). They are widely used for causality analysis [3, 31] and attack detection [14, 26, 33]. However, conventional provenance graphs lack temporal ordering, limiting their ability to accurately reflect causality. To address this,

versioned provenance graphs [40] encode event sequences by creating a new version of an entity whenever its state changes. As shown in Figure 1, when event $(1, \text{"write"}, 2, t_i)$ occurs at timestamp t_i, a new node $S_2^{t_i}$ is created to represent the updated version of entity 2, along with a temporal and dependency edge that capture both temporal and causal relations.

Causality Analysis is proposed to analyze causality relationships on (versioned) provenance graphs. It was initially introduced by King et al. [37] to trace the root causes of suspicious system activities, and then extended to file system forensics and intrusion recovery [25, 42, 56]. Building on this foundation, subsequent research has proposed causality analysis variants [3, 4, 20, 27, 28, 30, 31, 39, 42, 45] to minimize unnecessary logs for manual analysis.

2.2 Related Integrity Validation Approaches

Log Integrity Validation. Prior work has employed trusted execution environments (TEEs) [35, 50, 54] (*e.g.,* CUSTOS) and cryptographic approaches [29] to provide tamper evidence for endpoint logs. However, these methods are ill-suited for validating causality analysis results in the cloud. Typically, extending them for such validation requires retrieving full endpoint logs, verifying their integrity, reconstructing the provenance graph, and re-running the analysis, which together incur substantial overhead. Alternatively, other studies have explored specialized hardware [1, 22] (*e.g.,* HARDLOG) to protect log integrity, but such solutions are costly and inherently lack support for cloud-side causality analysis.

Graph Integrity Validation. Prior work has proposed various graph integrity validation schemes [6, 52, 58]. While applicable to provenance graphs, they lack support for time-fuzzy node queries essential to causality analysis. Moreover, verifying the neighborhood of a node typically requires either costly upfront commitments (*e.g., signing and synchronizing all nodes*) or expensive validation of the entire graph. These schemes also offer limited support for graph updates.

Other Validation Approaches. Other related studies focus on verifiable and authenticated provenance storage [2, 32, 43, 53], but do not support causality analysis with verifiable proofs. Alternatively, generic SNARK schemes [8, 9, 13] could be leveraged to enable such analysis, but large-scale log data make the arithmetic circuits prohibitively large and expensive. Likewise, deploying TEEs [51] in the cloud may incur substantial runtime overhead, as processing massive log volumes within enclaves involves frequent I/O operations and enclave-to-nonenclave context switches.

3 System Overview

3.1 Threat Model

This work considers a cloud-based endpoint auditing infrastructure in enterprise networks, comprising a set of endpoints, a remote security administrator, and a cloud service. The administrator relies on the cloud to manage system-level event logs from endpoints and to perform causality analysis for attack investigation. According to standard threat models in cloud computing [61, 63], security risks in such settings often arise from the untrusted cloud. Sophisticated attackers who have infiltrated the infrastructure may tamper with raw logs or directly alter causality analysis results on the cloud, erasing traces of their malicious activities [24]. While existing log protection schemes [1, 29, 35, 50] can secure log collection, caching, and processing at endpoints, they are not inherently designed to protect logs or validate causality analysis results in the cloud. In this work, we build upon existing log protection schemes and focus on addressing the integrity issues of stored logs and causality analysis in the cloud.

Beyond external threats, the cloud itself may be untrustworthy in maintaining the integrity of logs and associated causality analysis results, due to misconfiguration or self-interest [12, 63]. For instance, when handling large volumes of event logs with limited configured resources, it may delete logs to free storage or return incomplete analysis results to conserve computational resources. To account for these risks, we assume the cloud may exhibit Byzantine faults [63], meaning it can: (i) manipulate system event logs by inserting, modifying, or deleting arbitrary data; and (ii) return incomplete, outdated, or falsified causality analysis results. Our work focuses on the integrity issues rather than confidentiality, as the latter is orthogonal to our goals and has been addressed by existing privacy-preserving graph search schemes [10, 60].

3.2 Problem Formulation

In this work, we build versioned provenance graphs from system logs to support verifiable causality analysis. We focus on providing validation for the generic form of causality analysis. Additional variants [20, 28, 39, 42, 45] that prune unnecessary analysis paths are also supported, as their outputs are subsumed by the general causality analysis and thus covered by our validation (see § 9). Following prior work [20, 21, 31], the causality analysis can be formulated as a subgraph query over a versioned provenance graph, as follows:

Definition 1 (Causality Analysis) *This process begins with a node query $N(s, \preceq t)$ to locate the point-of-interest version node n, whose system entity ID matches s and whose creation timestamp is the nearest one preceding or equal to t. Subsequently, backward and forward depth-first searches are performed to extract the backward and forward causally related components, i.e., $\{V_{\to n}, E_{\to n}\}$ and $\{V_{n\to}, E_{n\to}\}$:*

$$V_{\to n} = \{i \mid Path_{i,n} \neq \varnothing\}, E_{\to n} = \{i.E_{in} \mid Path_{i,n} \neq \varnothing\} \quad (1)$$

$$V_{n\to} = \{j \mid Path_{n,j} \neq \varnothing\}, E_{n\to} = \{j.E_{out} \mid Path_{n,j} \neq \varnothing\} \quad (2)$$

Here, $Path_{i,j}$ denotes paths between node i and j, and E_{in} and E_{out} refer to a node's incoming and outgoing edges.

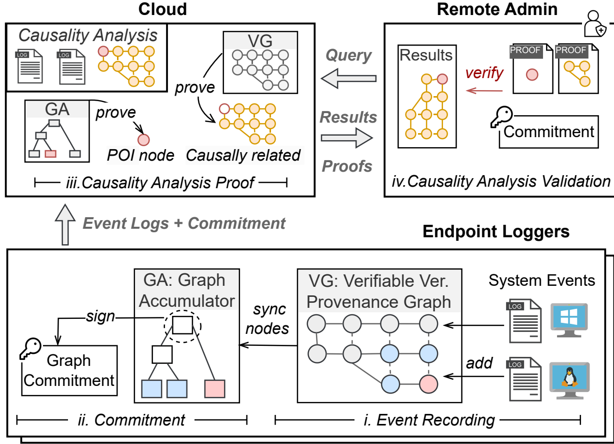


Figure 2: Architecture of vCAUSE

By definition, node n's forward and backward causally related components (*i.e.*, $\{V_{\to n}, E_{\to n}\}$ and $\{V_{n\to}, E_{n\to}\}$) are actually the nodes and edges along n's incoming and outgoing paths.

Causality Analysis Validation. Accordingly, verifying a causality analysis result should (i) check the correctness of the initially queried node, and (ii) validate the integrity of forward and backward causally related nodes and edges.

3.3 System Model and Workflow

Figure 2 shows the architecture of vCAUSE, an efficient and verifiable causality analysis system for cloud-based endpoint auditing. Overall, vCAUSE interacts with three parties: (i) *endpoint loggers*, which upload real-time system event logs to the cloud; (ii) the *cloud*, which manages endpoint logs and provides a verifiable causality analysis service; (iii) *a remote security administrator*, who performs causality analysis to investigate suspicious system activities.

To achieve causality analysis validation, vCAUSE adopts two authenticated data structures: a *graph accumulator* and a *verifiable versioned provenance graph structure*. The structures respectively provide proofs for the initially queried point-of-interest (POI) node and its causally related components.

Graph Accumulator. To provide proofs for node queries on a versioned provenance graph, a naive solution is to use an indexed Merkle tree [41, 57] for node storage. However, a single tree only supports single-keyword queries, while causality analysis typically requires two: a system entity ID and a timestamp. Our accumulator addresses this with a *hierarchical Merkle tree* design. Locally, each system entity is assigned an indexed Merkle tree that stores its versioned nodes by timestamp, enabling timestamp-based proofs. Globally, a Merkle tree aggregates these local trees by entity ID, enabling entity-based proofs. By combining local and global proofs, the accumulator produces complete node-query proofs. To support graph evolution, we also design a dynamic Merkle tree structure that supports efficient node insertions.

Verifiable Versioned Provenance Graph. The graph structure extends the original provenance graph, capturing causality relations in system events while also providing verifiable proofs for a node's causally related components. According to Equation 1, a node's causally related components are located in its incoming and outgoing paths. To enable their validation, we employ a recursive hash mechanism that computes two digests for each node: the incoming and outgoing path digests. These digests capture the structural information of a node's incoming/outgoing paths and jointly serve as proofs of its causally related components. Notably, as the graph continuously incorporates new nodes to encode real-time events, the outgoing path structures of many existing nodes may change frequently, necessitating frequent updates to their outgoing path digests. To mitigate this overhead, we adopt *segmented outgoing path digests*, generated based on segmentation strategies that partition the graph into smaller dependency trees. This ensures that adding a new node affects only the nodes within a single tree, significantly reducing the number of required digest updates.

Based on the structures, vCAUSE operates in four phases:

1) Event Recording: Each endpoint logger continuously captures system events and updates the corresponding verifiable versioned provenance graph to encode these events.

2) Commitment: Each endpoint logger periodically synchronizes node changes during event recording into the Merkle tree of the graph accumulator, signs the tree root as a graph commitment, and sends it along with event logs to the cloud.

3) Causality Analysis Proof: The cloud uses event logs from each endpoint to reconstruct graph accumulators and verifiable versioned provenance graphs. Upon receiving a request from the administrator, it performs causality analysis and generates corresponding proofs using these structures.

4) Causality Analysis Validation: By retrieving the graph commitment and node proofs from the cloud, the administrator first verifies the integrity of the initially queried node. Then, using the node's incoming and outgoing path digests, the administrator validates its causally related components.

4 Graph Accumulator

This section introduces a graph accumulator that provides single-node proofs on versioned provenance graphs. A variant supporting range proofs is detailed in Appendix A.2.

4.1 Hierarchical Tree Accumulation Structure

Our graph accumulator applies indexed Merkle trees for node storage and extends them into a hierarchical structure to support node queries in causality analysis, which involve two keywords: system entity ID and timestamp (Definition 1). The hierarchical structure is shown in Figure 3. Locally, multiple Merkle trees are used to accumulate version nodes for

each system entity separately. These trees store the nodes at leaf positions and index them by timestamps, enabling proofs for timestamp-based queries. Globally, a single Merkle tree aggregates all local trees. It stores the roots of the local trees at leaf positions and indexes them by entity IDs, providing proofs for entity-based queries. By combining the proofs from both levels, our accumulator can produce complete proofs for any node queries. Notably, the root of the global tree serves as a cryptographic commitment to the entire graph, allowing anyone with the proofs to validate the results of node queries.

