TROY UNIVERSITY
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF MATHEMATICS AND INFORMATICS



FINAL GROUP PROJECT REPORT

Security and Complexity Evaluation of Hash-Chain Audit Logs for Detecting Data Tampering

Supervisor: Assoc. Prof. Dr. Nguyen Dinh Han

Course: CS-3372 — Formal Languages and the Theory of Computation

Group: [GROUP]

Students:

1. [STUDENT_1] — [ID_1]
2. [STUDENT_2] — [ID_2]
3. [STUDENT_3] — [ID_3]
4. [STUDENT_4] — [ID_4]

HANOI, AUGUST 2026

Troy University Resources

Summary of Member Contributions

1. [STUDENT_1] – [ID_1] (Rating: 10/10)

- As the **group leader**, [S1] kept the team on track, helped set the research direction, and coordinated the work from beginning to end.
- He was involved in almost every stage of the project, including narrowing the scope, developing the threat model, helping design the prototype, running tampering experiments, and reviewing performance results.
- Early in the project, he worked with [S2] to refine the research questions on append efficiency, full-chain verification, attacker recomputation, and the use of trusted checkpoints.
- He worked closely with [S2] on the formal model and complexity analysis, with [S3] on attacker assumptions and attack cases, and with [S4] to keep the data model, append logic, verifier, and report equations aligned.
- He also took part in the controlled attack runs with [S3], covering modification, deletion, insertion, reordering, suffix recomputation, truncation, rollback, and complete-history replacement.
- During the final stage, he coordinated the technical review and integrated the implementation, experimental results, figures, and report materials into the final submission.
- **Products:** Overall research framework, integrated technical design, tampering-test coordination, final technical review, report compilation, and presentation coordination.

2. [STUDENT_2] – [ID_2] (Rating: 8.75/10)

- [S2] focused mainly on **formal modeling**, analytical work, and performance evaluation.
- He turned the main design ideas into precise definitions for the record structure, deterministic encoding, sequential hash relation, and verification procedure used throughout the report.
- Working with [S3], he separated three ideas that could easily be confused: evidence of tampering, an internally consistent chain, and proof that the stored history is authentic. This distinction shaped the discussion of recomputation, rollback, and truncation.
- He also contributed to the **security literature review** and helped explain attacker recomputation, truncation, rollback, forward security, external commitments, and the limitations of a locally stored unkeyed hash chain. This helped present the prototype as a useful experimental baseline rather than a complete secure-logging solution.
- Together with [S1], he analyzed append and verification complexity, selected the benchmark sizes, compared the runtime results with the expected linear trend, and reviewed storage and checkpoint-frequency trade-offs. He also worked with [S3] to connect the attack outcomes to the research questions.
- **Products:** Formal hash-chain model, security comparison, complexity analysis, performance interpretation, and the mathematical review of the final report.

3. [STUDENT_3] – [ID_3] (Rating: 8.75/10)

- [S3] focused on **adversarial analysis**: how the log and verifier would behave under different tampering attempts and attacker capabilities.
- He and [S1] turned the threat model into concrete test cases for modification, deletion, insertion, reordering, suffix recomputation, truncation, rollback, and whole-history replacement.
- He worked with [S4] on checks for log identifiers, sequence numbers, predecessor hashes, deterministic record reconstruction, recomputed hash values, and clear failure reporting. He then worked with [S1] to develop and run the **controlled attack suite**.
- A key part of his evaluation was comparing attacks that failed immediately with attacks that became locally valid after rebuilding the affected suffix. He helped clarify that this recomputation does not require finding a SHA-256 collision.
- After the experiments, he worked with [S2] to review the results and explain why some attacks broke the chain while others remained locally valid. They also examined why truncation and rollback can preserve a valid prefix and how **independently protected checkpoints** change the verification result.
- Before the security discussion was finalized, he also checked the attack-result matrix, the checkpoint-boundary experiment, and the evidence produced by the automated tests.
- **Products:** Threat model, verification support, tampering-test suite, attack-result matrix, checkpoint experiments, and the report's security analysis.

4. [STUDENT_4] – [ID_4] (Rating: 8.75/10)

- [S4] was the **primary developer** of the prototype and translated the formal hash-chain model into working code.
- He implemented the full append workflow: creating new audit records, retrieving the previous chain value, deterministically encoding the authenticated fields, performing the core **SHA-256 append operations**, and storing the resulting chain metadata.
- He worked with [S3] on the **complete-chain verification process**, including checks for sequence breaks, incorrect predecessor links, and reconstructed-hash mismatches.
- This implementation provided the technical base for the modification, deletion, insertion, reordering, recomputation, truncation, rollback, and complete-history replacement experiments.
- He also supported the experimental workflow by generating legitimate datasets, writing reproducible JSONL files, creating controlled tampered copies, and running both local and checkpoint-aware verification.
- During testing, he ran automated checks, fixed implementation issues, and verified that the code and generated files matched the formal equations and the procedure described in the Methodology and Results sections.
- **Products:** Working hash-chain prototype, append and verification logic, deterministic encoding, experimental data generation, test support, and source-code appendices.

Troy University Resources

Table of Contents

Summary of Member Contributions **i**

1. [STUDENT_1] – [ID_1] (Rating: 10/10) i
2. [STUDENT_2] – [ID_2] (Rating: 8.75/10) i
3. [STUDENT_3] – [ID_3] (Rating: 8.75/10) ii
4. [STUDENT_4] – [ID_4] (Rating: 8.75/10) ii

List of Figures **v**

List of Tables **vi**

Introduction **1**

1 Motivation and Project Context 1
2 Research Problem 1
3 Project Requirements 1
4 Research Objectives 2
5 Research Questions 2
6 Research Basis and Related Work 3
7 Proposed Approach and Strategy 3
8 Project Deliverables 4

Methodology **6**

1 Threat Model and Practical Attack Scenarios 6
2 Secure-Logging Foundations Beyond Plain Hash Chains 7
3 History Trees and External Commitments 9
4 Theory-of-Computation Perspective 10
5 Audit-Record Model 11
6 Deterministic Encoding 12
7 Hash-Chain Construction 13
8 Append Algorithm 13
9 Chain-Verification Algorithm 14
10 Tampering Scenarios 14
11 Trusted Checkpoints and Backup Hashes 16
12 Prototype Implementation and Experimental Tools 16
13 Experimental Workflow 17
14 Dataset Sizes 18
15 Performance and Security Metrics 18
16 Experimental Repetition and Statistical Analysis 19

Results and Discussion **20**

1 Experimental Run Overview 20
2 Normal-Chain Verification 20
3 Observed Tampering-Detection Results 21
4 Modification, Insertion, Deletion, and Reordering 22
5 Truncation, Rollback, and Whole-History Replacement 23
6 Checkpoint-Window Experiment 23
7 Append-Time Evaluation 24
8 Full-Chain Verification Evaluation 25
9 Variable-Payload Verification 26
10 Storage-Overhead Evaluation 27
11 Comparison with Theoretical Complexity 28
12 Security Properties and Limitations 29
13 Comparison with Secure-Logging Literature 30
14 Answers to the Research Questions 30
15 Research Contributions 31
16 Limitations of the Experimental Study 32
17 Conclusion and Future Work 32

References **33**

Appendices **35**
Appendix A. Project Work Plan 35
Appendix B. Advanced Secure-Logging Constructions 37
Appendix C. History-Tree Technical Details 41
Appendix D. Additional Theory-of-Computation Formalisms 43
Appendix E. Detailed Experimental Data 45
Appendix F. Tampering Test Cases 46
Appendix G. Selected Source Code 46

Troy University Resources

List of Figures

1 Implemented hash-chain creation and verification strategy. 4
2 Local and checkpoint-aware verification outcomes for controlled attack scenarios. 22
3 Measured append cost with the current chain head supplied directly; error bars show approximate 95% normal confidence intervals. 25
4 Measured full-chain verification time and least-squares linear fit; error bars show approximate 95% normal confidence intervals. 26
5 Verification time versus total serialized log size for a fixed 100,000-record dataset. 27
6 Measured JSONL overhead with two stored hashes versus current-hash only. 28

List of Tables

1 Adversary levels and their role in the evaluation 6
2 Plain SHA-256 prototype compared with stronger secure logging 9
3 Conceptual comparison of hash chains and history trees 10
4 Audit-record structure 12
5 Expected append scaling by storage strategy 14
6 Prototype implementation environment 17
7 Software components and experimental roles 17
8 Experimental log sizes 18
9 Observed baseline verification results 20
10 Observed tampering outcomes in the prototype 21
11 Observed checkpoint-boundary behavior with a checkpoint at record 200 . . 23
12 Projected checkpoint-storage trade-off assuming a 182-byte serialized checkpoint 24
13 Measured cryptographic append cost with direct access to the chain head . . 24
14 Observed full-chain verification performance 25
15 Observed verification performance for 100,000 records with variable payloads 27
16 Measured serialized storage overhead 28
17 Experimental baseline compared with representative secure-logging approaches 30
18 Completed project work plan and status 35

Troy University Resources

Introduction

1 Motivation and Project Context

Audit logs are an important source of evidence for system monitoring, accountability, incident investigation, and computer forensics.

However, when audit records are stored on a system that may later be compromised, an attacker may attempt to modify, delete, insert, or reorder records in order to hide previous activities.

Cryptographic audit-log mechanisms therefore aim to make unauthorized changes detectable [1].

This project studies the use of a sequential hash chain as a simple cryptographic mechanism for protecting audit-log integrity.

2 Research Problem

Ordinary audit logs may be changed without leaving an obvious indication that their contents or sequence have been altered.

The central research problem of this project is therefore:

> How effectively and efficiently can a hash-chain-based audit-log mechanism detect tampering in sequential log records, and what security limitations remain when an attacker is able to modify the stored log and recompute hashes?

3 Project Requirements

The project has the following requirements:

1. Each record must use deterministic encoding and a unique chain identifier (`log_id`).
2. The prototype must support appending new records.
3. The prototype must verify the integrity of the complete audit-log chain.
4. The system must evaluate detection of:
    - unauthorized modification;
    - record deletion;
    - record insertion;
    - record reordering.
5. The project must evaluate:

- append time;
- full-chain verification time;
- storage overhead.

6. Security assumptions and limitations must be analyzed, including collision resistance, attacker hash recomputation, and the role of trusted checkpoints or backup hashes.

4 Research Objectives

The objectives of the project are:

1. Formalize a sequential hash-chain model for audit logs.
2. Develop a working prototype for creating, appending, and verifying audit records.
3. Experimentally evaluate tampering detection.
4. Analyze the computational complexity of append and verification operations.
5. Measure practical runtime and storage costs.
6. Identify security limitations of a plain hash chain.
7. Evaluate trusted checkpoints or externally stored backup hashes as possible improvements.

5 Research Questions

RQ1: What is the cryptographic append-time complexity of the hash-chain audit log, and does the isolated append computation remain independent of the existing log length?

RQ2: How does full-chain verification time scale with record count and total encoded data size?

RQ3: Which modification, deletion, insertion, and reordering attacks are detected when the attacker cannot recompute stored cryptographic values?

RQ4: Which attacks remain possible when the attacker can reconstruct affected portions of the chain, including final-record modification, truncation, rollback, and whole-log replacement?

RQ5: How does externally protected checkpoint frequency trade checkpoint overhead against the maximum amount of recent history that can be rewritten without contradicting a trusted commitment?

Troy University Resources

6 Research Basis and Related Work

Hash chaining has a long history in mechanisms that commit to an ordered sequence of events. Haber and Stornetta showed how linked cryptographic commitments can establish chronological evidence for digital documents [2]. Schneier and Kelsey later combined hash chaining with evolving authentication keys and a trusted remote verifier to protect audit records even when the logging machine is eventually compromised [1].

Ma and Tsudik formalized the stronger objective of *forward-secure stream integrity*: compromise of the current logging state should not permit undetectable insertion, alteration, deletion, or reordering of records created before compromise. Their analysis also emphasizes truncation attacks, in which an attacker removes a valid suffix of the log [3].

