International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Tamper-Evident Chain of Custody for AI-Based Deepfake Forensics: A Hash-Chained Audit Log with Merkle Commitment and Digitally Sealed Reports

Avinash Utikar, Anup Pund, Soham Shinde, Avinash Bhondave
avinash.utikar@mituniversity.edu.in, anuppund.123@gmail.com, 009sohamshinde@gmail.com, avinashbhondave3@gmail.com
Department of Computer Engineering, MIT ADT University Pune, India

Abstract— A deepfake verdict is only as dependable as the record that surrounds it. The first version of our Deepfake Forensics Tool stored a SHA-256 digest of every submitted file and a JSON audit trail, but both sat in the same database as the analysis results, so anyone able to write to that database could rewrite the history without leaving a trace. This paper closes that gap with a lightweight tamper-evident integrity layer for AI-based media forensics. Each custody event is appended to a per-evidence hash chain; the chain is committed to a Merkle tree; and, when the report is produced, a seal that binds the file hash, chain head, Merkle root and report hash is signed with Ed25519 by a service outside the application server, while the seal digest is written to an independent append-only anchor store. A verifier needs only the public key, the anchor record and the evidence file. We built the layer as a Python prototype and replayed 1,000 synthetic analysis sessions through nine attack scenarios (value edits, timestamp edits, deletion, insertion, reordering, tail truncation, full chain recomputation, report modification and seal substitution) against four configurations. The complete design rejected all 9,000 tampered cases and accepted all 1,000 untouched sessions, whereas a plain JSON trail missed four attack classes entirely and a fifth in part, and a hash chain without an external anchor missed tail truncation and chain recomputation. On a single virtual CPU, appending an entry took about 5 µs, sealing a ten-event session about 0.12 ms, and verifying a 100,000-entry log about half a second.

Keywords— Deepfake Forensics, Chain of Custody, Tamper-Evident Logging, Hash Chain, Merkle Tree, Digital Signature, Ed25519, Evidence Integrity, Forensic Report Sealing, Audit Trail, Digital Evidence, Media Authentication

1. INTRODUCTION:

Synthetic media now reaches courtrooms, newsrooms and corporate investigations faster than the procedures for judging it can adapt [1]. Detection software answers one question, whether a file looks manipulated. A tribunal asks several more: is the file that was analysed the file that was received, did the pipeline run the way the report says it did, and is the document in front of the court the one the system actually produced? International guidance on digital evidence places weight on exactly these points of identification, acquisition and preservation [3], and Indian law has become explicit about them: Section 63(4) of the Bharatiya Sakshya Adhiniyam, 2023, together with its Schedule, requires a certificate that discloses the hash value of an electronic record offered as evidence [4].

Our earlier Deepfake Forensics Tool [2] took the first steps. It computed a SHA-256 digest when a file arrived, issued a UUID-based Evidence ID, recorded a JSON audit trail and produced a PDF report. That design has a structural weakness, however. The digest, the audit trail, the result row and the path of the report all live in one PostgreSQL instance and one file store that the application itself controls. The digest lets an examiner notice that the media file changed, but it says nothing about whether the audit trail or the report changed. A person with write access could lower a confidence score, regenerate the PDF, edit the log to match, and nothing stored in the system would disagree.

This paper addresses that weakness. We keep the detection pipeline untouched and add an integrity layer around it (Fig. 1). The contributions are:

- A threat model for AI-based forensic pipelines in which the adversary controls the database and the application server but not an independent signing service or an external anchor store.
- A compact design that combines a per-evidence hash chain, a Merkle commitment, an Ed25519-signed seal over the report and an anchor record, together with a six-check verification procedure that an outside party can run.
- A working prototype and a reproducible attack campaign: 1,000 synthetic sessions, nine attack scenarios and four configurations, so that every design element is justified by an attack that the simpler alternatives fail to detect.
- Measured costs for appending, sealing and verifying, showing that the layer adds a negligible share to the time already spent on analysis.

The remainder of the paper is organised as follows. Section 2 reviews related work, Section 3 states the problem and the threat model, Section 4 presents the architecture, Section 5 lists the algorithms, Section 6 analyses security, Sections 7 and 8 describe the experiments and their results, and Sections 9 to 11 cover future scope, acknowledgments and conclusions.