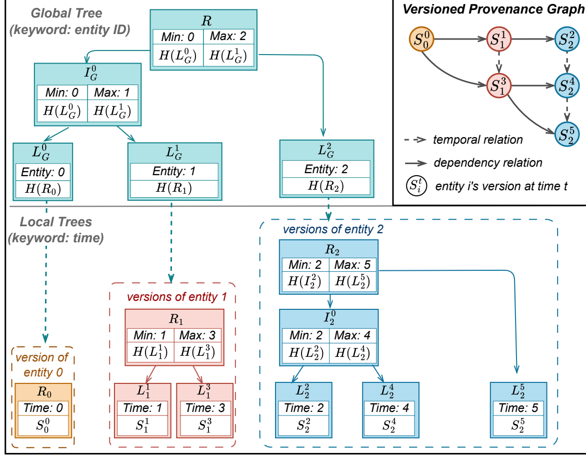


Figure 3: Graph accumulator. Here, $H(\cdot)$ is a hash function.

In the tree structure, each internal node includes index information that records the minimum and maximum keywords in its subtree, enabling efficient binary search from the root to locate the queried node and generate the corresponding proof.

4.2 Single-Node Proof and Validation

Node queries in causality analysis are *fuzzy*, rather than based on exact keyword matching. In particular, the timestamp of the queried node need not exactly match a given value t; instead, it only needs to be the closest to t (Definition 1). Our accumulator supports such time-fuzzy node queries in the form $N(s,r)$, where s denotes the queried system entity ID and r specifies a temporal relation. We consider two types of temporal relation queries: (i) $\succeq t$, which returns the nearest timestamp greater than or equal to t, and (ii) $\preceq t$, which returns the nearest timestamp less than or equal to t.

We now present the membership and non-membership proofs provided by our accumulator, which respectively confirm the existence and absence of a node matching a query.

Membership Proofs. For a node query $N(s,r)$, the membership proof is generated through a two-step search process in hierarchical Merkle trees. Specifically, it is structured as a tuple (ρ_G,ρ_L), where the proof ρ_G is generated by searching the global tree for the entity keyword s, and ρ_L is generated by searching the corresponding local tree according to the temporal relation r. The full algorithm is detailed in Appendix A.1. Both ρ_G and ρ_L contain nodes along their respective search paths, enabling the verifier to reconstruct the paths and validate if the queried node satisfies the query.

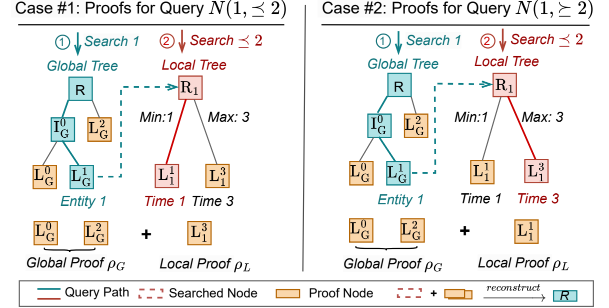


Figure 4: Single-node membership proof

Figure 4 shows the membership proof (ρ_G,ρ_L) for query $Q(1,\preceq 2)$ (Case #1). To generate ρ_G, an exact search is conducted in the global tree for the entity keyword 1. The sibling nodes encountered along this search path constitute the proof ρ_G. Then, a fuzzy search is conducted in the corresponding local tree for the relation $\preceq 2$, and the sibling nodes encountered form the local proof ρ_L. The figure also illustrates the proofs for query $Q(1,\succeq 2)$. Although the keywords are the same as the previous query, the different temporal relation yields a different matched node and local proof ρ_L.

Non-membership Proofs. Our accumulator also supports the non-membership proof for a node query, demonstrating that no nodes satisfy the query and preventing the cloud from falsely returning empty results. Similar to membership proof, it is generated through a two-step search process in hierarchical Merkle trees. Details are provided in Appendix A.1.

Validation. To verify (non-)membership for a node query $N(s,r)$, the verifier first reconstructs the original search path and corresponding tree structure using the provided proofs. If the root of the reconstructed tree matches the previously committed root, the proof is considered valid. The verifier then examines the index information of the reconstructed tree to validate node (non-)membership. For membership validation, if the queried node is located within the local tree of entity s and satisfies the queried temporal relation r, membership is confirmed. For non-membership validation, if no such node is found, non-membership is confirmed.

4.3 Dynamic Indexed Merkle Tree

To extend our accumulator to support node insertions and updates, we develop a dynamic indexed Merkle tree structure, denoted as DIM-Tree. Compared to existing dynamic Merkle trees (e.g., Merkle B+ tree [57]), our DIM-Tree offers more efficient node insertion operations, achieving constant amortized computational cost. This efficiency is beneficial for

system-wide versioned provenance graphs, where nodes are continuously created to encode system events.

Tree Structure. DIM-Tree is designed to efficiently store an unbounded stream of graph nodes. To facilitate system entity-based or timestamp-based queries, DIM-Tree indexes the nodes by entity or timestamp keywords. As the keywords typically increase over time, node insertions are arranged in temporal order, *i.e.,* each new node is appended to the current leaf node sequence. Notably, each insertion alters the tree structure and may require updating the hashes of numerous affected nodes. To improve efficiency, our DIM-Tree adopts a multi-subtree design composed of multiple *perfect binary subtrees*, as shown in Figure 5. This design ensures that inserting a node only involves merging it with partial subtrees and updating the necessary internal hashes.

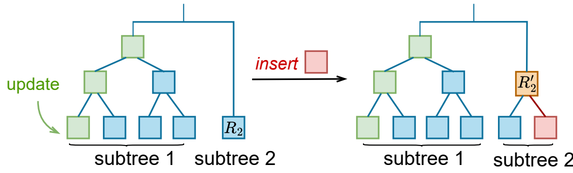


Figure 5: Structure of DIM-Tree. Node insertion triggers a merge with the last subtree, forming a larger subtree of size 2.

Insertion. Node insertion in DIM-Tree is essentially a subtree merge process. Each new node is treated as a height-1 subtree and recursively merged with preceding subtrees from right to left, forming a larger perfect binary subtree. The detailed algorithm is provided in Appendix B. Compared to conventional Merkle trees, DIM-Tree incurs lower insertion overhead, as only the hashes of affected internal nodes in partial subtrees need updating. Specifically, the amortized time complexity per insertion is $O(1)$. For instance, inserting 2^N nodes requires only 2^N subtree merges and hash updates, resulting in a near-constant average insertion time.

Update. When a node changes, the hashes of internal nodes along the path to the root of the corresponding subtree are recomputed. The update cost is $O(\log N_{sb})$, similar to the $O(\log N)$ in standard Merkle trees, but more efficient in practice, as the subtree size N_{sb} is smaller than the total size N.

After a series of insertions and updates, the entire DIM-Tree should be *finalized* by recursively merging the subtree roots from right to left to produce the overall tree root. When multiple DIM-Trees are organized hierarchically (*e.g.,* in a graph accumulator), local trees should be finalized first, followed by the global tree, to maintain consistency and efficiency.

5 Verifiable Versioned Provenance Graph

5.1 Incoming Path Digests

According to Definition 1, a node's backward causally related components lie along its incoming paths. To generate proofs for them, we employ an *incoming path hash mechanism* to compute an incoming path digest Π_I per node, representing the digest of all components in its incoming paths.

Incoming Path Hash. The hash mechanism begins by computing incoming path digests from entry nodes. As an entry node has no incoming edges, its incoming path digest Π_I is initialized as $H(\varnothing)$, where $H(\cdot)$ denotes a hash function over a set. The incoming path digests of subsequent nodes are then derived based on their predecessors. Specifically, for node n, its incoming path digest is defined as follows:

$$n.\Pi_I = H(\{(\underbrace{s(e)}_{\text{in-edge}} \,||\, \underbrace{e.src.\Pi_I}_{\text{a predecessor node's } \Pi_I}) \mid e \in n.E_{in}\}) \quad (3)$$

where $n.E_{in}$ denotes the incoming edges of n, and $s(e)$ is the string representation of an edge, formed by concatenating the associated nodes' fields `[id || time || ...]` and event data `[event_type || ...]`. The symbol $||$ denotes concatenation.

Dynamic Generation of Π_I. In a versioned provenance graph, nodes and edges are created continuously to record system events (see § 2.1). Upon the creation of a new node, its incoming path digest Π_I should be dynamically computed based on the previously connected nodes and edges, as defined in Equation 3. Once generated, the digest remains unchanged. This immutability arises from the properties of versioned provenance graphs: each node represents a version of an entity, and its incoming edges capture its creation histories. Since this history is immutable, the node cannot receive new incoming edges, and its incoming path digest remains unchanged.

5.2 Backward Causality Relation Validation

Proof. According to Equation 3, a node's incoming path digest is computed via a recursive hash mechanism, capturing all components along its incoming paths, *i.e.,* backward causally related components. Thus, the incoming path digest of the node and its backward-connected components can serve as a cumulative proof for its backward causality relations.

Validation. To verify a node's backward causality relations, we recursively hash the components along its incoming paths from entry nodes to *regenerate* an incoming path digest. If the regenerated digest matches the node's original one, the integrity of the node's backward causality relations is confirmed. The detailed algorithm is described in Algorithm 4.

5.3 Outgoing Path Digests

A node's forward causally related components lie along its outgoing paths. To generate proofs for them, we employ an *outgoing path hash mechanism* that computes an outgoing path digest Π_O per node, capturing all related components.

Outgoing Path Hash. The hash mechanism operates inversely to incoming path hash: it starts from exit nodes (*i.e.,* nodes without outgoing edges). For an exit node, its outgoing

path digest Π_O is computed as $H(\varnothing)$, where $H(\cdot)$ represents a hash function. Tracing backward, the digest of each preceding node is derived from the digests of its successors. Specifically, for node n, its outgoing path digest is defined as follows:

$$n.\Pi_O = H(\{(\underbrace{s(e)}_{\text{out-edge}} \;||\; \underbrace{e.dst.\Pi_O}_{\text{a successor node's }\Pi_O}) \mid e \in n.E_{out}\}) \quad (4)$$

where $n.E_{out}$ denotes n's outgoing edges, and $s(e)$ is the string representation of edge e formed by concatenating related node fields and event data. The symbol $||$ denotes concatenation.

Notably, a node's Π_O changes when new outgoing edges are added. To update efficiently, we adopt ordering-invariant incremental hashing [7] over a set, enabling homomorphic addition (or subtraction) of component digests to the node's Π_O without recomputing the entire digest.

Dynamic Generation of Π_O. In a versioned provenance graph, nodes are continuously added to encode system events. Hence, a node's outgoing path digest Π_O should be generated dynamically as events occur. Since a newly created node has no outgoing edges, its Π_O is initialized as $H(\varnothing)$.