External commitments provide another important line of defense. Waters et al. form encrypted audit records into a hash chain and periodically publish verification values to other servers, thereby creating checkpoints outside the mutable log [4]. Maniatis and Baker similarly bind histories through timeline entanglement, while Crosby and Wallach use authenticated history trees to support externally auditable consistency and logarithmic-size proofs [5, 6].

Later schemes explore forward-secure aggregation, efficient public verification, selective disclosure, and robustness after corruption [7–12]. Hartung's attacks on several proposed secure-logging constructions also demonstrate that security claims must be interpreted under the precise adversary model and proof assumptions [13].

This project does not claim to invent secure hash-chain logging or to reproduce a complete forward-secure protocol. Instead, it isolates the sequential hash-chain mechanism, characterizes the boundary between internal consistency and historical authenticity, and evaluates the role of independently protected checkpoints.

7 Proposed Approach and Strategy

Each audit record is linked to the preceding record using SHA-256.

The project uses the following model:

$$H_i = \mathrm{SHA256}\left(\mathrm{Encode}(\text{log_id}, i, T_i, R_i, H_{i-1})\right), \tag{1}$$

where:

- log_id identifies the audit-log chain;
- i is the record sequence number;
- T_i is the record timestamp;
- R_i represents the remaining record data;

- H_{i-1} is the previous record hash;
- H_i is the hash of the current record.

The prototype is evaluated on legitimate logs and controlled tampered datasets. The central distinction studied throughout the report is

$$\text{valid internal chain structure} \neq \text{proof of the original history.} \tag{2}$$

A plain chain can show that the supplied records agree with the supplied hashes. Establishing that this is the same history observed at an earlier trusted time requires an external commitment or another independently protected authentication mechanism.

Figure 1 summarizes the implemented data path and the corresponding verification strategy.

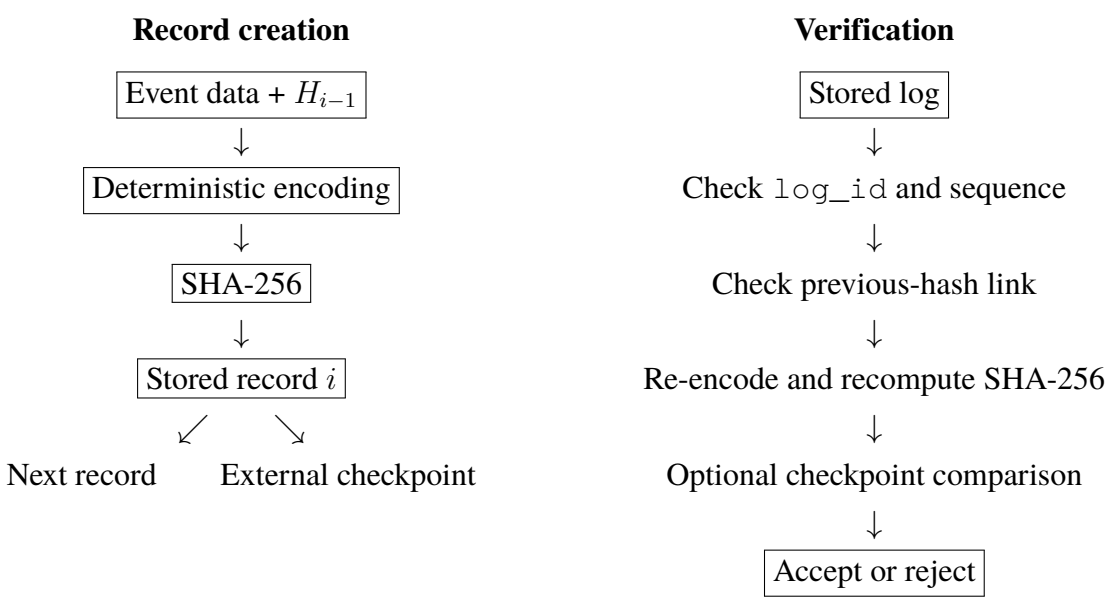


Figure 1: Implemented hash-chain creation and verification strategy.

8 Project Deliverables

The project is organized into the following stages:

1. research-scope definition and literature review;
2. threat-model and requirement definition;
3. formalization of the hash-chain model;
4. prototype design and implementation;
5. tampering-test development;

Troy University Resources

6. correctness evaluation;
7. runtime and storage experiments;
8. integration and interpretation of findings;
9. final report and presentation preparation.

The complete project work plan is provided in Appendix A. All planned technical and documentation activities were completed within the project schedule. The prototype, automated tests, experimental results, final report, presentation materials, and supporting submission package were completed and reviewed by the group.

Methodology

1 Threat Model and Practical Attack Scenarios

The log is assumed to be generated correctly before compromise. The attacker may later obtain write access to the stored file and attempt record modification, deletion, insertion, reordering, suffix replacement, rollback, or complete history replacement. Four adversary levels clarify which security claims are being tested.

1.1 A1: Naive Tampering Adversary

The attacker changes stored records or their order but retains the original cryptographic values. This model tests the direct tamper evidence provided by the recursive chain dependency.

1.2 A2: Recomputation Adversary

The attacker can change records and recompute every affected hash. This is the critical model for demonstrating that an unkeyed chain stored in the same mutable file provides internal consistency but not historical authenticity.

1.3 A3: Full Logger-Compromise Adversary

The attacker controls the logging machine, its current cryptographic state, and subsequent records. Forward-secure logging schemes address this stronger model by evolving keys so that compromise of the current state does not permit forgery for earlier epochs [3, 12, 14].

1.4 A4: Equivocating Logger

A malicious logger maintains multiple apparently valid histories and presents different versions to different observers. Preventing this behavior requires externally visible commitments or cross-system auditing rather than a locally stored chain alone [5, 6].

Table 1: Adversary levels and their role in the evaluation

Adversary	Practical interpretation	Evaluated?
A1: Naive tampering	Edits, deletes, inserts, or reorders records while leaving stored hashes unchanged.	Yes
A2: Recomputation	Changes the stored history and recomputes every affected SHA-256 value.	Yes
A3: Logger compromise	Controls the active logger, its current cryptographic state, and subsequent records.	Literature comparison
A4: Equivocation	Presents different apparently valid histories to different observers.	Literature comparison

Troy University Resources

The prototype principally evaluates A1 and A2. A3 and A4 define literature-based limitations and motivate future extensions. A2 is the central adversary for the prototype because it exposes the difference between detecting unrepaired corruption and authenticating the original history.

2 Secure-Logging Foundations Beyond Plain Hash Chains

The implemented prototype deliberately uses a simple unkeyed SHA-256 chain so that its security boundary can be measured directly. Stronger secure-logging research adds key evolution, compact authentication, independently visible commitments, or fault tolerance. These constructions are not implemented here; they provide the academic basis for interpreting what the prototype does and does not prove [3, 7, 11].

2.1 Why a Plain Hash Chain Is Insufficient

Against A1, changing a record without repairing its stored cryptographic values breaks the record hash or a later link. Against A2, however, the attacker can change record data and rebuild the affected suffix using the public SHA-256 function. The reconstructed file can once again satisfy every local check.

Consequently, local verification proves only that the supplied records are consistent with the supplied chain state:

$$\text{local consistency} \neq \text{historical authenticity}. \tag{3}$$

Authenticating an earlier history requires some state that the storage attacker cannot rewrite together with the log, such as an external checkpoint, a digital signature, or a forward-secure authentication value.

2.2 Forward Security and Key Evolution

Forward-secure logging evolves secret state through ordered epochs,

$$sk_0 \rightarrow sk_1 \rightarrow \cdots \rightarrow sk_T, \tag{4}$$

and erases obsolete keys. Compromise of the current key should therefore not permit forgery of authentication values created in earlier epochs. This is the essential protection missing from an unkeyed local hash chain: an A2 attacker can evaluate SHA-256 but should not be able to recover erased signing state [1, 3].

2.3 Aggregate and Robust Authentication

Sequential aggregate schemes combine many record-authentication values into a compact aggregate. This reduces authentication storage, but it does not imply that the verifier can ignore the authenticated history. For example, a design may use only a constant number of expensive public-key operations while still processing every record:

$$T_{\text{verify}} = O(1)\ \text{ExpOps} + O(L)\ \text{hash/basic operations}. \tag{5}$$

Robust constructions additionally aim to preserve evidence for unaffected records when a bounded number of positions are corrupted and may authenticate the current log length to resist rollback. These guarantees are stronger than the all-or-nothing local verification implemented in this project [11, 15].

2.4 Cryptanalytic Lessons

Mathematical complexity does not by itself establish secure logging. Later cryptanalysis identified attacks against several proposed aggregate-logging constructions under their stated verification equations [13]. The methodological lesson for this project is to state attacker capabilities explicitly and test the actual verifier rather than treating the presence of cryptography as evidence of security. Detailed constructions and attack algebra are retained in Appendix B.

2.5 Relation to the Implemented Prototype

Table 2 separates the measured prototype from stronger literature-based goals.

Troy University Resources

Table 2: Plain SHA-256 prototype compared with stronger secure logging

Capability	**Plain SHA-256 prototype**	**Stronger secure logging**
Detect unrepaired modification	Yes	Yes
Detect structural inconsistency	Yes	Yes
Resist attacker recomputation	No, unless an independent anchor contradicts the replacement	Forward-secure schemes target this property
Protect pre-compromise history	No	Key-evolution schemes target this property
Detect valid truncation	Only relative to a trusted checkpoint	Some schemes authenticate state or log length
Public verification	No	Available in some designs
Fault tolerance	No	Available in robust constructions
Implementation cost	Low	Higher cryptographic and operational complexity
Implemented in this project	**Yes**	**No; literature comparison only**

3 History Trees and External Commitments

A hash chain binds history sequentially,

$$C_i = H(C_{i-1} \parallel X_i), \tag{6}$$

whereas a history tree organizes the same idea hierarchically through a versioned Merkle structure. Its root commits to a complete prefix and supports compact membership and incremental-consistency proofs between versions [6]. For a complete tree of depth d, the commitment may be represented as

$$C_n = A^n_{0,d}. \tag{7}$$

History trees are not implemented in the prototype. Their role is to show how external commitments can support efficient selective auditing beyond the sequential full scan used here. The underlying indexing, recurrences, frozen hashes, and proof algorithms are retained in Appendix C.

Table 3: Conceptual comparison of hash chains and history trees

Property	**Hash chain**	**History tree**
Structure	Sequential links	Versioned tree
Append concept	Link to previous state	Update the versioned root
Full sequential verification	Natural	Possible
Selective historical proof	Requires intermediate history	Compact tree proof
Used in prototype	Yes	No
Role in this report	Implemented baseline	Literature and future direction

4 Theory-of-Computation Perspective

The implemented verifier can be described directly using formal languages and deterministic computation. Infinite-trace formalisms provide a broader view of continuous logging, but they are not claimed as implemented components.

4.1 Finite Audit Logs as Formal-Language Inputs

Let an encoded finite audit log be

$$x = \mathrm{Encode}(R_1, \ldots, R_n) \in \Sigma^*. \tag{8}$$

Define the language of locally valid encodings as

$$L_{\mathrm{local}} = \{x \in \Sigma^* \mid \mathsf{VerifyLocal}(x) = 1\}. \tag{9}$$

The name L_{local} is intentional. Language membership captures internal consistency, not provenance:

$$x \in L_{\mathrm{local}} \not\Rightarrow x \text{ is the authentic historical log.} \tag{10}$$

4.2 Verification as a Deterministic Decision Procedure

The local verifier decides membership through a deterministic procedure,

$$\mathsf{VerifyLocal}(x) = \begin{cases} 1, & \text{if every structural and cryptographic check succeeds,} \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

For each record, it checks the chain identifier, expected sequence number, previous-

Troy University Resources

hash link, deterministic encoding, and recomputed SHA-256 value. It rejects at the first failed condition or accepts after the final record. An optional independently supplied checkpoint adds a comparison against trusted state, but does not change the deterministic nature of the procedure.