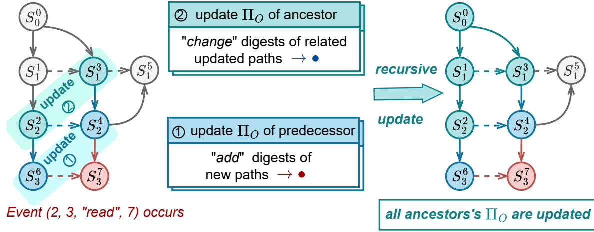


Figure 6: Updating outgoing path digests upon event occurrence. Here, Π_O denotes outgoing path digests

Dynamic Update of Π_O. Along with node creation, the graph also generates associated temporal and dependency edges. These components alter the outgoing path structures of connected ancestors, necessitating updates to their Π_O. As shown in Figure 6, event $(2, 3, \text{"read"}, 7)$ triggers the creation of node S_3^7 along with edges e_7^t and e_7^4. These components affect the outgoing path structures of nodes $\{S_0^0, S_1^1, S_1^3, S_2^2, S_2^4, S_3^6\}$, requiring corresponding updates to their Π_O.

The update process is recursive, beginning with the immediate predecessors of the newly created node.

1) Update Π_O of Immediate Predecessors: The addition of S_i^j and its associated edges expands the outgoing path structures of their predecessors. To synchronize the changes, we *add* the digest of the new elements to each predecessor p's original Π_O using an incremental hash function, as follows:

$$p.\Pi_O \oplus = H(s(e_{p \to S_i^j}) \;||\; S_i^j.\Pi_O), \text{when } S_i^j \text{ is added.} \quad (5)$$

Here, $e_{p \to S_i^j}$ denotes the edge from p to S_i^j, $s(e_{p \to S_i^j})$ represents the string representation of the edge and associated nodes, and $\oplus$ signifies homomorphic hash addition.

2) Update Π_O of Earlier Ancestors: Updates to the immediate predecessors' Π_O recursively propagate to earlier ancestors. Specifically, when a node n is updated, Π_O of its predecessors should be updated accordingly. Since n typically affects only part of its predecessors' outgoing path structures rather than the entire structure, we employ an incremental hash function [7] to update the affected digests efficiently.

The update for each predecessor p of node n is defined as:

$$\begin{aligned} p.\Pi_O \; & \underbrace{\ominus = H(s(e_{p \to n}) \;||\; n.\Pi_O)}_{\text{remove old path digest related to } n} \\ & \oplus \underbrace{H(s(e_{p \to n}) \;||\; n.\Pi_O^*)}_{\text{add new path digest related to } n}, \text{when } n \text{ is modified.} \end{aligned} \quad (6)$$

Here, $e_{p \to n}$ is the edge from p to n, $s(e_{p \to n})$ denotes the string representation of the edge and connected nodes, $n.\Pi_O$ and $n.\Pi_O^*$ denotes n's old and new outgoing path digests, and $\ominus$ and $\oplus$ denote homomorphic hash subtraction and addition.

Challenge of Exponential Digest Updates. In a versioned provenance graph, each node connects to two predecessor nodes via temporal and dependency edges. Thus, if paths of length L exist, adding a node may affect the outgoing path structures of 2^L ancestors, thereby resulting in exponential digest update overhead. To address this, we adopt the following update-efficient outgoing path digests.

5.4 Segmented Outgoing Path Digests

To reduce the update overhead of outgoing path digests, we apply the following segmentation strategies to *split* specific paths at first. The strategies reduce the number and length of paths leading to a node, ensuring that node insertion affects fewer ancestors, thereby reducing digest update overhead.

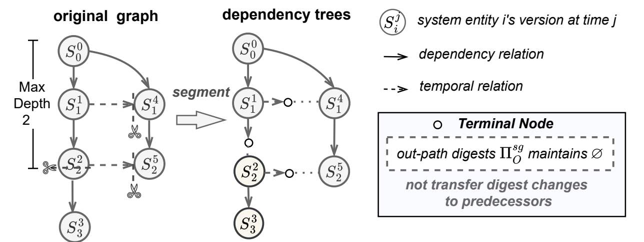


Figure 7: Segmentation strategies

1) Temporal Edge Segmentation: This strategy detaches each temporal edge from its destination node. The original destination is replaced with a terminal node. The terminal node has no outgoing edges and never receives new ones, so its outgoing path digests remain empty. To preserve logical connectivity, it maintains a pointer to the original destination. As shown in Figure 7, this segmentation converts the provenance graph into multiple trees, where each node has only one incoming dependency edge. Consequently, when a node is

created, only its ancestors along a single path require updates, reducing the update overhead to linear complexity.

2) Deep Tree Segmentation: While the above strategy reduces digest update complexity by forming smaller trees, updates still become time-consuming if trees grow too deep. To address this, we define a segmentation depth L. If a dependency edge causes the dependency depth of a tree to exceed L, the edge and connected nodes are moved into a new tree.

After segmentation, each node's outgoing path digest over the segmented paths (Π_O^{sg}) can be computed using Equation 4. Similar to the original Π_O, a node's Π_O^{sg} is initialized as $H(\varnothing)$ when a new event triggers its creation.

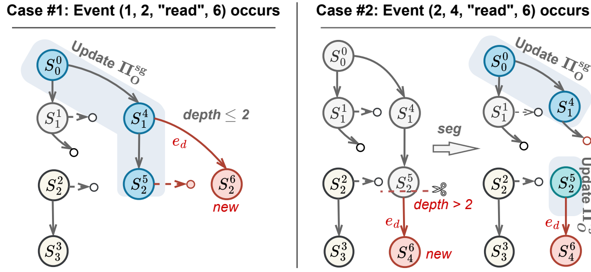


Figure 8: Updating segmented outgoing path digests Π_O^{sg} upon event occurrence. Here, the tree segmentation depth is 2.

Dynamic Update of Π_O^{sg}. As with the original outgoing path digests, Π_O^{sg} should be dynamically updated when associated dependency and temporal edges are added to the graph. However, the update process is more complex, as edge additions may trigger different structural changes and corresponding update procedures. According to segmentation strategies, the temporal edge does not increase the tree's dependency depth and thus remains within the original tree. Thus, affected ancestors' Π_O^{sg} can be directly updated using Equations 5 and 6. In contrast, the dependency edge increases the dependency depth of the tree. As shown in Figure 8, this may result in two structural changes, each requiring a distinct update:

1) Case #1. If the dependency edge e^d does not increase the tree depth beyond the threshold L, the edge and its connected nodes remain in the tree. In this case, Π_O^{sg} of connected ancestors can be updated using Equations 5 and 6.

2) Case #2. If the dependency edge e^d increases the tree depth beyond the threshold L, the edge is moved to a new tree. The connected nodes S_2^5 and S_4^6 are also transferred to the new tree, and the original position of S_2^5 in the old tree is replaced with a terminal node. Structural changes in the old and new trees affect different nodes. In the original tree, the added terminal node influences its ancestors S_0^0 and S_1^4, while in the new tree, the new edge e^d affects only root S_2^5. To accommodate the changes, we update Π_O^{sg} of the affected nodes in the old and new trees using Equations 5 and 6, respectively.

In summary, compared to original outgoing path digests with exponential update overhead, the use of segmented outgoing path digests reduces this overhead to linear complexity $O(L)$, where L is the predefined tree segmentation depth.

5.5 Forward Causality Relation Validation

Proof. A node's outgoing path digest Π_O represents the hash of all components along its outgoing paths (*i.e.,* forward causally related components). Thus, Π_O of a node and its forward-connected nodes can serve as a cumulative proof of its forward causality relations. In practice, however, the proof process may be more complex, as segmented outgoing path digests Π_O^{sg} are often used instead of Π_O for efficiency.

A node's segmented outgoing path digest Π_O^{sg} captures only the components within a single dependency tree and does not encompass all forward causally connected components. Therefore, utilizing the node's Π_O^{sg} is insufficient for validating its complete causality relations. To ensure full validation, Π_O^{sg} of roots from subsequent connected trees should be utilized to validate the remaining components. Notably, to confirm the authenticity of the Π_O^{sg}, node proofs of the roots should also be retrieved from the graph accumulator.

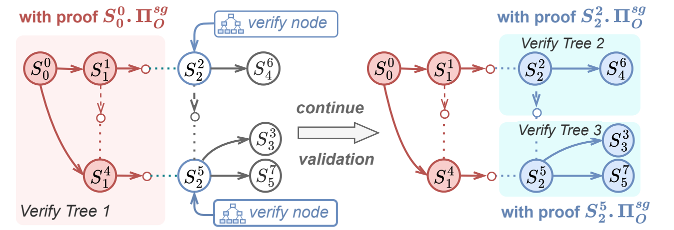


Figure 9: Forward relation validation with Π_O^{sg}

Validation. Figure 9 illustrates the validation process using segmented outgoing path digests Π_O^{sg}. The process begins by using the initial node's Π_O^{sg} to validate forward causally related components within the dependency tree 1. If successful, the roots of subsequent trees (*i.e.,* S_2^2 and S_2^5) are then validated using node proofs from the graph accumulator. To enhance the efficiency of this process, temporal range proofs can be applied to enable batch validation for these roots (see Appendix A.2). Once the roots are validated, their Π_O^{sg} are then used to verify the remaining connected components. The detailed validation algorithm is given in Algorithm 5.

6 Verifiable Causality Analysis System

By integrating our graph accumulators and versioned provenance graphs, vCause provides complete proofs for causality analysis. The system involves three parties—endpoint loggers, a cloud, and a security administrator—and supports four core operations: *event recording*, *commitment*, *causality analysis proof*, and *validation*, as described below:

Event Recording. When a system event occurs, the endpoint logger updates its local verifiable versioned provenance graph by creating a new node representing the current system entity

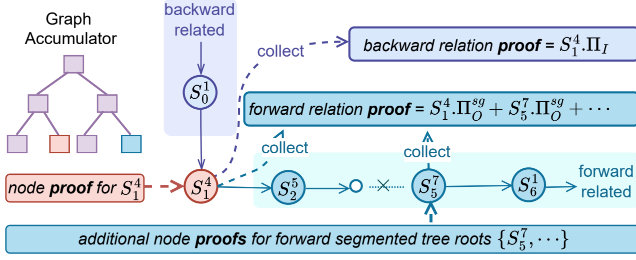


Figure 10: Proof for a causality analysis query. The query targets entity 1 at timestamp t_4. S_1^4 is the corresponding node; Π_I and Π_O^{sg} denote incoming/segmented outgoing path digests.

version, along with its associated temporal and dependency edges. It then computes the node's incoming and (segmented) outgoing path digests and updates the digests of affected ancestors (see § 5.1 and § 5.4).