4.3 Complexity of Verification

If B_i is the encoded byte length of record i, a complete scan has

$$T_{\text{verify}} = \Theta\left(\sum_{i=1}^{n} B_i\right). \tag{12}$$

For bounded-size records this reduces to

$$T_{\text{verify}}(n) = \Theta(n). \tag{13}$$

The empirical results test this prediction by fitting complete-chain verification time against record count.

4.4 Continuous Logging as an Extended Formal View

A continuously operating logger may be modeled as a reactive system with an infinite trace. For example, an LTL response property can require that every written record is eventually audited:

$$\mathbf{G}\left(\mathsf{LogWritten} \rightarrow \mathbf{F}\mathsf{Audited}\right). \tag{14}$$

This project does not implement an LTL model checker. Büchi automata, product construction, emptiness checking, and the program-checking analogy are included in Appendix D as an extended formal interpretation of continuous logging.

5 Audit-Record Model

Each audit record contains:

- chain identifier (`log_id`);
- record number;
- timestamp;
- user;
- action;

- affected resource;
- event payload;
- previous hash;
- current hash;
- encoding or protocol version.

Table 4: Audit-record structure

Field	Description
Log identifier	Identifies the audit-log chain and prevents accidental mixing of records from different logs when the expected identifier is trusted.
Record number	Sequential position of the record.
Timestamp	Time at which the event is generated.
User	User or process responsible for the event.
Action	Operation recorded by the audit system.
Affected resource	File or other resource affected by the operation.
Payload	Event-specific data authenticated as part of the audit record.
Previous hash	Hash stored in the preceding chain relationship.
Current hash	SHA-256 value calculated for the current record.
Encoding version	Identifies the deterministic serialization rules used by the verifier.

6 Deterministic Encoding

A cryptographic hash operates on bytes rather than logical records. The writer and verifier therefore use one deterministic compact UTF-8 JSON serialization with fixed field order and unambiguous framing. For record i, define

$$M_i = \mathrm{Encode}(v, \texttt{log_id}, i, T_i, U_i, A_i, F_i, P_i, H_{i-1}) \,, \tag{15}$$

where v is the encoding version, U_i is the user/process, A_i is the action, F_i is the affected resource, and P_i is the event payload. The stored field `current_hash` is deliberately excluded from M_i.

The implementation uses application-specific deterministic compact JSON with a fixed field order, UTF-8 encoding, no insignificant whitespace, and the same rules at the

Troy University Resources

writer and verifier. It does not claim conformance to a standardized canonicalization scheme such as the JSON Canonicalization Scheme in RFC 8785 [16]. The expected `log_id` must be obtained from trusted verifier configuration rather than trusted merely because it appears inside the log.

7 Hash-Chain Construction

The chain relation is

$$H_i = \mathrm{SHA256}(M_i), \tag{16}$$

with a predefined genesis value H_0. Because H_{i-1} is already a field of M_i, it is not concatenated a second time. Each stored record contains `previous_hash`$= H_{i-1}$ and `current_hash`$= H_i$. SHA-256 is specified in the NIST Secure Hash Standard, FIPS 180-4 [17].

This design makes field boundaries explicit in the serialization. The unkeyed chain is not a secret-prefix construction such as $H(K \parallel M)$, so length extension is not the relevant threat. The relevant requirements are unambiguous deterministic encoding and protection of an external historical commitment.

8 Append Algorithm

Appending a new audit record consists of:

1. obtaining the latest sequence number;
2. retrieving the latest hash;
3. constructing the new record;
4. deterministically encoding its fields;
5. calculating SHA-256;
6. storing both previous and current hash values.

Let B_i be the encoded size of record i. The cryptographic work for one append is

$$T_{\mathrm{hash}}(i) = \Theta(B_i). \tag{17}$$

For bounded record size and direct access to the latest hash, this is $\Theta(1)$ with respect to the number n of existing records. The end-to-end implementation cost may nevertheless become $O(n)$ if the program scans, parses, or rewrites the complete file for every append.

Table 5: Expected append scaling by storage strategy

Storage strategy	Expected scaling
Chain head retained in memory	Approximately $O(1)$
Direct access to the final record	Approximately $O(1)$
Complete-file scan for the latest hash	$O(n)$
Complete JSON-array parsing	$O(n)$
Complete-file rewrite on every append	$O(n)$

9 Chain-Verification Algorithm

Verification starts from the beginning of the chain.

For each record i, the verifier:

1. checks the expected sequence number;
2. checks its link to H_{i-1};
3. reconstructs the deterministic encoding;
4. recomputes H_i;
5. compares the recomputed hash against the stored hash;
6. reports the first detected inconsistency.

Because every record must be processed, the complete verification cost is

$$T_{\text{verify}} = \Theta\left(\sum_{i=1}^{n} B_i\right). \tag{18}$$

For approximately constant-size records B, this reduces to $\Theta(nB) = \Theta(n)$. Runtime measurements should therefore be tested against a linear empirical model $t(n) = \alpha + \beta n$, while also reporting the total encoded byte count so that record-size effects are not confused with record-count effects.

10 Tampering Scenarios

10.1 Modification Attack

A legitimate field in an existing record is changed. One test retains the original hashes; a second test recomputes the complete affected suffix.

Troy University Resources

10.2 Interior Deletion Attack

One or more records are removed from the middle of the sequence, both with and without rebuilding later hashes.

10.3 Tail-Truncation Attack

The final record or final k records are removed. The remaining prefix may already be a valid chain, so this case is tested separately from interior deletion.

10.4 Insertion Attack

A forged record is inserted between two legitimate records, first without and then with reconstruction of the affected suffix.

10.5 Reordering Attack

Two or more legitimate records exchange positions, again under both A1 and A2 attacker capabilities.

10.6 Suffix-Recomputation Attack

A record is modified and all hashes from that point onward are recomputed by the attacker.

10.7 Final-Record Attack

The final record is modified and its hash is recomputed.

10.8 Rollback Attack

The current log is replaced by a previously authentic shorter version. Every hash in the restored file may be genuine, so detection requires a fresh or monotonic external commitment to the later state.

10.9 Whole-History Replacement

The original file is replaced by a completely different sequence with a newly computed, internally valid chain. This tests whether the verifier has trusted state independent of the supplied file.

The suffix-recomputation, final-record, truncation, rollback, and whole-history cases expose limitations that cannot necessarily be detected by an unanchored hash chain and do not require finding a SHA-256 collision.

11 Trusted Checkpoints and Backup Hashes

An enhanced configuration periodically stores a structured chain state outside the modifiable audit-log file. For checkpoint position k, the stored object is

$$C_k = (L, k, H_k, T_k, v), \tag{19}$$

where L is the trusted `log_id`, T_k is the timestamp copied from record k, and v identifies the encoding or protocol version. During verification, the verifier reconstructs the chain through record k and checks

$$H'_k \stackrel{?}{=} H_k. \tag{20}$$

A mismatch indicates that the committed prefix has been modified. The verifier also compares the chain identifier, committed length, record timestamp, and version. The stored record timestamp is not an independent freshness proof: a deployed remote service would need to authenticate its own publication time, monotonic counter, or equivalent freshness metadata. The checkpoint itself must be independently protected, for example by a remote service or by a digital signature whose signing key is isolated from the logger; otherwise the attacker can replace both the log and its checkpoint [4].

If a checkpoint is created every m records, the records after the most recent checkpoint form an unanchored window with approximate maximum size

$$W_{\max} \leq m - 1. \tag{21}$$

Smaller intervals reduce the amount of recent history that can be rewritten but increase external storage and communication overhead.

12 Prototype Implementation and Experimental Tools

A working prototype was implemented in Python to perform the append, verification, tampering, and checkpoint experiments. It uses a fixed compact UTF-8 JSON Lines encoding and SHA-256 through `hashlib`. The performance experiments were run in the Windows environment described below. These absolute timings are environment-specific; the security outcomes and asymptotic scaling are the primary experimental targets.

Troy University Resources

Table 6: Prototype implementation environment

Item	Configuration
Programming language	Python 3.13.3
Operating system	Windows-11-10.0.26200-SP0
Processor	Intel64 Family 6 Model 158 Stepping 10 (Intel)
Memory	25,093,856 kB
Hash function	SHA-256 via `hashlib` / OpenSSL 3.0.16 (11 Feb 2025)
Log-storage format	Fixed compact JSON Lines (JSONL), UTF-8
Serialization library	`orjson` 3.12.0 with standard-library fallback
Benchmark timer	`time.perf_counter_ns()`
Record-count benchmark payload size	64 bytes per event
Verification repetitions	10 for all dataset sizes

The implementation and evaluation pipeline use the components summarized in Table 7. This makes the relationship between the prototype, automated tests, measurements, and figures explicit.

Table 7: Software components and experimental roles

Tool or component	Role
Python 3.13.3	Prototype and experiment implementation
`hashlib` / OpenSSL	SHA-256 computation
`orjson`	Deterministic compact JSON serialization
JSON Lines (JSONL)	Sequential audit-record storage
`unittest`	Automated baseline and tampering tests
`perf_counter_ns()`	High-resolution runtime measurement
CSV and JSON	Machine-readable benchmark and metadata output
Matplotlib	Result visualization

13 Experimental Workflow

The workflow evaluates correctness, adversarial behavior, runtime, storage, and checkpoint trade-offs through the following reproducible sequence:

1. generate a legitimate hash-chain log;
2. verify the untampered baseline;

3. create independent controlled tampered copies;
4. test each copy with the local verifier;
5. test applicable cases with an independently held checkpoint;
6. run the checkpoint-window experiment;
7. measure isolated cryptographic append cost;
8. measure complete-chain verification time;
9. measure serialized storage and projected checkpoint overhead;
10. export CSV/JSON evidence and generate the report figures.

14 Dataset Sizes

Table 8: Experimental log sizes

Dataset	Number of Records
D1	100
D2	1,000
D3	10,000
D4	100,000
D5	1,000,000

15 Performance and Security Metrics

The following metrics are collected:

- average record-append time;
- median append time and standard deviation;
- full-chain verification time;
- linear-fit parameters and goodness of fit for verification;
- storage size;
- additional bytes required by hash-chain metadata;
- detection outcome for each tampering scenario.
- checkpoint bytes and writes per record;
- maximum unanchored record window.

Troy University Resources

16 Experimental Repetition and Statistical Analysis

Each runtime experiment was repeated multiple times.

For m measurements $t_1, \ldots, t_m$, the mean runtime is:

$$\bar{t} = \frac{1}{m} \sum_{j=1}^{m} t_j. \tag{22}$$

For sample standard deviation s, the reported approximate two-sided 95% normal confidence interval for a mean is

$$\bar{t} \pm 1.96 \frac{s}{\sqrt{m}}, \tag{23}$$

The benchmark implementation uses the same interval calculation. For the ten-run experiments it is only an approximation, not a distribution-free guarantee. Data generation is timed separately from append. Modification, deletion, insertion, and reordering are tested under both non-recomputing and recomputing attacker variants. Truncation, rollback, final-record rewriting, and whole-history replacement are evaluated separately according to their relevant attacker capabilities.

The record-count benchmark fixes the payload at 64 bytes. A second experiment holds the record count at 100,000 and varies payload size over 16, 64, 256, and 1,024 bytes. Exact serialized file sizes are recorded so that the dependence $T_{\text{verify}} = \Theta(\sum_i B_i)$ remains explicit. Checkpoint intervals of 1, 10, 100, 1,000, and 10,000 records are analyzed using the measured serialized checkpoint size to project the storage/security trade-off and maximum unanchored window.

Results and Discussion

1 Experimental Run Overview

A working prototype was implemented to test the security and complexity claims developed in the Methodology section. It performs application-specific deterministic compact JSON serialization, record appending, complete-chain verification, controlled tampering, and checkpoint verification. The executable implementation and automated tests are included with the supporting materials.