Commitment. To enable timely validation, each endpoint logger periodically commits node changes during event recording into the Merkle trees of a graph accumulator (§ 4.3). It then signs the tree root R along with the current timestamp t to produce a fresh signature σ_R. The tuple (R,t,σ_R) serves as the cryptographic commitment to the entire graph and is sent to the cloud along with the corresponding event logs.

Causality Analysis Proof. To generate proofs for causality analysis queries, the cloud reconstructs a verifiable versioned provenance graph and a graph accumulator from the logs of each endpoint. As shown in Figure 10, upon receiving a query for an endpoint's system entity at a given timestamp, the cloud first locates the corresponding graph node (*i.e.,* S_1^4) and its causally related components. It then uses the accumulator to generate a proof for the node (§ 4.2) and collects its incoming and outgoing path digests as evidence of backward and forward causal relationships (§ 5.2 and § 5.5). If segmented outgoing digests are applied, the forward causally related components may be segmented accordingly. For completeness, the cloud also needs to retrieve node proofs of the corresponding segmented tree roots from the accumulator.

Causality Analysis Validation. To validate causality analysis results, the administrator retrieves the latest cryptographic graph commitment from the cloud and verifies its signature. If valid, the administrator validates the queried node using its node proof (§ 4.2), and then validates backward and forward causally related components with the node's incoming and (segmented) outgoing path digests (§ 5.2 and § 5.5).

7 Security Analysis

We follow the notion of existential unforgeability under chosen message attacks (EUF-CMA) [18] to define the security of vCAUSE against an adaptive adversary $\mathcal{A}$ that can add, delete, or modify components in causality analysis results. The property is formalized via the following game $Forge_{\mathcal{A}}^{\text{vCAUSE}}(k)$:

1) Causality Analysis Challenges. $\mathcal{A}$ makes a polynomial number of queries Q to the causality analysis oracle. For each query $q \in Q$, the oracle identifies a point-of-interest node n and returns its backward and forward causally related components: $\{V_{\to n}, E_{\to n}\}$ and $\{V_{n\to}, E_{n\to}\}$. Additionally, it provides the corresponding commitment (R,t,σ_R) and the node proof ρ_n for n. The internal incoming and outgoing path digests of n are treated as its causality analysis proofs.

2) Forgery. $\mathcal{A}$ outputs a forged causality analysis result $C^* = \{n^*, \{V^*_{\to n}, E^*_{\to n}\}, \{V^*_{n\to}, E^*_{n\to}\}\}$ for a query $q^* \notin Q$, along with a forged proof ρ_n^* and a commitment (R^*, t^*, σ_R^*).

3) Verification. $\mathcal{A}$ submits the forged results C^*, proof ρ_n^*, and commitment (R^*, t^*, σ_R^*). The challenger first validates the signature of the commitment. It then checks whether C^* and ρ_n^* are valid with respect to q^*. If all verifications pass, the adversary wins the game and outputs 1.

Definition 2 (Unforgeability of Causality Analysis Results) We say vCAUSE achieves unforgeability of causality analysis results if, for any polynomial-time adversary $\mathcal{A}$:

$$\Pr(Forge_{\mathcal{A}}^{\text{vCAUSE}}(k) = 1) \leq \text{negl}(k) \tag{7}$$

where $\text{negl}(k)$ denotes a negligible function.

Now, we give the following security theorem. (formal security theorem and proof can be found in Appendix D)

Theorem 1 vCAUSE *achieves unforgeability of causality analysis results if the signature scheme is EUF-CMA-secure [18], the Merkle tree structure is position-binding [11], and the verifiable versioned provenance graph upholds causality relation unforgeability (Appendix D.1).*

8 Experimental Evaluation

Implementation. We implement a prototype of vCAUSE in 3,500 lines of C++ code using OpenSSL 1.1.1w [49]. The graph accumulator adopts a hierarchical indexed Merkle tree built with SHA3-256 hash function, while digests in the versioned provenance graph leverage an incremental hash function [7]. To generate a cryptographic commitment for the graph, the Merkle root is signed using ECDSA [34].

Table 1: Summary of datasets. $|V|$ and $|E|$ denote the number of versioned nodes and edges in versioned provenance graphs built from a dataset. SS. denotes the StreamSpot dataset.

Dataset	# of Endpoints	Avg # of Events	Avg |V|	Avg |E|
SS.	600	149K	149K	290K
OpTC	1,000	17M	17M	32M

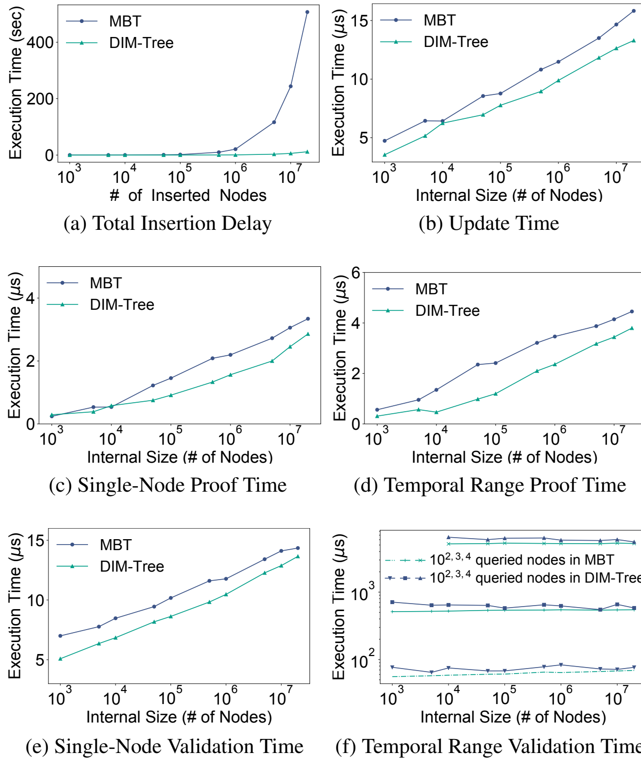
Datasets. We evaluate vCAUSE on two real-world endpoint auditing datasets: DARPA OpTC [17] and StreamSpot [46],

both containing large volumes of realistic system event logs. Table 1 summarizes their key statistics. The StreamSpot dataset contains system events from six simulated web browsing scenarios, each executed 100 times. On average, each execution yields thousands of event logs. In our experiments, we treat these 600 executions as logs from 600 endpoints. The OpTC dataset, produced by DARPA's Transparent Computing (TC) program, includes over 17 billion events collected from 1,000 real hosts over six days in an enterprise network.

Experiments are performed on a Ubuntu 22.04 PC with 4 Intel Xeon 3.60GHz processors and 32GB RAM. The evaluation covers (i) efficiency of proposed authenticated data structures (§ 8.1 and § 8.2); (ii) VCAUSE's computational costs (§ 8.3); (iii) runtime overhead under realistic deployment settings (§ 8.4); (iv) communication and storage costs (§ 8.5).

8.1 Efficiency of Graph Accumulators

Recall that the computational cost of our accumulators mainly depends on the number of accumulated nodes. To evaluate its efficiency, we use data from an endpoint in the larger dataset (OpTC), yielding a versioned provenance graph with 25 million nodes. We then measure the execution time of four operations: insertion, update, proof generation, and validation.



(a) Total Insertion Delay

(b) Update Time

(c) Single-Node Proof Time

(d) Temporal Range Proof Time

(e) Single-Node Validation Time

(f) Temporal Range Validation Time

Figure 11: Performance of graph accumulators. Here, MPT refers to the Merkle B+ tree-based accumulator, while DIM-Tree denotes our DIM-Tree-based accumulator.

Notably, our accumulator adopts a tailored dynamic Merkle tree, DIM-Tree, which supports efficient node insertion with $O(1)$ amortized time complexity. To evaluate its efficiency, we implement a baseline accumulator using a conventional dynamic Merkle tree (Merkle B+ tree) and compare both designs using the same 25-million-node dataset.

Node Insertion. Figure 11.(a) presents the insertion time for different node counts. Compared to the Merkle B+ tree-based design, our DIM-Tree-based accumulator demonstrates significantly faster performance. Its insertion time grows linearly with the number of nodes, confirming an average constant cost per insertion. In total, inserting 25 million nodes takes ≈ 14 seconds, with each insertion costing about 0.57 μs.

Node Update. The time of updating a node in the accumulator depends on the number of internal nodes. To assess this, we measure the update latency across different internal sizes. As shown in Figure 11.(b), the update time of both accumulators increases with the number of internal nodes. Particularly, our DIM-Tree-based accumulator achieves slightly lower update time than the Merkle B+ tree-based accumulator.

Node Proof. To comprehensively evaluate the proof time for both single-node and temporal range queries, we vary the number of stored nodes in each accumulator and generate proofs for 1,000 single-node and temporal range queries. Figures 11.(c) and 11.(d) show the average proof time for both query types. We observe that the proof time increases with the number of internal nodes.

Node Validation. We evaluate the validation time for both single-node and temporal range queries. Recall that validation time depends on both the size of the original tree and the number of queried nodes (see § 4.2 and Appendix A.2). To comprehensively assess this cost, we vary the internal size of each accumulator, generate proofs for 3,000 single-node and temporal range queries, and perform validation. Particularly, temporal range queries are grouped into three scales: 10^3, 10^4, and 10^5 successive nodes within a given timestamp range. Figures 11.(e) and 11.(f) show the average validation time for single-node and temporal range queries, respectively. The results indicate that single-node validation time primarily increases with the number of internal nodes, while temporal range validation time scales with the number of queried nodes.

8.2 Performance of Verifiable Versioned Provenance Graphs

We evaluate our verifiable versioned provenance graphs by constructing two graph instances from logs in StreamSpot and OpTC, containing 716K and 25M nodes, respectively. For both graphs, we measure the generation and update time of incoming and outgoing path digests during construction.

Digest Generation. A node's incoming and outgoing path digests are generated immediately upon its creation. Specifically, the outgoing path digest is always initialized to $H(\varnothing)$, while the incoming path digest is computed based solely on

Table 2: Average generation time of path digests

Operation	StreamSpot	OpTC
Incoming path digest generation	1.59 μs	1.64 μs
Outgoing path digest generation	1.34 μs	1.37 μs

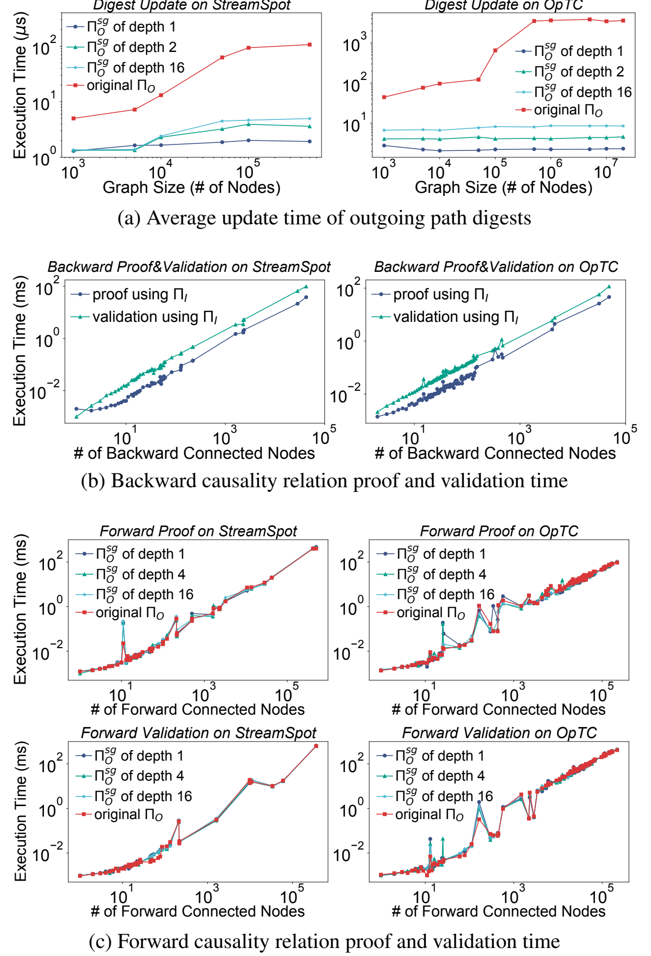
the node's intermediate predecessors. Thus, their generation time is independent of the graph structure. Results from the StreamSpot and OpTC datasets confirm this. As shown in Table 2, the average generation time for both digests remains consistent across datasets, ranging from 1 to 2 μs.

Digest Update. We do not consider the update of incoming path digests as they remain unchanged once generated due to properties of versioned provenance graphs (see § 5.1). Our experiments focus on measuring the update time of outgoing path digests (Π_O). According to Equations 5 and 6, the update process is triggered by node insertion. The insertion changes the outgoing paths of connected ancestors, necessitating updates to their Π_O. Thus, the overall update time grows with the graph size, since larger graphs tend to have more affected ancestors. To assess this impact, we generate graphs of varying sizes from StreamSpot and OpTC datasets. Figure 12.(a) shows the average update time of Π_O across these graphs, revealing a substantial increase as graph size grows.

Additionally, we evaluate the update time of segmented outgoing path digests (Π_O^{sg}), which are optimized variants of the original Π_O. These digests are generated based on graph segmentation strategies, reducing the update time to linear complexity $O(L)$, where L is the predefined segmentation depth (§ 5.4). To assess the impact of L, we vary its value and measure the corresponding average update time for Π_O^{sg}. The results, shown in Figure 12.(a), indicate that Π_O^{sg} consistently achieves significantly lower update time than the original Π_O across all tested values of L. Moreover, the update time is nearly independent of graph size and primarily determined by L: smaller values of L lead to lower update time costs.

Causality Relation Proof. We evaluate the execution time of causality relation proof operations based on the incoming and outgoing path digests (Π_I and Π_O). To assess the time across different graph structures, we construct two graph instances using data from OpTC and StreamSpot. For each graph, we generate forward and backward causality relation proofs for 1,000 randomly selected nodes and record the proof time. The results, shown in Figures 12.(b) and 12.(c), show that the proof time for both types of proofs scales proportionally with the number of connected nodes.

Notably, forward causality proofs can also be derived from segmented outgoing path digests (Π_O^{sg}). We therefore additionally measure the corresponding proof time. Recall that each node's Π_O^{sg} captures only components within a segmented tree and does not extend to subsequent trees. Thus, generating a complete proof may require retrieving node proofs of segmented tree roots from the graph accumulator (§ 5.5), potentially making the process slower than using Π_O directly. Typically, the proof time is affected by the predefined tree segmentation depth L. To evaluate this effect, we vary L and record the corresponding proof time, as shown in Figure 12.(c). Interestingly, the proof time using Π_O^{sg} is comparable to that of Π_O, and remains nearly independent of the segmentation depth. This is likely because the overhead of retrieving node proofs constitutes only a small fraction of the total time.



(a) Average update time of outgoing path digests

(b) Backward causality relation proof and validation time

(c) Forward causality relation proof and validation time

Figure 12: Performance of verifiable versioned provenance graphs

Causality Relation Validation. Based on the generated proofs, we validate forward and backward causality relations and record the corresponding validation time. As shown in Figures 12.(b) and 12.(c), the validation time is proportional to the number of validated nodes. Moreover, it remains similar regardless of whether the proofs are generated from segmented or unsegmented outgoing path digests (Π_O^{sg} and Π_O).

Remark. Figures 12.(a)–(c) show that segmented outgoing path digests substantially reduce update time while preserving similar proof and validation time to unsegmented ones. A segmentation depth of 1 offers the best trade-off, minimizing

update time with negligible overhead in proof and validation.

8.3 Computational Cost of VCAUSE

Based on prior experimental results, we evaluate the computational cost of the entire VCAUSE workflow. We configure the underlying verifiable graphs with the optimal setup—using segmented outgoing path digests and a segmentation depth of 1. To assess VCAUSE in practice, we set up two endpoint auditing environments using data from 600 and 1,000 distinct endpoints in StreamSpot and OpTC. In these environments, we measure key VCAUSE operations: event addition, commitment, causality proof generation, and validation.

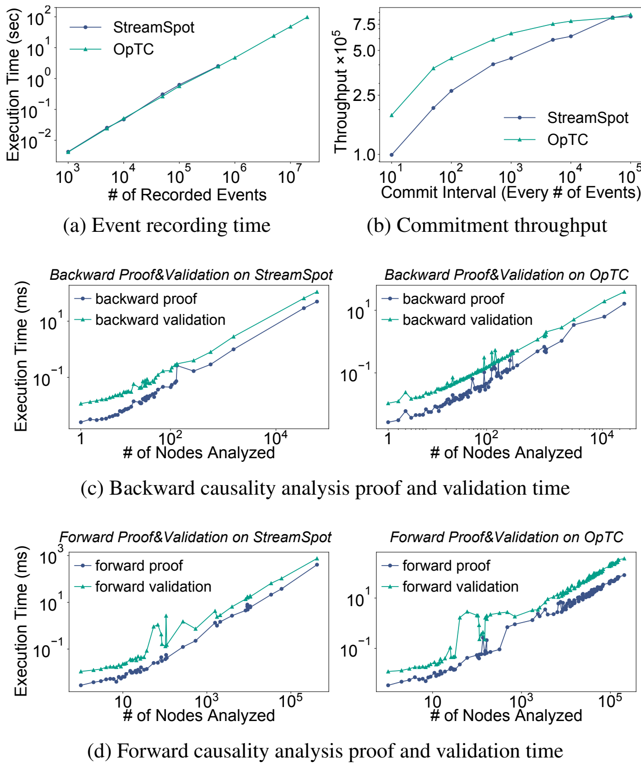


Figure 13: Computational cost of VCAUSE

Event Recording. Figure 13.(a) shows the average event recording time per endpoint. The results indicate that the recording time scales linearly with the number of recorded events. Additionally, it remains similar across datasets: for both StreamSpot and OpTC, the per-event recording time ranges from 4 to 6 μs. Notably, compared to the entire six-day duration of the OpTC dataset, VCAUSE records 25 million events in just 119 seconds—introducing negligible overhead.

Commitment. Recall that each endpoint periodically uses the graph accumulator to commit node changes during event recording (see § 6), enabling timely validation. For endpoints with high event volumes, the commitment frequency affects throughput: overly frequent commitments may redundantly process repeated changes to the same nodes, degrading performance, whereas an appropriate interval allows multiple changes to be merged into a single commitment. To evaluate this, we vary the interval between successive commitments and measure throughput. As shown in Figure 13(b), throughput increases with longer intervals. However, longer intervals may also increase the latency of validating the latest results, indicating a performance trade-off in selecting the interval. Overall, all tested intervals achieve acceptable throughput, ranging from 10^5 to 8×10^5 node changes per second.

Causality Analysis Proof and Validation. We issue 1,000 random forward and backward causality analysis queries on endpoint logs and record the corresponding proof generation time. As shown in Figures 13.(c) and (d), the time for both forward and backward proofs increases with the number of queried nodes. Overall, both operations are efficient, *e.g.,* generating a forward proof over 10^5 nodes takes only 49 ms. Based on the generated proofs, we measure the execution time of causality analysis validation. As shown in Figures 13.(c) and 13.(d), the validation time also scales proportionally with the number of validated nodes.

Comparison with Enclave-based Validation. To demonstrate the efficiency of VCAUSE, we implement an SGX enclave-based causality analysis system and compare it with VCAUSE using 1,000 random forward queries on 2 GB of OpTC endpoint data. To enable the enclave to process and validate logs at this scale, we partition the logs into chunks and generate an HMAC per chunk. The results show the enclave-based approach incurs an average latency of 2408 ms, whereas VCAUSE requires only 3.08 ms. As noted in § 2.2, this gap stems from substantial enclave context-switch overhead during full log ingestion, integrity validation, and large-scale graph queries.

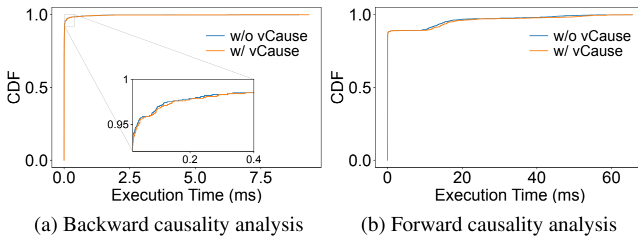
8.4 Runtime Overhead in Practice

To evaluate VCAUSE's runtime overhead in realistic deployments, we integrate it into endpoint systems using two widely used open-source Linux loggers: auditd [55] and Falco [47]. At runtime, VCAUSE ingests system event logs from these loggers and processes them in real time. Following prior results, VCAUSE is configured with segmented outgoing path digests (depth 1) and a commitment interval of 1,000 events.

Since real-world systems often run applications under high workloads, VCAUSE may introduce performance interference. To evaluate this, we deploy two common web application environments: Apache 2 [5] and Nginx [48], each hosting a WordPress website [62]. We then use Apache Bench [23] to issue 10,000 requests with 10 threads and measure the average execution time per request. Furthermore, we evaluate compute-intensive workloads from seven programs from the NAS Parallel Benchmarks. As shown in Table 3, when auditd is used as the logger, VCAUSE introduces less than 1% overhead compared to the baseline. Similar results are observed

Table 3: Runtime overhead of VCAUSE in realistic deployments. We use Apache Bench to issue 10,000 requests with 10 threads on Apache2 and Nginx, and run seven programs from NAS Parallel Benchmarks, reporting the average execution time per request (or run) when auditd is used as the logger.