The authenticated preimage contains the encoding version, `log_id`, sequence number, timestamp, event fields, payload, and `previous_hash`. The field `current_hash` is excluded from its own preimage. Thus, `previous_hash` occurs exactly once in the encoded preimage and the implementation avoids a circular hash definition.

Experiments were executed under the Windows environment reported in Table 6. In the supporting package, `environment.json` records the machine-readable environment and `summary.json` records fitted statistics and final chain metadata. Absolute timings are machine-specific and should not be interpreted as deployment benchmarks; the primary claims concern scaling behavior and attack outcomes. All runtime tables and CSV files refer to this single Windows run.

Fourteen automated unit tests were executed and all 14 passed. The suite covers the untampered baseline and the principal modification, deletion, insertion, reordering, recomputation, truncation, rollback, and replacement cases. It also tests joint replacement of the log and an untrusted checkpoint.

2 Normal-Chain Verification

A single maximum-size JSONL chain of one million records was generated and its prefixes were used for datasets D1–D5. Every untampered prefix passed sequence, link, deterministic-encoding, and recomputed-hash checks.

Table 9: Observed baseline verification results

Dataset	Records	Observed Result
D1	100	Valid
D2	1,000	Valid
D3	10,000	Valid
D4	100,000	Valid
D5	1,000,000	Valid

This establishes functional correctness for the baseline case but does not by itself

Troy University Resources

establish historical authenticity. A newly generated attacker-controlled history can also satisfy the same local verification procedure.

3 Observed Tampering-Detection Results

The controlled attack suite used a 100-record baseline chain and a separately held trusted checkpoint for the authentic record-100 endpoint. The distinction between local consistency and historical authenticity was directly observable. Table 10 summarizes the principal runs.

Table 10: Observed tampering outcomes in the prototype

Case	Local Verify	Trusted Checkpoint	Observed Reason
Untampered baseline	Accept	Accept	All checks match
Modify middle, hashes retained	Reject	Reject	Hash mismatch at record 50
Modify middle, suffix rebuilt	Accept	Reject	Authentic checkpoint hash mismatch
Modify final record, hash retained	Reject	Reject	Record hash mismatch at record 100
Modify final record, hash rebuilt	Accept	Reject	Authentic checkpoint hash mismatch
Delete middle, hashes retained	Reject	Reject	Sequence inconsistency locally; checkpointed log is too short
Delete middle, suffix rebuilt	Accept	Reject	Log shorter than trusted checkpoint
Insert middle, hashes retained	Reject	Reject	Sequence inconsistency
Insert middle, suffix rebuilt	Accept	Reject	Authentic checkpoint hash mismatch
Reorder adjacent records, hashes retained	Reject	Reject	Sequence inconsistency
Reorder and rebuild suffix	Accept	Reject	Authentic checkpoint hash mismatch
Delete final 10 records	Accept	Reject	Log shorter than trusted checkpoint
Rollback to record 60	Accept	Reject	Log shorter than trusted checkpoint
Replace complete history	Accept	Reject	Authentic checkpoint hash mismatch
Replace history and checkpoint together	Accept	Accept	No independent trust remains

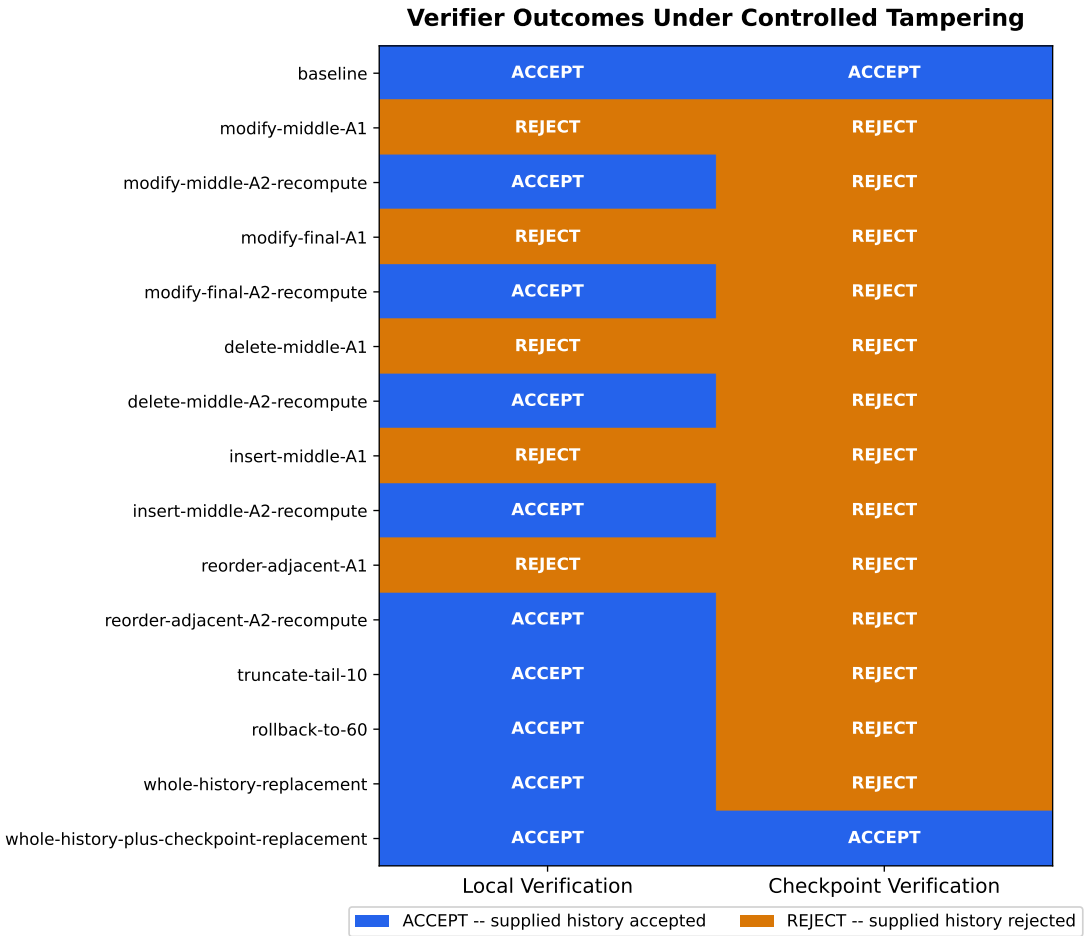


Figure 2: Local and checkpoint-aware verification outcomes for controlled attack scenarios.

These results experimentally confirm the security boundary predicted in the threat model. A1-style modifications that preserve the original cryptographic values are detected by local verification. In contrast, an A2 adversary can rebuild the affected suffix and obtain another internally valid chain without finding a SHA-256 collision. The checkpoint distinguishes the replacement history only because its authentic value is supplied independently of the attacker-controlled file. This is consistent with the distinction between post-hoc tamper evidence and stronger secure-logging guarantees described in the literature [1, 3].

4 Modification, Insertion, Deletion, and Reordering

For an unrepaired modification at record i, let the changed event data be represented by R'_i. The verifier recomputes

$$\begin{aligned} M'_i &= \mathrm{Encode}(\ldots, R'_i, \ldots, H_{i-1}), \\ H'_i &= \mathrm{SHA256}(M'_i). \end{aligned} \tag{24}$$

and observes $H'_i \neq H_i$. In the test suite, modification of record 50 was therefore rejected exactly at record 50. Interior deletion, insertion, and reordering similarly caused

Troy University Resources

either a sequence-number error or a predecessor-link error before the original suffix could be accepted.

After suffix reconstruction, however, each of these manipulated histories was accepted by the local verifier. This observation is important: order sensitivity of a recursive chain guarantees consistency relative to the supplied endpoint, not uniqueness of the historical sequence.

5 Truncation, Rollback, and Whole-History Replacement

Tail truncation behaved differently from interior deletion. Removing the final 10 records left records 1–90 unchanged, so the remaining file was already a valid chain and local verification accepted it without any recomputation. Rollback to an authentic 60-record snapshot behaved in the same way. These observations reproduce the well-known truncation problem in chained secure logging [3].

Whole-history replacement was also accepted locally after the attacker generated new hashes from the genesis value. Again, no collision was required. The attack failed only when the verifier compared the reconstructed endpoint with an independently retained authentic checkpoint. When the experiment deliberately replaced the checkpoint together with the log, verification succeeded. This provides direct experimental evidence that a checkpoint stored under the same attacker control does not create an independent trust anchor.

6 Checkpoint-Window Experiment

A separate experiment used 250 records with a trusted checkpoint at record 200, leaving 50 records in the current unanchored suffix. Four attacks were then executed:

Table 11: Observed checkpoint-boundary behavior with a checkpoint at record 200

Experiment	Local Verify	Checkpoint Verify
Modify record 167 and rebuild the suffix	Accept	Reject
Modify record 225 and rebuild the suffix	Accept	Accept
Truncate records 201–250	Accept	Accept
Rollback to record 199	Accept	Reject

The result shows the checkpoint boundary directly. Changes to a committed prefix are detectable, whereas changes confined to records after the latest checkpoint remain outside the historical commitment. Consequently, if a checkpoint is successfully published every m records, the worst-case unanchored interval is

$$W_{\max} = m - 1. \tag{25}$$

The representative serialized checkpoint used by the prototype contains `log_id`, in-

dex, chain hash, record timestamp, and encoding version and occupies 182 bytes for the measured example. Because textual length may vary with fields such as the decimal index and timestamps, Table 12 is a projection that assumes a constant 182-byte checkpoint; it is not a measurement obtained by writing one million checkpoint objects. The figures exclude transport framing, digital signatures, TLS, and remote-service attestation metadata.

Table 12: Projected checkpoint-storage trade-off assuming a 182-byte serialized checkpoint

m	Max. Unanchored	Checkpoint Writes	Projected Bytes
1	0	1,000,000	182,000,000
10	9	100,000	18,200,000
100	99	10,000	1,820,000
1,000	999	1,000	182,000
10,000	9,999	100	18,200

Thus checkpoint frequency is not merely a performance parameter: it defines an explicit bound on the recent history not yet covered by an external commitment. Freshness remains essential; an old but authentic checkpoint cannot prove that no later state previously existed. External history mechanisms such as timeline entanglement and authenticated history trees address related historical consistency problems [5, 6].

7 Append-Time Evaluation

The append microbenchmark isolates record construction, deterministic compact JSON serialization, and one SHA-256 evaluation while the previous chain value is already available. It intentionally excludes durable disk synchronization such as `fsync()`. Each reported value is the mean of 30 batches of 1,000 append computations.

Table 13: Measured cryptographic append cost with direct access to the chain head

Prior Records	Mean (μs)	Median (μs)	Std. Dev. (μs)	Approx. 95% CI (μs)
100	4.212	4.138	0.189	[4.145, 4.280]
1,000	4.331	4.253	0.234	[4.247, 4.414]
10,000	4.268	4.212	0.194	[4.199, 4.337]
100,000	4.399	4.334	0.229	[4.317, 4.481]
1,000,000	4.647	4.692	0.256	[4.555, 4.739]

The intervals use Eq. (23) over the 30 batch means.

The measured means remain within approximately 4.21–$4.65\,\mu$s across sequence positions corresponding to 100 through one million prior records, with the predecessor hash supplied directly. A linear fit against the supplied prior-record count used to set the sequence

Troy University Resources

position gives a slope of approximately 3.63×10^{-7} μs per prior record with $R^2 = 0.880441$. The fit shows a small environment-specific upward drift, but not growth proportional to the sequence position. The measurements are consistent with the predicted $O(1)$ dependence on prior-record count for the isolated append computation; the asymptotic result follows from the algorithmic structure rather than from the benchmark alone. It does not imply that a file implementation that scans or rewrites the complete log has $O(1)$ end-to-end latency.