Application	Baseline	with VCAUSE	Overhead
Apache2	17.127 *ms*	17.146 *ms*	0.11%
Nginx	15.765 *ms*	15.780 *ms*	0.10%
NAS-bench	41.211 *s*	41.354 *s*	0.35%



(a) Backward causality analysis (b) Forward causality analysis

Figure 14: CDF of causality analysis time

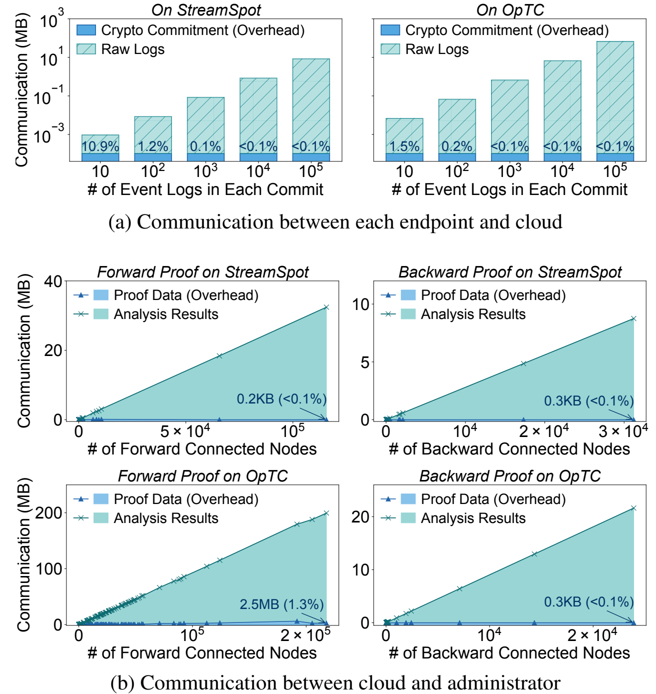
with the Falco logger (see Table 5 in Appendix E).

Based on endpoint logs generated by the benchmarking tool, we further evaluate the overhead of cloud-side causality analysis. For comparison, we implement a baseline causality analysis system without validation. Both systems process 1,000 randomly selected queries, and we record their execution time. Figure 14 shows the cumulative distribution function (CDF) of causality analysis time, indicating that VCAUSE incurs only minor overhead while achieving the same proportion of queries. On average, the overhead of forward and backward proof generation is 3.36% and 8.92%, respectively.

8.5 Communication and Storage Costs

We run VCAUSE on the StreamSpot and OpTC datasets to evaluate its communication and storage costs at scale. We first measure the communication cost between endpoints and the cloud, focusing on the size of cryptographic commitments (*i.e.,* the signed Merkle tree root and associated metadata) transmitted during each commitment. As shown in Figure 15.(a), this cost remains constant at roughly 108 bytes, introducing negligible overhead relative to raw log transmission—below 1.2% when more than 10^2 logs are sent.

We also evaluate the communication cost between the cloud and the administrator, which includes causality analysis results and the additional proof data introduced by VCAUSE. Specifically, we simulate 1,000 random forward and backward queries and record the corresponding costs. As shown in Figure 15.(b), the proof data incurs only minor overhead relative to the returned results. Particularly, the proof size grows slightly with the number of returned nodes. This is because



(a) Communication between each endpoint and cloud

(b) Communication between cloud and administrator

Figure 15: Communication cost of VCAUSE

only the proof for partial nodes and associated path digests need to be returned for validation (see § 6).

Table 4: Average storage cost per endpoint. ADS refers to the authenticated data structures introduced by VCAUSE.

Dataset	Original Graph	ADS	Overhead
StreamSpot	41.85 MB	8.29 MB	19.81%
OpTC	15.76 GB	1.10 GB	6.98%

To assess VCAUSE's storage cost, we measure the size of the authenticated data structures (*i.e.,* graph accumulators and path digests of nodes) stored in the cloud. Since forward and backward validations are always performed jointly on provenance graphs, we apply an optimization that merges each node's incoming and outgoing digests into a single digest. Table 4 reports the average storage per endpoint. Compared to the original graph data, VCAUSE introduces only modest overhead from authentication data. For the OpTC dataset, the overhead is 6.98%, which we view as more representative of practical storage requirements, since OpTC captures real-world, system-wide endpoint logs, while StreamSpot primarily contains logs collected in a controlled lab environment.

9 Discussion

Integration with Causality Analysis Variants. VCAUSE focuses on providing validation for the general form of causality

analysis, while also supporting integration with advanced variants that prune or prioritize analysis paths [20, 28, 39, 42, 45]. Notably, vCAUSE's causality analysis results encompass the outputs of these variants. Thus, users can use vCAUSE to first verify the complete subgraph related to a point-of-interest node and then pass it to downstream methods for refinement. To further reduce overhead, endpoints can selectively include only critical components in the verifiable graph.

10 Conclusion

This paper introduces vCAUSE, an efficient verifiable causality analysis system for cloud-based endpoint auditing. It employs two tailored authenticated data structures: a verifiable versioned provenance graph and a graph accumulator. The graph accumulator enables proof generation for point-of-interest node queries, while the provenance graph uses incoming and outgoing path digests to provide proofs for associated causality relations. Security analysis and experimental results demonstrate the security and efficiency of vCAUSE.

Acknowledgments

We are grateful to all the anonymous reviewers for their insightful comments, which have greatly improved our paper. The research is supported in part by the National Natural Science Foundation of China (No. 62202465), the National Key Research and Development Program of China (No. 2021YFB2910109), the Beijing Key Laboratory of Network Security Protection Technology (No. 2022YFB3103900), and the Outstanding Talent Scheme (Category B) - Qihang Zhou (E3YY141116).

Ethical Considerations

Stakeholders and Potential Impacts. This work proposes a verifiable causality analysis framework for cloud-based endpoint auditing. In this context, the stakeholders include endpoint users whose activities generate audit logs, enterprise security operators, cloud service providers, and the broader security and academic communities. The expected impacts of this work are enhanced integrity, transparency, and trustworthiness in cloud-based endpoint auditing, leading to greater accountability for service providers and stronger resilience against malicious behaviors.

Dual-Use Considerations and Mitigations. While the framework is designed for defensive auditing, verifiable provenance-based causality tracking could, in principle, be misused for intrusive monitoring if deployed without proper safeguards. To mitigate such risks, all experiments were conducted in controlled lab environments and used publicly available datasets that contain no sensitive raw user information. The system design adheres to the principle of least privilege and collects only the necessary system-call logs required for verification. In deployment, endpoint loggers can be configured with privacy-aware policies to control the scope of log collection.

Ethical Reasoning for Publication. With reasonable configurations—such as limited log collection and privacy controls—the framework poses low risk while significantly improving transparency and accountability in cloud-based endpoint auditing. We believe its publication can support community efforts toward more trustworthy and responsible auditing.

Open Science

To comply with open science principles, we make our code and associated materials publicly available through open repositories. In particular, our prototype implementation, data preprocessing scripts, and test datasets are accessible at https://doi.org/10.5281/zenodo.17908629, enabling others to review, build upon our work, and validate the associated results.

References

[1] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1791–1807. IEEE, 2022.

[2] Raza Ahmad, Eunjin Jung, Carolina de Senne Garcia, Hassaan Irshad, and Ashish Gehani. Discrepancy detection in whole network provenance. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*, 2020.

[3] Enes Altinisik, Fatih Deniz, and Hüsrev Taha Sencar. Provg-searcher: A graph representation learning approach for efficient provenance graph search. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2247–2261, 2023.

[4] Md Monowar Anjum, Shahrear Iqbal, and Benoit Hamelin. Anubis: a provenance graph-based framework for advanced persistent threat detection. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1684–1693, 2022.

[5] Apache. Apache license, version 2.0. https://www.apache.org/licenses/LICENSE-2.0, 2004.

[6] Muhammad U Arshad, Ashish Kundu, Elisa Bertino, Arif Ghafoor, and Chinmay Kundu. Efficient and scalable integrity verification of data and query results for graph databases. *IEEE Transactions on Knowledge and Data Engineering*, 30(5):866–879, 2017.

[7] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 163–192. Springer, 1997.

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.

[9] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 677–706. Springer, 2020.

[10] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *2011 31st International Conference on Distributed Computing Systems*, pages 393–402. IEEE, 2011.

[11] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*, pages 55–72. Springer, 2013.

[12] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *2012 IEEE international conference on communications (ICC)*, pages 917–922. IEEE, 2012.

[13] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks. *arXiv preprint arXiv:2202.06877*, 2022.

[14] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical intrusion detection and investigation using whole-system provenance. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3533–3551. IEEE, 2024.

[15] Catalin Cimpanu. Hackers are increasingly destroying logs to hide attacks. https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/, 2023.

[16] Crowdstrike. Cybersecurity's platform for the xdr era. https://www.crowdstrike.com/platform/, 2024.

[17] DARPA. Operationally transparent cyber dataset. https://github.com/FiveDirections/OpTC-data, 2019.

[18] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak, and Daniel Wichs. Message authentication, revisited. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, pages 355–374. Springer, 2012.

[19] Feng Dong, Shaofei Li, Peng Jiang, Ding Li, Haoyu Wang, Liangyi Huang, Xusheng Xiao, Jiedong Chen, Xiapu Luo, Yao Guo, et al. Are we there yet? an industrial viewpoint on provenance-based endpoint detection and response tools. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2396–2410, 2023.

[20] Pengcheng Fang, Peng Gao, Changlin Liu, Erman Ayday, Kangkook Jee, Ting Wang, Yanfang Fanny Ye, Zhuotao Liu, and Xusheng Xiao. {Back-Propagating} system dependency impact for attack investigation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2461–2478, 2022.

[21] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. {SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2987–3004, 2021.

[22] Flexxon. Worm (write once read many). https://www.flexxon.com/worm/, 2024.

[23] The Apache Software Foundation. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[24] geeksforgeeks. What is log tampering? https://www.geeksforgeeks.org/what-is-log-tampering/, 2022.

[25] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, 2005.

[26] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[27] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1172–1189. IEEE, 2020.

[28] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *network and distributed systems security symposium*, 2019.

[29] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via {Fixed-Key} blockcipher. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2389–2406, 2022.

[30] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1139–1155. IEEE, 2020.

[31] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. {Dependence-Preserving} data compaction for scalable forensic analysis. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1723–1740, 2018.

[32] Fuzel Jamil, Abid Khan, Adeel Anjum, Mansoor Ahmed, Farhana Jabeen, and Nadeem Javaid. Secure provenance using an authenticated data structure approach. *computers & security*, 73:34–56, 2018.

[33] Zian Jia, Yun Xiong, Yuhong Nan, Yao Zhang, Jinjing Zhao, and Mi Wen. {MAGIC}: Detecting advanced persistent threats via masked graph representation learning. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5197–5214, 2024.

[34] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, 2001.

[35] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 19–30, 2017.

[36] kbandla. Apt note. https://github.com/kbandla/APTnotes, 2023.

[37] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[38] Ben Laurie. Certificate transparency: Public, verifiable, append-only logs. *Queue*, 12(8):10–19, 2014.

[39] Shaofei Li, Feng Dong, Xusheng Xiao, Haoyu Wang, Fei Shao, Jiedong Chen, Yao Guo, Xiangqun Chen, and Ding Li. Nodlink: An online system for fine-grained apt attack detection and investigation. *arXiv preprint arXiv:2311.02331*, 2023.

[40] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, Yan Chen, and Wei Ruan. Threat detection and investigation with system-level provenance graphs: A survey. *Computers & Security*, 106:102282, 2021.

[41] Haojun Liu, Xinbo Luo, Hongrui Liu, and Xubo Xia. Merkle tree: A fundamental component of blockchains. In *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*, pages 556–561. IEEE, 2021.

[42] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[43] Wu Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. Tprov: towards a trusted provenance-aware service based on trusted computing. In *International Conference on Web Services*, pages 67–83. Springer, 2018.

[44] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):1–21, 2009.

[45] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network And Distributed System Security Symposium (NDSS 2016)*. Internet Soc, 2016.

[46] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1035–1044, 2016.

[47] Francisco Neves, Nuno Machado, et al. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 534–541. IEEE, 2018.

[48] Nginx. Nginx projects. https://nginx.org/, 2004.

[49] OpenSSL. Openssl-library. https://openssl-library.org/, 2024.

[50] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Network and distributed system security symposium*, 2020.

[51] Arttu Paju, Muhammad Owais Javed, Juha Nurmi, Juha Savimäki, Brian McGillion, and Billy Bob Brumley. Sok: A systematic review of tee usage for developing trusted applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–15, 2023.

[52] Fábio Reina, Hylson Vescovi Netto, Luciana Rech, and Aldelir Fernando Luiz. A method to verify data integrity in graph databases. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00220–00223. IEEE, 2018.

[53] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[54] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, volume 1, pages 57–64. IEEE, 2015.

[55] Sematext. What is auditd in linux: A brief tutorial. https://sematext.com/glossary/auditd/, 2021.

[56] Sriranjani Sitaraman and Subbarayan Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE international workshop on information assurance (IWIA'05)*, pages 154–163. IEEE, 2005.

[57] Chase Smith and Alex Rusnak. Dynamic merkle b-tree with efficient proofs. *arXiv preprint arXiv:2006.01994*, 2020.

[58] Amril Syalim, Takashi Nishide, and Kouichi Sakurai. Preserving integrity and confidentiality of a directed acyclic graph model of provenance. In *Data and Applications Security and Privacy XXIV: 24th Annual IFIP WG 11.3 Working Conference, Rome, Italy, June 21-23, 2010. Proceedings 24*, pages 311–318. Springer, 2010.

[59] Symantec. Symantec endpoint detection and response. https://docs.broadcom.com/doc/endpoint-detection-and-response-atp-endpoint-en, 2024.

[60] Songlei Wang, Yifeng Zheng, Xiaohua Jia, and Xun Yi. Pegraph: A system for privacy-preserving and efficient search over encrypted social graphs. *IEEE Transactions on Information Forensics and Security*, 17:3179–3194, 2022.

[61] Weijie Wang, Yujie Lu, Charalampos Papamanthou, and Fan Zhang. The locality of memory checking. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1820–1834, 2023.

[62] WordPress. Meet wordpress-the open source publishing platform of choice for millions of websites worldwide. https://wordpress.org/, 2024.

[63] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. Veridb: An sgx-based verifiable database. In *SIGMOD International Conference on Management of Data*, pages 2182–2194, 2021.

A Proofs for Node Queries

Algorithm 1: Search and Proof in Hierarchical Trees

Input: Global and local tree roots r_G, r_L, the queried entity s and temporal relation $r \in \{'\preceq t', '\succeq t'\}$, and the proof (ρ_G, ρ_L).
Output: A boolean value and the matched node.

```
1  Function GlobalSearch(r_G, s, ρ_G):
2    if r_G is a leaf node then
3      if r_G.leaf.min == s then
4        return True, r_G;              ▷ exact matching
5      return False, r_G;
6    if s ∈ [r_G.left.min, r_G.left.max] then
7      ρ_G.Append(r_G.right);
8      return GlobalSearch(r_G.left, s, ρ_G);
9    else if s ∈ [r_G.right.min, r_G.right.max] then
10     ρ_G.Append(r_G.left);
11     return GlobalSearch(r_G.right, s, ρ_G);
12   else                ▷ not in left and right subtrees
13     ρ_G.Append(r_G);
14     return False, r_G;
15 Function LocalSearch(r_L, r, ρ_L):
16   if r_L is a leaf node then
17     return True, r_G;                ▷ fuzzy matching
18   if (r = ⪯ t and t ∈ [r_L.left.min, r_L.right.min)) or (r
       is ⪰ t and t ≤ r_L.left.max) then
19     ρ_L.Append(r_L.right);
20     return GlobalSearch(r_L.left, r, ρ_L);
21   else if r is ⪯ t and t ≥ r_L.right.min or (r is ⪰ t
       and t ∈ (r_L.left.max, r_L.right.max]) then
22     ρ_L.Append(r_L.right);
23     return GlobalSearch(r_L.left, r, ρ_L);
24   else            ▷ all subtrees not satisfy r
25     ρ_L.Append(r_L); return False, r_L;
```

A.1 Single-Node Proof

Membership Proof. For a single-node query $N(s,r)$, the membership proof is generated by searching the global tree

for the system entity keyword s and the local tree for the temporal relation r. These searches are implemented by two recursive functions, *GlobalSearch* and *LocalSearch*, as shown in Algorithm 1. *GlobalSearch* performs an exact search using s, while *LocalSearch* conducts a fuzzy search based on the temporal relation $r \in \{\preceq t, \succeq t\}$.

Non-Membership Proofs. A non-membership proof for a node query $N(s,r)$ is represented as a tuple $(\tilde{\rho}_G, \tilde{\rho}_L)$, where $\tilde{\rho}_G$ is obtained by searching the global tree for the entity keyword s, and $\tilde{\rho}_L$ is obtained by searching the corresponding local tree for the temporal relation r. Two cases may arise:

1) Condition #1. No node matches s, and the search terminates in the global tree. The proof contains only $\tilde{\rho}_G$, consisting of the sibling nodes along the search path.

2) Condition #2. Nodes matching s exist, but none satisfy r. The search reaches the local tree of s but terminates without a match for r. The proof contains both $\tilde{\rho}_G$ and $\tilde{\rho}_L$, each comprising the sibling nodes encountered in the global and local trees, respectively.

A.2 Entity-Temporal Range Proof

We now extend our graph accumulator to provide proofs for temporal range queries associated with an entity, *i.e.,* $Q(s,[a,b])$, where s denotes the entity ID and $[a,b]$ indicates a closed timestamp range.

Entity-Temporal Range Proofs. The proofs are structured as a tuple $(\rho_G, (\rho_L^l, \rho_L^r))$, where ρ_G is a proof derived from the global tree, and (ρ_L^l, ρ_L^r) represent range proofs from the corresponding local tree. Specifically, ρ_L^l and ρ_L^r together prove the integrity of successive versioned nodes spanning the temporal range $[a,b]$. Notably, when $a = b$, the proof $(\rho_G, (\rho_L^l, \rho_L^r))$ becomes equivalent to a single-node proof.

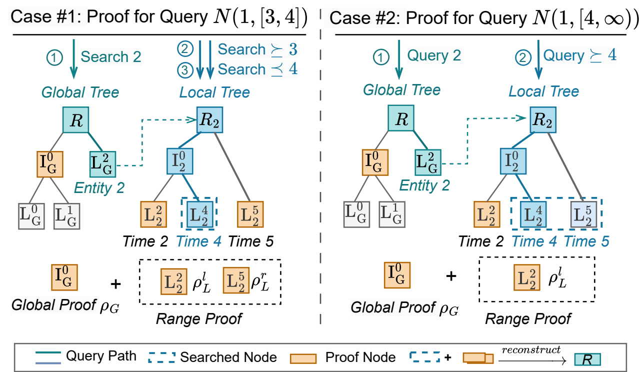


Figure 16: Temporal range proof

Figure 16 illustrates the generation of the proof tuple $(\rho_G, (\rho_L^l, \rho_L^r))$ for query $Q(s,[a,b])$. The proof ρ_G is generated through an exact search in the global tree. Siblings encountered along the search path are included in ρ_G. The local range proofs ρ_L^l and ρ_L^r are derived from separate searches in the local tree of s for the leftmost and rightmost nodes whose timestamp keywords are within the range $[a,b]$. The left-hand siblings along the search path to the leftmost are integrated into ρ_L^l, while the right-hand siblings along the path to the rightmost are incorporated into ρ_L^r.

Algorithm 2: Proving Nodes for Entity-Temporal Range Query $\mathbf{Q}(\mathbf{s},[\mathbf{a},\mathbf{b}])$

Input: The root r of the global tree, the queried system entity s, the root r_s of system entity s's local tree, and the queried timestamp range $[a,b]$.
Output: A boolean value indicating node existence, and the temporal range proof $(\rho_G, (\rho_L^l, \rho_L^r))$.

1 $\rho_G = []$;
2 st_s, n_s = GlobalSearch(r, s, ρ_G);
3 **if** $st_s == False$ **then**
4   **return** *False*, $(\rho_G, (\varnothing, \varnothing))$;
5 $\rho_L^l = [], \rho_L^r = []$;
6 st_a, *lmost* = LocalSearch($r_s, '\succeq a', \rho_L^l$);
7 st_b, *rmost* = LocalSearch($r_s, '\preceq b', \rho_L^r$);
8 $\rho_L^l = LeftSib(\rho_L^l)$, $\rho_L^r = RightSib(\rho_L^r)$;
9 **if** *rmost.time* $> a$ **or** *lmost.time* $< b$ **then**
10   **return** False, $(\rho_G, (\rho_L^l, \rho_L^r))$;
11 **return** True, $(\rho_G, (\rho_L^l, \rho_L^r))$;

For a query $Q(s,[a,b])$, the corresponding proof generation algorithm is shown in Algorithm 2, which leverages the *GlobalSearch* and *LocalSearch* described in Algorithm 1.