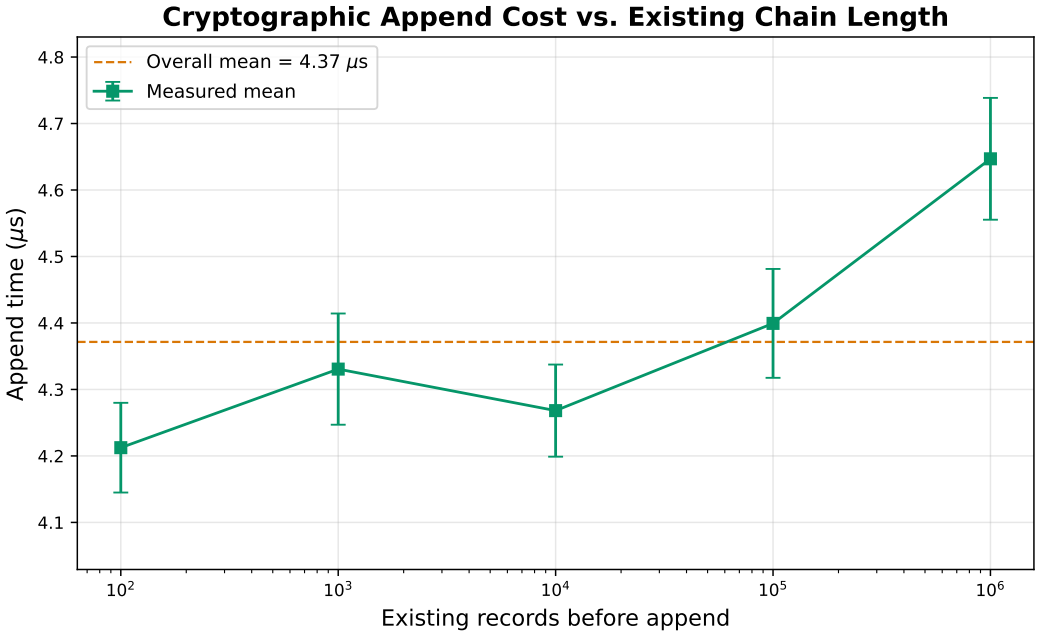


Figure 3: Measured append cost with the current chain head supplied directly; error bars show approximate 95% normal confidence intervals.

8 Full-Chain Verification Evaluation

Full verification was measured using the same deterministic JSONL chain. A warm-up scan was excluded for each dataset. All datasets were measured 10 times each.

Table 14: Observed full-chain verification performance

Records	Runs	Mean (s)	Std. Dev. (s)	Approx. 95% CI (s)	Records per second
100	10	0.001049	0.000096	[0.000990, 0.001108]	95,330
1,000	10	0.005205	0.000146	[0.005115, 0.005296]	192,110
10,000	10	0.048076	0.000676	[0.047657, 0.048495]	208,005
100,000	10	0.462514	0.008010	[0.457549, 0.467479]	216,210
1,000,000	10	4.537667	0.065352	[4.497161, 4.578172]	220,378

Least-squares fitting of the mean measurements gives

$$\widehat{t}(n) = 0.003101 + 4.535 \times 10^{-6} n \quad \text{seconds}, \tag{26}$$

with

$$R^2 = 0.999997. \tag{27}$$

The fitted slope corresponds to approximately 220,500 records/s. The one-million-record file occupies 416,888,896 bytes, so the fitted large-n rate corresponds to roughly 92 MB/s of protected JSONL data on this execution environment. The high R^2 is empirical evidence consistent with linear scaling; it is not by itself a mathematical proof of $\Theta(n)$. The asymptotic result follows from the algorithm, which must decode and hash every record.

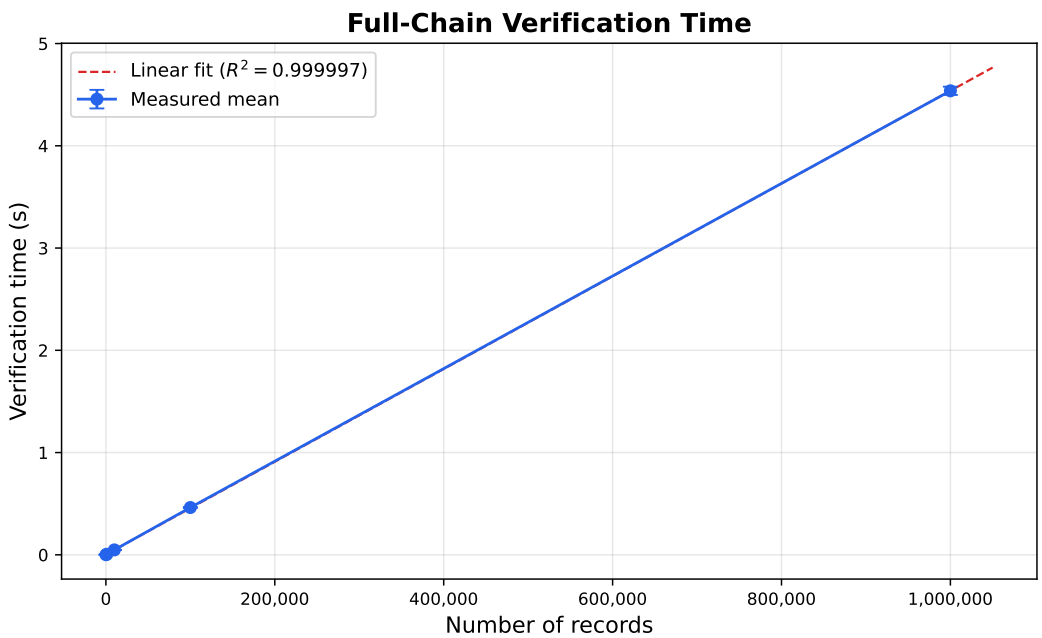


Figure 4: Measured full-chain verification time and least-squares linear fit; error bars show approximate 95% normal confidence intervals.

The confidence intervals in Table 14 use Eq. (23). The smallest dataset has lower throughput because interpreter, timer, scheduling, and file-opening overheads are large relative to the millisecond-scale useful work. At larger sizes these fixed costs are amortized, and throughput rises to approximately 192–220 thousand records/s.

9 Variable-Payload Verification

To test the encoded-size term directly, a second experiment held the record count at 100,000 and varied the synthetic payload from 16 to 1,024 bytes. Each configuration was measured 10 times after an excluded warm-up scan.

Troy University Resources

Table 15: Observed verification performance for 100,000 records with variable payloads

Payload (bytes)	Protected (MiB)	Mean (s)	Std. Dev. (s)	Approx. 95% CI (s)	Records per second
16	35.08	0.439954	0.009958	[0.433782, 0.446126]	227,297
64	39.66	0.454480	0.010905	[0.447721, 0.461239]	220,032
256	57.97	0.521177	0.014799	[0.512005, 0.530350]	191,873
1,024	131.21	0.792787	0.017578	[0.781892, 0.803682]	126,137

A least-squares fit against protected size S in MiB gives

$$\widehat{t}(S) = 0.309006 + 0.003684S \quad \text{seconds}, \tag{28}$$

with $R^2 = 0.999933$. These four measurements show a strong size-dependent trend consistent with the $\Theta(\sum_i B_i)$ analysis. They corroborate the encoded-size model but do not, by themselves, prove an asymptotic bound.

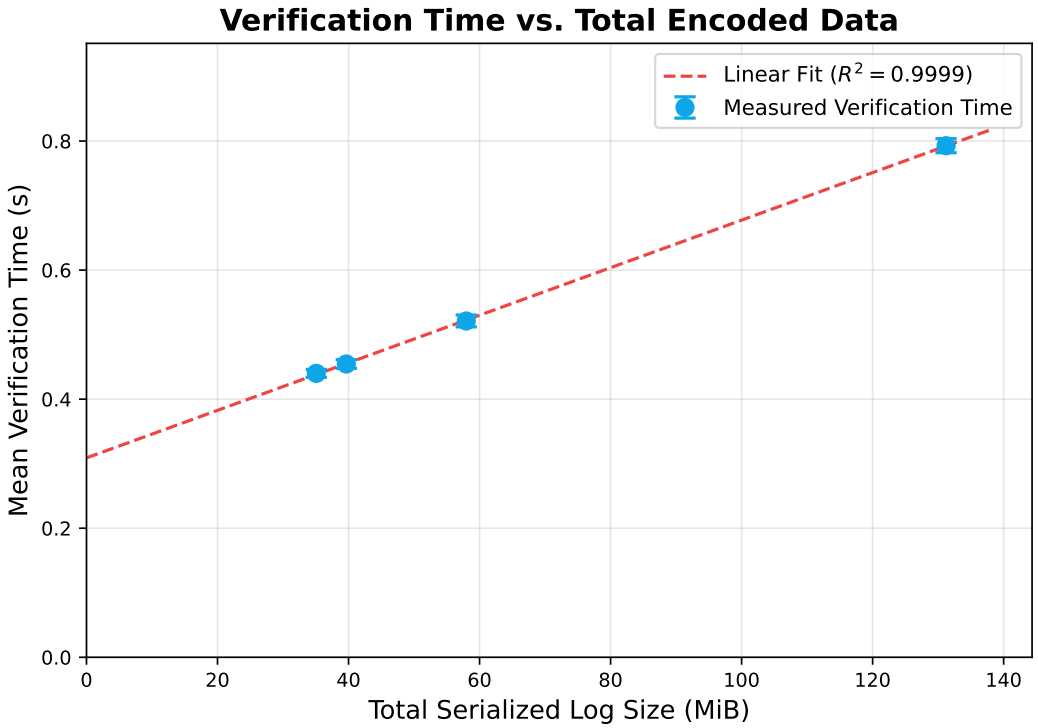


Figure 5: Verification time versus total serialized log size for a fixed 100,000-record dataset.

10 Storage-Overhead Evaluation

Storage was measured from the exact serialized UTF-8 JSONL representation rather than only from the theoretical 32-byte SHA-256 digest size. The protected format stores both predecessor and current hash fields. A second representation stores only the current hash, allowing the redundancy to be quantified directly.

Table 16: Measured serialized storage overhead

Records	Raw Bytes	Protected Bytes	Overhead	Current-Only Overhead
100	24,792	41,292	66.55%	33.08%
1,000	248,893	413,893	66.29%	32.95%
10,000	2,498,894	4,148,894	66.03%	32.81%
100,000	25,088,895	41,588,895	65.77%	32.68%
1,000,000	251,888,896	416,888,896	65.51%	32.55%

At one million records, the protected file is approximately 397.6 MiB versus 240.2 MiB for the raw JSONL representation. Removing the redundant stored `previous_hash` would reduce the protected representation to 333,888,896 bytes (approximately 318.4 MiB). In this JSON representation, the redundant field costs 83 bytes per record, or 83,000,000 bytes at one million records; the value is larger than 64 bytes because the field name, quotation marks, colon, and comma are also serialized.

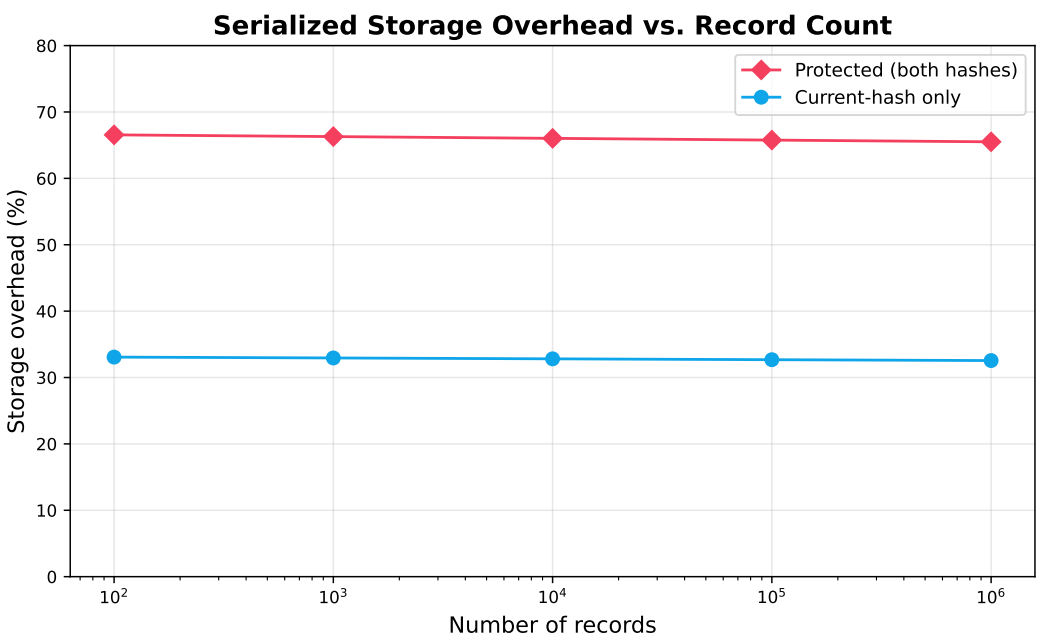


Figure 6: Measured JSONL overhead with two stored hashes versus current-hash only.

This result illustrates why theoretical digest size and actual file overhead must be reported separately.

11 Comparison with Theoretical Complexity

The measurements agree with the asymptotic analysis when each operation is matched to the correct input model. The isolated append path performs one bounded-size serialization and one hash, so its cost is approximately constant in the existing record count. Complete verification processes every encoded record and grows approximately linearly.

Troy University Resources

For records with encoded lengths B_i, the more precise statement remains

$$T_{\text{verify}} = \Theta\left(\sum_{i=1}^{n} B_i\right), \tag{29}$$

which becomes $\Theta(n)$ only when record size is bounded or approximately constant. The fixed-payload experiment isolates record-count scaling, while the variable-payload experiment shows that verification time also increases with total encoded size. A production logger should therefore report both record count and total serialized bytes.

The experiment also separates cryptographic cost from system cost. File scans, JSON decoding, memory allocation, operating-system page caching, synchronous disk writes, and storage-device behavior can dominate an end-to-end logger even though one SHA-256 append remains constant in the number of prior records.

12 Security Properties and Limitations

The observed results support five distinct conclusions.

1. A plain recursive SHA-256 chain is effective at detecting unrepaired modification and structural corruption.

2. Hash recomputation is not a collision attack. An adversary that can replace records and locally stored hashes can construct another valid chain by evaluating SHA-256 normally.

3. Tail truncation and rollback are particularly important because a retained authentic prefix may require no cryptographic recomputation.

4. An independently protected checkpoint converts a previously stored chain value into a historical reference and detects changes to the committed prefix.

5. Checkpoint protection is only as strong as its independence and freshness. Joint replacement of the log and checkpoint succeeded in the controlled experiment.

These findings are consistent with secure-logging work that augments hash chaining with forward-secure authentication, trusted verification state, or externally auditable history mechanisms [1, 3, 6].

The experiment does not claim protection against A3 full compromise or A4 logger equivocation. The prototype contains no evolving authentication keys, public-key signatures, confidentiality mechanism, distributed witnesses, or fault-tolerant aggregate authentication. The advanced constructions reviewed in the Methodology section therefore remain comparison points rather than implemented features.

13 Comparison with Secure-Logging Literature

Table 17 positions the empirical baseline against representative secure-logging mechanisms. The purpose of this comparison is not to claim equivalent security, but to identify which guarantees are absent from the simple chain.

Table 17: Experimental baseline compared with representative secure-logging approaches

Approach	Additional Capability	Relation to This Experiment
Plain hash chain	Order-sensitive local consistency	Implemented and measured; vulnerable to recomputation, truncation, rollback, and whole-history replacement without an anchor
Schneier–Kelsey [1]	Forward-secure authentication and trusted verification architecture	Addresses stronger compromise assumptions than the unkeyed prototype
Ma–Tsudik [3, 7]	Forward-secure stream integrity and sequential aggregation	Protects pre-compromise history using evolving secret state; not implemented here
History tree [6]	Compact membership and historical-consistency proofs	Reduces proof size for selective auditing but requires authenticated commitments
Timeline entanglement [5]	Cross-system historical commitments	Provides an external-history mechanism relevant to rollback/equivocation
Robust aggregate logging [11]	Forward security plus bounded corruption tolerance and truncation protection	Stronger robustness at substantially greater cryptographic complexity

14 Answers to the Research Questions

RQ1 — Append Efficiency

The measurements are consistent with the predicted $O(1)$ dependence on prior-record count for the isolated append computation; the asymptotic result follows from the algorithmic structure rather than from the benchmark alone. Mean cost ranged from 4.21 to $4.65\,\mu$s with the predecessor hash supplied directly. This does not establish $O(1)$ durable file-update cost.

RQ2 — Verification Complexity

The fixed-64-byte experiment measured approximately 0.001049 s at 100 records and 4.538 s at one million records. Its linear fit has $R^2 = 0.999997$, strongly corroborating $\Theta(n)$ scaling for bounded-size records. In the 100,000-record experiment, payloads from 16 to 1,024 bytes produced a fit against protected serialized size with $R^2 = 0.999933$. Together, the experiments are consistent with the general bound $\Theta(\sum_i B_i)$.

RQ3 — Detection of Tampering

Under the non-recomputing A1 model, modification, interior deletion, insertion, and reordering were all rejected. The verifier reported the first sequence, link, or hash inconsistency. Tail

Troy University Resources

truncation was an important exception because the surviving prefix remained locally valid.

RQ4 — Attacker Recomputation

When the attacker recomputed the affected suffix, modification, insertion, deletion, reordering, final-record rewriting, and whole-history replacement all became locally valid. Truncation and rollback required no recomputation at all. None of these successful local attacks required a SHA-256 collision.

RQ5 — Trusted Checkpoints

A separately trusted checkpoint detected every tested manipulation affecting its committed prefix and detected rollback to a state shorter than the latest checkpoint. It did not detect a recomputed modification confined to the unanchored suffix, nor truncation that removed only records after the latest checkpoint. The experiment therefore confirms the checkpoint trade-off $W_{\max} = m-1$ under successful periodic publication. Replacing both the log and an untrusted checkpoint also succeeded, showing why checkpoint independence and freshness are necessary. In particular, the record timestamp copied into the prototype checkpoint is not an independent publication time. A deployed external service must authenticate freshness metadata that the storage attacker cannot rewrite.

15 Research Contributions

The completed study adds an empirical component to the analytical model developed earlier in the report. Its main contributions are:

1. an executable deterministic SHA-256 hash-chain prototype with complete append, verification, tampering, and checkpoint logic;
2. a 14-test automated verification suite, with all tests passing, together with a controlled evaluation covering 14 attack scenarios plus the untampered baseline;
3. direct experimental confirmation that recomputed suffixes, truncation, rollback, and whole-history replacement do not require breaking SHA-256;
4. runtime measurements up to one million records, including a linear verification fit with $R^2 = 0.999997$ and a separate variable-payload fit with $R^2 = 0.999933$;
5. exact serialized storage measurements showing approximately 65.5% overhead for the two-hash JSONL representation at one million records and approximately 32.6% when the redundant previous-hash field is removed;

6. a checkpoint-boundary experiment that directly demonstrates the anchored/unanchored security boundary, together with a checkpoint-interval analysis that projects storage overhead as a function of m.

16 Limitations of the Experimental Study

The empirical findings should be interpreted with the following limitations:

- the datasets are synthetic; the record-count benchmark uses a fixed 64-byte payload, while the variable-payload experiment tests only four sizes from 16 to 1,024 bytes;
- the benchmark represents one experimental run in a Windows environment rather than a broad sample of production hardware;
- verification measurements are warm-cache measurements after an excluded warm-up pass;
- the append microbenchmark excludes durable disk synchronization and therefore isolates serialization plus hashing rather than full storage latency;
- each verification dataset was measured 10 times;
- the prototype uses only unkeyed SHA-256 and does not implement forward-secure keys, digital signatures, confidentiality, or public verification;
- the checkpoint is modeled as an independently trusted object, but no remote checkpoint service, digital-signature infrastructure, monotonic counter, or trusted publication-time authority was implemented;
- the experiment does not implement A4 equivocation between multiple independent auditors.

These limitations affect absolute performance and the strength of the deployed trust model, but they do not invalidate the observed distinction between local chain consistency and externally anchored historical consistency.

17 Conclusion and Future Work

The experiments confirm that a sequential SHA-256 chain is a simple and computationally efficient mechanism for detecting unrepaired changes in an ordered audit log. Full verification scales linearly with log size, while the isolated append computation remains approximately constant when the previous chain state is directly available.

The central security result is negative but important: a locally valid chain is not proof of the original history. An attacker that controls mutable storage can recompute a modified suffix, replace the final record, truncate a valid suffix, restore an older authentic snapshot, or generate a completely new chain without breaking SHA-256. An independent checkpoint detects only changes that contradict its committed prefix; recent records after the checkpoint remain unanchored.

Future work should therefore move from an unkeyed local chain toward mechanisms that protect historical state independently of the compromised logger. Relevant directions include forward-secure key evolution, remotely or publicly anchored checkpoints, authenticated history trees, transparency-style witnesses, and robust forward-secure aggregate authentication [1, 6, 7, 11].

Troy University Resources

References

[1] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security*, vol. 2, no. 2, pp. 159–176, May 1999.

[2] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991.

[3] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage*, vol. 5, no. 1, pp. 1–21, 2009.

[4] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, "Building an encrypted and searchable audit log," in *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[5] P. Maniatis and M. Baker, "Secure history preservation through timeline entanglement," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[6] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *18th USENIX Security Symposium*. USENIX Association, 2009, pp. 317–334.

[7] D. Ma and G. Tsudik, "Extended abstract: Forward-secure sequential aggregate authentication," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007, pp. 86–91.

[8] A. A. Yavuz and P. Ning, "BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems," in *Proceedings of the Annual Computer Security Applications Conference*, 2009, pp. 219–228.

[9] P. Kampanakis and A. A. Yavuz, "BAFi: A practical cryptographic secure audit logging scheme for digital forensics," *Security and Communication Networks*, vol. 8, pp. 3180–3190, 2015.

[10] G. Hartung, "Secure audit logs with verifiable excerpts," in *Proceedings of CT-RSA*, 2016.

[11] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann, "Practical and robust secure logging from fault-tolerant sequential aggregate signatures," in *Provable Security*, ser. Lecture Notes in Computer Science, vol. 10592. Springer, 2017, pp. 87–106.

[12] J. Kim and H. Oh, "FAS: Forward secure sequential aggregate signatures for secure logging," *Information Sciences*, vol. 471, pp. 115–131, 2019.

[13] G. Hartung, "Attacks on secure logging schemes," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, vol. 10322. Springer, 2017, pp. 268–284.

[14] G. A. Marson and B. Poettering, "Practical secure logging: Seekable sequential key generators," in *Proceedings of ESORICS*, 2013.

[15] A. A. Yavuz, P. Ning, and M. K. Reiter, "Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, vol. 7397. Springer, 2012, pp. 148–163.

[16] A. Rundgren, B. Jordan, and S. Erdtman, "JSON canonicalization scheme (JCS)," RFC Editor, RFC 8785, Jun. 2020.

[17] National Institute of Standards and Technology, "Secure hash standard (shs)," U.S. Department of Commerce, Federal Information Processing Standards Publication 180-4, Aug. 2015.

[18] D. Ma, "Practical forward secure sequential aggregate signatures," in *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ACM, 2008, pp. 341–352.

[19] E. Rich, *Automata, Computability and Complexity: Theory and Applications*. Pearson, 2007.

[20] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

Troy University Resources

Appendices

Appendix A. Project Work Plan

This appendix shows the team's completed work plan, including the main tasks, assigned members, and project outcomes. All activities were finished on schedule, and individual contributions are summarized at the beginning of the report.