Validation. Given a temporal range proof $(\rho_G, (\rho_L^l, \rho_L^r))$ and the queried nodes, the verifier reconstructs the search path and tree structure. The proof is valid if the reconstructed root matches the committed root. To verify that the queried nodes satisfy $Q(s,[a,b])$, the verifier checks that they lie in the local tree of s and that their timestamp keywords fall within $[a,b]$. Notably, using temporal range proofs enables efficient batch verification of m nodes. Verifying nodes individually incurs $O(m \log N)$ complexity, whereas batch verification with temporal range proofs reduces this cost to $O(m)$.

Algorithm 3: DIM-Tree Node Insertion Operation

Input: A new leaf node t, the leaf node vector L, and a stack storing previous subtree roots s.
Output: a new stack s.

1 L.Append(t);
2 **while** s.Empty() **do**
3   **if** $s.top.height \neq t.height$ **then**
4   **break**;
5   t = TreeMerge($s.top, t$);
6   $t.height += 1$; s.Pop();
7 s.Push(t);
8 **return** s;

B Node Insertion Algorithm in DIM-Tree

Algorithm 3 shows the node insertion in DIM-Tree.

C Algorithms of Causality Relation Validation

Algorithm 4: Validating Backward Causality Relations for a Given Node

```
Input: a node n, its backward causally related nodes
       and edges {V_{→n}, E_{→n}}, and the hash function
       H(·).
Output: a boolean variable.
1  Function Main(n, {V_{→n}, E_{→n}}):
2      original = n.Π_I;
3      DFSInDigest(n, {V_{→n}, E_{→n}}, {});
4      if n.Π_I == original then
5          return True;
6      return False;
7  Function DFSInDigest(n, {V_{→n}, E_{→n}}, visisted):
8      if n ∈ visisted then
9          return;
10     visited.Add(n);
11     byte = '';
12     for e ∈ n.E_in do
13         if e ∉ E_{→n} or e.src ∉ V_{→n} then
14             return;
15         DFSInDigest(e.src);
16         byte ||= str(e) || e.src.Π_I;
17     n.Π_I = H(byte);
```

Algorithm 4 and 5 detail the process of validating a node n's forward and backward causality relations using incoming and segmented outgoing path digests.

Algorithm 5: Validating Forward Causality Relations for a Given Node

```
Input: a node n, its forward causally related nodes and
       edges {V_{n→}, E_{n→}}, temporal range node proofs
       P, and the hash function H(·).
Output: a boolean variable.
1  Function Main(n, {V_{n→}, E_{n→}}):
2      rts = ExtractRoot(V_{n→});
3      NodeVerify(rts, P);
4      foreach s ∈ {n} ⋃ rts do
5          original = s.Π_O^{sg};
6          DFSOutDigest(s, {V_{→n}, E_{→n}}, {});
7          if s.Π_O^{sg} ≠ original then
8              return False;
9      return True;
10 Function DFSOutDigest(n, {V_{n→}, E_{n→}}, visit):
11     if n ∈ visit then
12         return;
13     if n is a terminal node then
14         return;
15     visit.Add(n);
16     byte = '';
17     for e ∈ n.E_out do
18         if e ∉ E_{n→} or e.dst ∉ V_{n→} then
19             return;
20         DFSOutDigest(e.dst);
21         byte ||= str(e) || e.dst.Π_O^{sg};
22     n.Π_O^{sg} = H(byte);
```

D Detailed Security Definitions and Proofs

D.1 Security of Verifiable Versioned Provenance Graphs

We follow the preimage resistance property of hash functions to define the key security property of our verifiable versioned provenance graphs, *i.e.,* causality relation unforgeability. It is formalized via the following game $Forge_{\mathcal{A}}^{vGraph}(k)$:

1) Setup. The challenger initializes system parameters.

2) Challenge. The challenger selects a node n and sends its backward and forward causally related nodes and edges $\{V_{\to n}, E_{\to n}\}$ and $\{V_{n\to}, E_{n\to}\}$ to $\mathcal{A}$.

3) Forgery. The adversary $\mathcal{A}$ attempts to add, delete, or modify a node and edge in either $\{V_{\to n}, E_{\to n}\}$ or $\{V_{n\to}, E_{n\to}\}$, then sends them back to the challenger.

4) Validation. The challenger examines n's forged causally related nodes and edges, $\{V^*_{\to n}, E^*_{\to n}\}$ and $\{V^*_{n\to}, E^*_{n\to}\}$, using n's incoming and outgoing path digests, respectively. If the validation succeeds, $\mathcal{A}$ wins the game and outputs 1.

Definition 3 (Causality Relation Unforgeability) *A verifiable versioned provenance graph achieves causality relation unforgeability if, for any polynomial-time adversary $\mathcal{A}$:*

$$\Pr(Forge_{\mathcal{A}}^{vGraph}(k) = 1) \leq negl(k) \tag{8}$$

where $negl(k)$ denotes a negligible function.

Now, we give the following security theorem and proofs.

Theorem 2 *A verifiable versioned provenance graph achieves causality relation unforgeability if the used hash function is second-preimage resistant.*

Proof. *Case #1:* Suppose the adversary $\mathcal{A}$ adds, deletes, or modifies a node or edge within the backward causality set $V_{\to n}, E_{\to n}$. If there exists a path of length l from the queried node n to the tampered node or edge, then there should exist

at least one intermediate node n_i along the path for which the following condition holds:

Let the original incoming path string of n_i be defined as:

$$m = \{(s(e) \parallel e.src.\Pi_I) \mid e \in n_i.E_{in}\}$$

After tampering, the string becomes m^*. To win the game, **A** should ensure that the resulting hash digest $H(m^*)$ equals the original digest $n_i.\Pi_I = H(m)$, in order to pass the validation described in § 5.2. Given this equality, the adversary $\mathcal{B}$ can construct a second-preimage attack by outputting the pair (m, m^*) such that $m \neq m^*$ and $H(m) = H(m^*)$.

Case #2: If the added, deleted, or modified node or edge is within $\{V_{n\rightarrow}, E_{n\rightarrow}\}$, the construction for $\mathcal{B}$ is similar to the first case. That is, the modified outgoing path digest $n_i.\Pi_O$ after tampering yields another second-preimage attack pair (m, m^*) satisfying $H(m) = H(m^*)$ and $m \neq m^*$.

In conclusion, if the hash function is second-preimage resistant—meaning that the probability of such an attack succeeding is negligible—then the verifiable versioned provenance graph achieves causality relation unforgeability. □

D.2 Security of vCAUSE

We now present the security proof for vCAUSE's core security property, *i.e.,* the unforgeability of causality analysis results (Theorem 1). For simplicity, we focus on vCAUSE with unsegmented outgoing path digests. The analysis naturally extends to the segmented setting, where validation involves multiple analogous sub-validations over segmented trees.

Proof for Theorem 1. We prove the unforgeability of causality analysis results via a standard hybrid argument over four games, G_0–G_3, where G_0 refers to the original $Forge_{\mathcal{A}}^{\text{vCAUSE}}$ game. Each subsequent game modifies the previous one using a simulator S that emulates the signature scheme, Merkle tree, and verifiable versioned provenance graph.

1) Game G_0. G_0 is equivalent to the original $Forge_{\mathcal{A}}^{\text{vCAUSE}}$ game, except that it immediately outputs 1 (a win for the adversary) whenever a successful forgery is detected, regardless of which cryptographic component is compromised. Hence,

$$Pr[Forge_{\mathcal{A}}^{\text{vCAUSE}}(k) = 1] \leq Pr[G_0 = 1]. \tag{9}$$

From $\mathcal{A}$, we can construct three adversaries $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ that attack the signature, Merkle tree, and provenance graph.

2) Game G_1. This game is identical to G_0, except that S now simulates the signature scheme. Verification on (R, t, σ_R) is replaced by checking bookkeeping records; unmatched or invalid signatures are rejected. As S perfectly simulates signing, $\mathcal{B}_1$ has negligible advantage. Hence, comparing G_0 and G_1, we have:

$$|Pr[G_1 = 1] - Pr[G_0 = 1]| \leq Adv * \mathcal{B}_1{}^{sign}. \tag{10}$$

3) Game G_2. G_2 is identical to G_1, except that S now simulates the Merkle tree. For each node n, if proof ρ_n mismatches the recorded data or the queried value, verification rejects. As S can perfectly simulate the Merkle tree behavior, $\mathcal{B}_2$ cannot win a forgery game against the Merkle tree structure. Thus,

$$|Pr[G_2 = 1] - Pr[G_1 = 1]| \leq Adv * \mathcal{B}_2{}^{Merkle}. \tag{11}$$

4) Game G_3. S further simulates the verifiable provenance graph. The validation of node n's causally related components is replaced by a check against bookkeeping records; For node n, if its backward or forward components $\{V_{\rightarrow n}, E_{\rightarrow n}\}$ or $\{V_{n\rightarrow}, E_{n\rightarrow}\}$ mismatch their paths, verification rejects. Hence,

$$|Pr[G_3 = 1] - Pr[G_2 = 1]| \leq Adv * \mathcal{B}_3{}^{vGraph}. \tag{12}$$

5) Conclusion. By summing up the above results, we have:

$$\begin{aligned} Pr[Forge_{\mathcal{A}}^{\text{vCAUSE}} = 1] \leq Pr[G_0 = 1] \leq Adv_{\mathcal{B}_1}^{sign} \\ + Adv_{\mathcal{B}_2}^{Merkle} + Adv_{\mathcal{B}_3}^{vGraph} + Pr[G_3 = 1] \end{aligned} \tag{13}$$

All cryptographic components in G_3 are perfectly simulated, preventing any forgery; hence $Pr[G_3 = 1] = 0$. The adversary's overall advantage is therefore negligible, *i.e., $negl(k)$*, assuming the signature scheme is EUF-CMA secure, the Merkle tree is position-binding, and the verifiable versioned provenance graph ensures causality-relation unforgeability. □

E Additional Evaluation Results

Runtime Overhead of vCAUSE. Table 5 reports the average per-request (or per-run) runtime overhead for Apache2, Nginx, and NAS Benchmarks when Falco is used as the logger.

Table 5: Runtime overhead of vCAUSE under realistic deployment settings with Falco as the logger.

Application	Baseline	with vCAUSE	Overhead
Apache2	16.998 *ms*	17.018 *ms*	0.12%
Nginx	15.652 *ms*	15.668 *ms*	0.10%
NAS-Bench	41.312 *s*	41.489 *s*	0.43%

Case Study. We take a medium-scale backward/forward causality analysis on OpTC endpoint 660 as an example. The backward analysis involves 19,088 nodes and completes in 11.8 ms, while the forward analysis involves 639 nodes and completes in 0.8 ms. To evaluate vCAUSE 's verifiability, we modify each node's attributes and run validation. We observe that all tampering attempts are correctly detected. On average, validation takes 1.7 ms for forward analysis and 31.0 ms for backward analysis.