Table 18: Completed project work plan and status

Objective	Activities / Tasks	Results	Who	When	Status
Research scope	Review secure-logging literature; define the tampering problem, requirements, objectives, and research questions.	Approved scope, problem statement, requirements, objectives, and questions.	[S1], [S2]	23–24 Jul	Complete
Security model	Define trust assumptions, A1–A4 adversaries, and modification, deletion, insertion, reordering, recomputation, rollback, truncation, and replacement attacks.	Threat model and attack-case matrix.	[S1], [S3]	25–26 Jul	Complete
Formal model	Specify record fields, deterministic encoding, sequence numbers, the recursive SHA-256 relation, verification rules, and trusted checkpoints.	Formal chain and checkpoint specification.	[S2], [S1]	27–29 Jul	Complete
Security analysis	Analyze local consistency, authenticity, recomputation, rollback, truncation, forward security, and external commitments.	Security guarantees, limitations, and literature comparison.	[S2], [S3]	30 Jul–2 Aug	Complete
Prototype design	Define the JSONL format, append and verification procedures, checkpoint handling, and experimental workflow.	Prototype architecture and implementation specification.	[S4], [S1]	3–5 Aug	Complete
Append implementation	Implement deterministic record construction, chain-head retrieval, SHA-256 computation, metadata storage, and dataset generation.	Working append implementation and reproducible datasets.	[S4]	6–8 Aug	Complete

Continued on next page

Table 18 continued

Objective	Activities / Tasks	Results	Who	When	Status
Verifier implementation	Check log identifiers, sequence numbers, predecessor links, encoding versions, record hashes, and independently supplied checkpoints.	Local and checkpoint-aware verifier.	[S4], [S3]	9–11 Aug	Complete
Tampering suite	Implement modification, deletion, insertion, reordering, suffix rebuilding, truncation, rollback, and history-replacement cases.	Automated attack suite and controlled datasets.	[S3], [S1]	12–14 Aug	Complete
Tampering evaluation	Run the attack suite; record local and checkpoint outcomes and first failing conditions; compare A1 and recomputing attackers.	Attack matrix, checkpoint-boundary results, and security interpretation.	[S3], [S2]	15–16 Aug	Complete
Performance evaluation	Measure append, full verification, variable payloads, storage overhead, and checkpoint trade-offs; compare the results with theory.	Benchmarks, fitted models, storage data, and plots.	[S2], [S1]	17–20 Aug	Complete
Integration and validation	Cross-check the model, code, tests, attacks, benchmarks, figures, and conclusions; resolve inconsistencies.	Validated implementation and experimental evidence.	All members	21 Aug	Complete
Final report	Consolidate the report sections, references, appendices, source-code excerpts, and experimental evidence.	Final report and supporting documentation.	All members	22 Aug	Complete
Presentation and submission	Prepare slides, the demonstration, source/data package, tests, contribution records, and submission archive; conduct the final review.	Submission-ready report, presentation, demonstration, and project archive.	All members	23 Aug	Complete

Troy University Resources

Appendix B. Advanced Secure-Logging Constructions

This appendix preserves the mathematical detail behind the concise secure-logging comparison in the Methodology section. It is included for academic depth and future design work; none of these public-key constructions is claimed as part of the implemented SHA-256 prototype.

B.1 BLS-Based FssAgg

The BLS-based construction of Ma and Tsudik uses prime-order groups G_1, G_2, G_T and a bilinear map

$$e : G_1 \times G_2 \to G_T, \qquad e(u^a, v^b) = e(u, v)^{ab}. \tag{30}$$

Let g_2 generate G_2, and let $H_1 : \{0,1\}^* \to G_1$ be a full-domain hash. Starting from $x_0 \in_R \mathbb{Z}_q$, epoch secrets and public keys evolve as

$$x_i = H_0(x_{i-1}), \qquad v_i = g_2^{x_i}, \tag{31}$$

so the original verification key is

$$PK = (v_1, \ldots, v_T). \tag{32}$$

and an epoch-i signature is

$$\sigma_i = H_1(i \parallel M_i)^{x_i}. \tag{33}$$

The aggregate through epoch t is the group product

$$\sigma_{1,t} = \prod_{i=1}^{t} \sigma_i, \tag{34}$$

which is verified by checking

$$e(\sigma_{1,t}, g_2) \stackrel{?}{=} \prod_{i=1}^{t} e(H_1(i \parallel M_i), v_i)\,. \tag{35}$$

The storage profile is therefore

$$S_{\text{sig}} = O(1), \qquad S_{\text{sk}} = O(1), \qquad S_{\text{pk}} = O(T). \tag{36}$$

The aggregate and current secret state are constant-sized, but verification evaluates a history-dependent number of pairing terms. Thus the design reduces authentication storage at the signer without making complete verification constant-time [7].

B.2 Factoring-Based FssAgg

A second FssAgg family uses repeated squaring modulo a Blum integer $N = pq$, where $p \equiv q \equiv 3 \pmod 4$. Its secret components evolve by

$$S_{i,t} = S_{i,t-1}^2 \pmod N, \qquad r_t = r_{t-1}^2 \pmod N, \tag{37}$$

and each epoch contributes a value

$$(c_{1,t}, \ldots, c_{\ell,t}) = H(t, y, M_t) \tag{38}$$

whose challenge bits select the secret components in

$$z_t = r_t \prod_{i=1}^{\ell} S_{i,t}^{c_{i,t}} \pmod{N} \tag{39}$$

to the multiplicative aggregate

$$\sigma_{1,t} = \sigma_{1,t-1} z_t \pmod{N}. \tag{40}$$

Verification repeatedly squares the aggregate state while incorporating public values derived during setup. A representative recurrence begins with

$$W_t = \sigma_{1,t}^{2^{T+1-t}} y \prod_{i=1}^{\ell} u_i^{c_{i,t}} \pmod{N} \tag{41}$$

and applies analogous squaring steps for the preceding epochs. This connects one-way key evolution to a verifier that reconstructs the authenticated epoch history.

These constructions are historically relevant because they illustrate how one-way state evolution can be combined with aggregation. Later cryptanalysis, however, found practical known-message attacks against BM-FssAgg and related variants. They are therefore comparison points rather than recommended extensions for the prototype [12, 13, 18].

B.3 LogFAS

LogFAS combines Schnorr-style responses with incremental hashing to reduce the number of expensive public-key operations at verification [15]. Let $q \mid (p-1)$ and let α generate a subgroup of order q in $\mathbb{Z}_p^*$. A long-term value b defines

$$B = \alpha^{b^{-1}} \pmod{p}. \tag{42}$$

For epoch j, auxiliary random values determine

$$y_j = a_j - d_j \pmod{q}, \tag{43}$$
$$z_j = (a_j - x_j) b \pmod{q}, \tag{44}$$
$$R_j = \alpha^{r_j} \pmod{p}, \tag{45}$$
$$M_j = \alpha^{x_j - d_j} \pmod{p}. \tag{46}$$

An authenticated counter token

$$\beta_j = \mathsf{Sign}_{sk_b}(H(ID \parallel j)) \tag{47}$$

binds the signer identity to the epoch. For record D_ℓ, LogFAS derives a challenge

$$e_\ell = H(D_\ell \parallel \ell \parallel z_\ell \parallel R_\ell) \tag{48}$$

and response

$$s_\ell = r_\ell - e_\ell y_\ell \pmod{q}. \tag{49}$$

Responses and auxiliary group values are accumulated as

Troy University Resources

$$s_{0,\ell} = s_{0,\ell-1} + s_\ell \pmod q, \tag{50}$$
$$M'_{0,\ell} = M'_{0,\ell-1} M'_\ell \pmod p, \tag{51}$$
$$z_{0,\ell} = \sum_{j=0}^{\ell} e_j z_j \pmod q. \tag{52}$$

After validating an epoch token, the verifier checks an equation of the form

$$\prod_{j=0}^{\ell} R_j \stackrel{?}{\equiv} M'_{0,\ell} B^{z_{0,\ell}} \alpha^{s_{0,\ell}} \pmod p. \tag{53}$$

The claimed efficiency concerns expensive exponentiations, not total running time:

$$T_{\text{verify}} = O(1) \text{ ExpOps} + O(L) \text{ hash/basic operations}. \tag{54}$$

This distinction is relevant to the present experiment, where complete verification is also linear in encoded input size even though each individual SHA-256 operation is bounded.

B.4 Cryptanalytic Details

Hartung showed that several proposed aggregate-logging schemes did not meet their intended guarantees under the stated verification equations [13]. In the LogFAS analysis, an attacker can recompute the incremental challenge for a replacement sequence and choose the aggregate component

$$M'^{*}_{0,\ell} = R_{0,\ell} B^{-z^{*}_{0,\ell}} \alpha^{-s_{0,\ell}} \pmod p, \tag{55}$$

so that Eq. (53) still holds. Related attacks recover signing material from suitable BM-FssAgg and AR-FssAgg aggregates.

These attacks show that security claims depend on the exact construction and adversary model.

B.5 Fault-Tolerant Forward-Secure Aggregation

Later robust logging work uses fault-tolerant forward-secure sequential aggregate signatures to retain useful evidence even when some log positions are corrupted [11]. A d-cover-free family $\mathcal{F} = (S, \mathcal{B})$ satisfies

$$B \nsubseteq B_1 \cup \cdots \cup B_d \tag{56}$$

for every distinct $B, B_1, \ldots, B_d \in \mathcal{B}$. Its incidence matrix is

$$M[i,j] = \begin{cases} 1, & i \in B_j, \\ 0, & i \notin B_j. \end{cases} \tag{57}$$

The matrix selects which components of an aggregate vector

$$\tau = (\tau[1], \ldots, \tau[r]) \tag{58}$$

authenticate each claim. A claim c_j is incorporated into component $\tau[i]$ only when $M[i,j] = 1$. If at most d positions are corrupted, at least one authentication component remains available for each unaffected claim. A fault-tolerant verifier may consequently return

$$V[j] = \begin{cases} C[j], & \text{if claim } j \text{ remains authenticated}, \\ \perp, & \text{otherwise}. \end{cases} \tag{59}$$

This is stronger than all-or-nothing aggregate verification because a bounded number of damaged records need not destroy the forensic value of every unaffected record.

Robust schemes also authenticate the current log length with evolving forward-secure state,

$$s_i \leftarrow \mathsf{FS.Sign}(sk_{\mathrm{FS}}, i), \tag{60}$$

so rollback to an unauthenticated length requires a forgery. The construction therefore combines three guarantees absent from the prototype:

- protection of pre-compromise records through key evolution;
- bounded tolerance of corrupted records;
- explicit authentication of log length against truncation.

These benefits require substantially more cryptographic machinery than the plain hash chain measured in the Results and Discussion section.

Troy University Resources

Appendix C. History-Tree Technical Details

This appendix retains the indexing, versioned hash recurrence, frozen-node condition, and incremental-proof procedures underlying the conceptual comparison in the History Trees and External Commitments subsection of the Methodology section.

C.1 Tree Indexing

A version-n history contains

$$X_0, \ldots, X_n.$$

Each node is identified as

$$I_{i,r},$$

where i is the index of the first event represented by the node and r is its layer.
At layer $r = 0$, node $I_{i,0}$ contains event X_i.
An interior node $I_{i,r}$ has children

$$I_{i,r-1}$$

and

$$I_{i+2^{r-1},r-1}.$$

This indexing gives each complete subtree a stable range and layer, allowing its root to be reused when later versions append events.

C.2 Versioned Hash Recurrence

Let

$$A^v_{i,r}$$

denote the hash assigned to node $I_{i,r}$ when reconstructing the version-v view.
For a leaf,

$$A^v_{i,0} = H(0 \parallel X_i), \qquad v \geq i. \tag{61}$$

For an interior node,

$$A^v_{i,r} = \begin{cases} H\left(1 \parallel A^v_{i,r-1}\right), & v < i + 2^{r-1}, \\ H\left(1 \parallel A^v_{i,r-1} \parallel A^v_{i+2^{r-1},r-1}\right), & v \geq i + 2^{r-1}. \end{cases} \tag{62}$$

The commitment to a version-n tree of depth d is

$$C_n = A^n_{0,d}. \tag{63}$$

The depth required for $n + 1$ events is

$$d = \lceil \log_2(n + 1) \rceil . \tag{64}$$

C.3 Frozen Hashes

Once a subtree becomes complete, its root hash never changes when new events are appended.

Node $I_{i,r}$ becomes frozen when

$$n \geq i + 2^r - 1. \tag{65}$$

Its hash may therefore be cached as

$$FH_{i,r}.$$

In the history-tree construction,

$$A^v_{i,r} \equiv FH_{i,r} \tag{66}$$

whenever the node is frozen for the requested view.

The cache enables old commitments to be reconstructed without rehashing the complete log [6].

C.4 Incremental Proof Generation

The operation

$$\mathsf{H.INCR.GEN}(C_i, C_j) \to P$$

produces a pruned version-j history tree that contains sufficient information to reconstruct both the older commitment C_i and the newer commitment C_j.

The proof begins with the frozen hashes needed for the newer root and expands only the nodes on paths required to reconstruct the older commitment. Complete subtrees unrelated to those paths remain represented by one frozen hash.

C.5 Incremental Proof Verification

Given a proof P and two externally supplied commitments C'_i and C'_j, verification reconstructs the two corresponding views:

$$\widehat{C_i} = A^i_{0,d}, \qquad \widehat{C_j} = A^j_{0,d}. \tag{67}$$

The incremental proof is accepted exactly when

$$\widehat{C_i} = C'_i \tag{68}$$

and

$$\widehat{C_j} = C'_j. \tag{69}$$

The important security interpretation is that the proof does not merely establish that the current snapshot is internally well-formed. It establishes that the newer commitment makes the same claims about the shared historical prefix as the older commitment.

History trees therefore provide a useful contrast with the plain hash-chain baseline studied in this project. Classic hash chains require every intermediate event for an incremental or membership proof, whereas history trees use tree paths and pruned subtrees to provide substantially more compact proofs [6].

Troy University Resources

Appendix D. Additional Theory-of-Computation Formalisms

The main methodology models a finite audit log as a formal-language input and the verifier as a deterministic decision procedure. This appendix preserves the broader infinite-trace and program-checking formalisms.

D.1 Büchi Automata

A continuously operating logging system is naturally modeled as a reactive system whose execution may be infinite.

A nondeterministic Büchi automaton is a tuple

$$\mathcal{B} = (Q, \Sigma, \delta, q_0, F), \tag{70}$$

where

- Q is a finite set of states;
- Σ is an alphabet;
- δ is the transition relation;
- q_0 is the initial state;
- $F \subseteq Q$ is the set of accepting states.

Unlike an ordinary finite automaton, a Büchi automaton operates on infinite words

$$w \in \Sigma^{\omega}.$$

An infinite run is accepting if it visits at least one state in F infinitely often [19].

For model checking, the alphabet is commonly defined from a set of atomic propositions AP as

$$\Sigma = 2^{AP}. \tag{71}$$

For a logging system, possible propositions include

$$AP = \{\mathsf{LogWritten}, \mathsf{Checkpoint}, \mathsf{Audited}, \mathsf{TamperDetected}\}.$$

D.2 LTL Model Checking

Let

$$K = (S, S_0, R, AP, L)$$

be a Kripke structure representing the implementation.

The execution paths of K can be represented by a Büchi automaton

$$B_{\mathsf{IMP}}$$

whose language contains the possible infinite traces of the modeled system [19].

Suppose that the desired temporal requirement is expressed in Linear Temporal Logic (LTL) as

$$\varphi = \mathbf{G}\,(\mathsf{LogWritten} \rightarrow \mathbf{F}\mathsf{Audited})\,. \tag{72}$$

Here,

$$\mathbf{G}$$

means "globally/always", while

$$\mathbf{F}$$

means "eventually". Equation (72) is a response/liveness requirement: every logged event must eventually be covered by an audit.

Rather than constructing an automaton for φ and then explicitly complementing it, a standard model-checking pipeline translates

$$\neg\varphi$$

directly into a Büchi automaton

$$B_{\neg\varphi},$$

which accepts precisely the modeled traces that violate the property.

D.3 Product Automata and Emptiness

The product automaton is then

$$B_{\mathsf{BOTH}} = B_{\mathsf{IMP}} \cap B_{\neg\varphi}. \tag{73}$$

Verification reduces to the emptiness problem

$$L(B_{\mathsf{BOTH}}) \stackrel{?}{=} \varnothing. \tag{74}$$

Therefore,

$$L(B_{\mathsf{BOTH}}) = \varnothing \tag{75}$$

implies that no execution admitted by the implementation model violates φ, and hence

$$K \models \varphi. \tag{76}$$

Conversely,

$$L(B_{\mathsf{BOTH}}) \neq \varnothing \tag{77}$$

implies the existence of a modeled execution trace violating the requirement. Such a trace may be returned as a counterexample.

Operationally, the model checker intersects the implementation automaton with the automaton for the negated property and tests the product for an accepting cycle.

The conclusion is relative to the modeled state space and specification. Emptiness does not establish that the real implementation is "absolutely secure" against every attack outside the model.

D.4 Program Checking Analogy

Program checking provides another useful, although more indirect, theoretical comparison.

Blum and Kannan introduced program checking as a weaker alternative to proving that a pro-

Troy University Resources

gram is correct on every possible input. A checker may execute a claimed program P as a black-box subroutine and attempt to determine whether its answer on a particular input x is incorrect [20].

In a standard formulation, a checker C for computational task $\mathcal{T}$ is a probabilistic polynomial-time machine with black-box access to P.

If P correctly computes $\mathcal{T}$ on all inputs, the checker satisfies a completeness condition such as

$$\Pr[C^P(x) = \mathsf{accept}] \geq \frac{2}{3}. \tag{78}$$

If

$$P(x) \neq \mathcal{T}(x),$$

the checker must reject with high probability, equivalently

$$\Pr[C^P(x) = \mathsf{accept}] < \frac{1}{3}. \tag{79}$$

The comparison with secure logging should be interpreted carefully. A hash-chain or history-tree verifier performs *instance-level verification*: given a concrete log and its associated cryptographic evidence, it decides whether the evidence is consistent with the claimed history.

This resembles the motivation of program checking because the verifier checks a concrete output rather than proving the correctness of the complete logging implementation.

However, the verifier developed in this project is *not formally claimed to be a Blum–Kannan program checker*. Establishing such a claim would require defining a computational task $\mathcal{T}$ and proving the corresponding checker completeness and soundness guarantees.

The program-checking connection is therefore used only as a theoretical analogy for instance-by-instance verification, while the actual security evaluation of this project remains based on cryptographic chain verification, adversarial tampering experiments, and explicit trust assumptions.

Appendix E. Detailed Experimental Data

The supporting materials include the machine-readable benchmark data, environment metadata, attack outcomes, and checkpoint measurements. The one-million-record JSONL file is omitted because it can be reproduced from the provided source code and experiment configuration.

- baseline verification results: `baseline_results.csv`;
- append measurements: `append_benchmark.csv`;
- full-chain verification measurements: `verify_benchmark.csv`;
- variable-payload verification measurements: `variable_payload_verify.csv`;
- storage measurements: `storage_overhead.csv`;
- checkpoint interval measurements: `checkpoint_overhead.csv`;
- checkpoint-window attack outcomes: `checkpoint_window_demo.csv`;
- controlled attack outcomes: `tampering_results.csv`;
- execution environment and fitted statistics: `environment.json` and `summary.json`.

Appendix F. Tampering Test Cases

The controlled evaluation used an untampered 100-record baseline together with 14 attack scenarios. The evaluated cases were middle-record modification with and without suffix recomputation, final-record modification with and without recomputation, interior deletion with and without recomputation, insertion with and without recomputation, adjacent-record reordering with and without recomputation, tail truncation by 10 records, rollback to record 60, whole-history replacement, and joint replacement of the history and checkpoint.

Local and checkpoint-aware outcomes for these cases are summarized in Table 10. The separate automated verification suite contains 14 unit tests, all of which passed.

Appendix G. Selected Source Code

The experiments use the implementation provided in `audit_log_demo.py`. The following excerpts are taken directly from the tested source code.

Listing 1: Deterministic record encoding and SHA-256 computation

```
def canonical_record_bytes(record: Dict[str, object])
    -> bytes:
    """Canonical UTF-8 JSON bytes authenticated by SHA
        -256."""
    obj = {field: record[field] for field in
        HASH_FIELDS}
    return _compact_json_bytes(obj)


def raw_record_bytes(record: Dict[str, object]) ->
    bytes:
    """Event data without hash metadata, used as
        baseline for storage comparison."""
    obj = {field: record[field] for field in RAW_FIELDS
        }
    return _compact_json_bytes(obj)


def protected_record_bytes(record: Dict[str, object])
    -> bytes:
    """Full stored record with current_hash included.""
        "
    obj = {field: record[field] for field in
        HASH_FIELDS}
    obj["current_hash"] = record["current_hash"]
```


Troy University Resources

```
    return _compact_json_bytes(obj)


def current_only_record_bytes(record: Dict[str, object]) -> bytes:
    """Record without previous_hash to evaluate storage saving."""
    obj = {field: record[field] for field in RAW_FIELDS}
    obj["current_hash"] = record["current_hash"]
    return _compact_json_bytes(obj)


def compute_record_hash(record: Dict[str, object]) -> str:
    return hashlib.sha256(canonical_record_bytes(record)).hexdigest()
```

Listing 2: Local and checkpoint-aware chain verification

```
def verify_records(records: Sequence[Dict[str, object]],
                   expected_log_id: str = DEFAULT_LOG_ID,
                   checkpoint: Optional[Checkpoint] = None) -> VerificationResult:
    previous = GENESIS_HASH

    if checkpoint is not None:
        if checkpoint.log_id != expected_log_id:
            return VerificationResult(False, 0, None, "checkpoint log_id mismatch", None)
        if checkpoint.encoding_version != ENCODING_VERSION:
            return VerificationResult(False, 0, None, "checkpoint encoding version mismatch", None)
        if len(records) < checkpoint.index:
            return VerificationResult(False, len(records), None, "log shorter than trusted checkpoint", None)

```

```
    for expected_index, record in enumerate(records, start=1):
        if record.get("log_id") != expected_log_id:
            return VerificationResult(False, expected_index - 1, expected_index,
                                      "wrong log identifier", previous)
        if record.get("encoding_version") != ENCODING_VERSION:
            return VerificationResult(False, expected_index - 1, expected_index,
                                      "wrong encoding version", previous)
        if record.get("sequence_number") != expected_index:
            return VerificationResult(False, expected_index - 1, expected_index,
                                      "wrong sequence number", previous)
        if record.get("previous_hash") != previous:
            return VerificationResult(False, expected_index - 1, expected_index,
                                      "broken previous link", previous)

        computed = compute_record_hash(record)
        if record.get("current_hash") != computed:
            return VerificationResult(False, expected_index - 1, expected_index,
                                      "record hash mismatch", previous)

        if checkpoint is not None and expected_index == checkpoint.index:
            if computed != checkpoint.chain_hash:
                return VerificationResult(False,
```


Troy University Resources

```
                    expected_index, expected_index,
36                                              "checkpoint
                                                   hash
                                                   mismatch",
                                                   computed)
37              if int(record["timestamp_ms"]) !=
                   checkpoint.timestamp_ms:
38                  return VerificationResult(False,
                       expected_index, expected_index,
39                                              "checkpoint
                                                   timestamp
                                                   mismatch",
                                                   computed)
40
41          previous = computed
42
43      return VerificationResult(True, len(records), None,
            "valid chain", previous if records else
           GENESIS_HASH)
```

The complete source additionally implements the controlled attack suite, checkpoint-window experiment, JSONL generation, runtime benchmarks, storage measurements, and CSV export. Automated tests are provided in `test_audit_log_demo.py`.