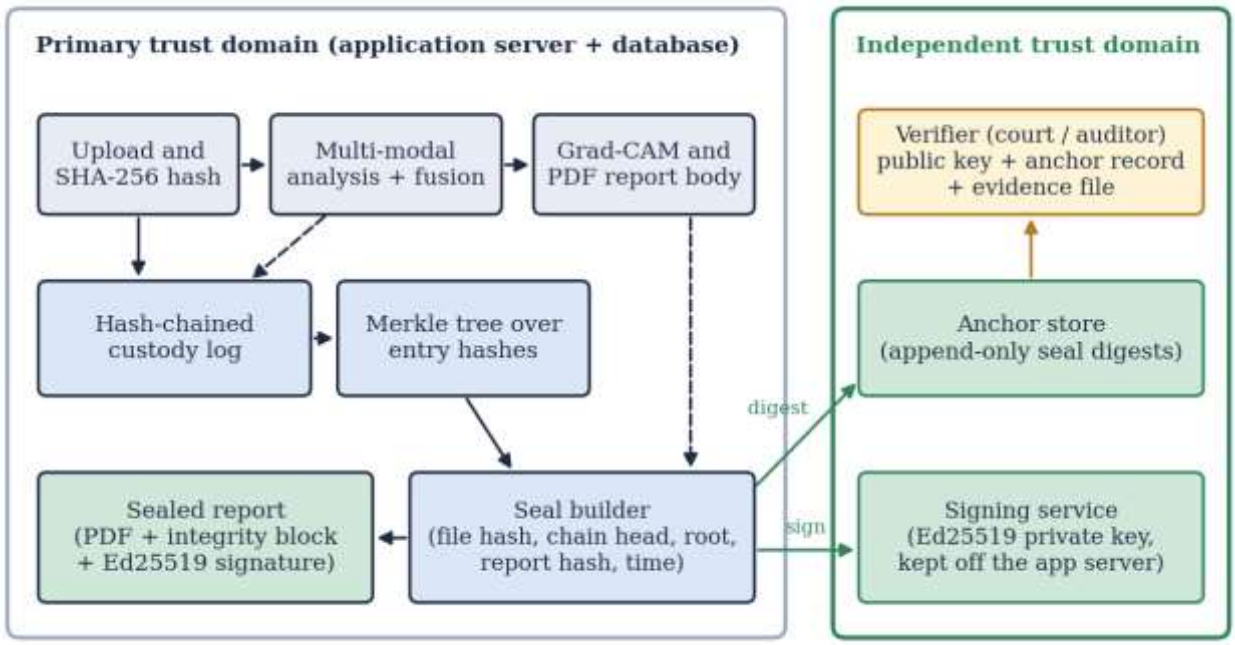


Fig. 1. Proposed tamper-evident integrity layer. Dashed arrows indicate data used for sealing; the right-hand domain cannot be modified from the application server.

2. LITERATURE SURVEY:

2.1 Deepfake detection and forensic reporting:

Research on detecting manipulated faces has been surveyed repeatedly [5], [8], and progress has been driven by large benchmarks such as FaceForensics++ [6] and the DFDC corpus [7]. These works concentrate on accuracy and on generalisation to unseen generators. Saliency methods such as Grad-CAM [9] make a prediction easier to inspect, which helps an examiner explain a result. What the literature treats far less is the custody of the result itself: how a score, its heatmap and the narrative around them stay trustworthy after they leave the model.

2.2 Tamper-evident logging:

Haber and Stornetta showed that documents can be linked through hashes so that later alteration breaks the link [10]. Schneier and Kelsey adapted the idea to audit logs kept on machines that might be compromised, using hash chaining and evolving keys [11]. Crosby and Wallach introduced history trees that support compact proofs about the presence of an entry and the consistency of two log versions [12]. Merkle trees [13] supply the logarithmic proofs that these systems rely on, and Certificate Transparency standardised the audit-path construction and its leaf and node prefixes [14]; we reuse that prefix convention in our commitment step.

2.3 Blockchains and trusted timestamps:

B-CoC [15] stores custody transfers of digital evidence on an Ethereum-based ledger, so that the record is held jointly rather than by one operator. Public ledgers of the Bitcoin type [16] and the time-stamp authorities defined in RFC 3161 [17] likewise act as outside witnesses for a digest. They are attractive anchors, yet each brings its own infrastructure, and neither prescribes what should be committed for an AI analysis pipeline. In our design they are interchangeable back-ends for the anchor store rather than the core of the contribution.

2.4 Content provenance:

The C2PA specification [18] defines signed manifests that travel with a media file and describe how it was captured and edited. This is complementary to our goal. C2PA speaks about the life of the media before it reaches an investigator, whereas the layer proposed here records what the forensic tool did after receiving it.

2.5 Gap addressed:

Among the sources surveyed, we did not find a design that treats the whole output of an AI deepfake-analysis session (events, verdict and report) as one sealed, independently verifiable object while remaining cheap enough to run on every submission. The present work targets that combination.

3. PROBLEM STATEMENT AND THREAT MODEL:

3.1 Weaknesses of the baseline:

Inspection of the first-generation tool [2] shows four weaknesses. **W1:** the JSON audit trail can be edited without any trace. **W2:** the stored file digest can be replaced together with the file. **W3:** the PDF report is not cryptographically tied to the log or to the analysis record. **W4:** every check requires trusting the operator of the database, because no verification can be performed outside it.

3.2 Adversary:

We assume an adversary who can read and write the database and the file storage, run arbitrary code on the application server and who knows every algorithm used. The adversary cannot reach the private signing key, which is held by a separate service, cannot modify the external anchor store, and cannot find SHA-256 collisions or forge Ed25519 signatures. Manipulation of a file before upload, poisoning of the detection models, theft of the signing key, denial of service and confidentiality of the evidence are outside the scope of this paper.

3.3 Security goals:

- **G1 Event integrity:** no logged event can be changed unnoticed.
- **G2 Order and completeness:** events cannot be inserted, removed or reordered, including at the end of a sealed log.
- **G3 Report binding:** a report can be matched to exactly one log, one file digest and one moment of sealing.
- **G4 Non-transferability:** a valid seal cannot be reused for a different evidence item.
- **G5 Independent verifiability:** a third party can check G1 to G4 without trusting the operator.

4. PROPOSED INTEGRITY LAYER:

4.1 Overview:

The layer sits beside the existing pipeline (Fig. 1). Every lifecycle event is appended to a hash-chained custody log. When the report body has been produced, a seal builder collects the file digest, the head of the chain, the Merkle root of the entry hashes and the digest of the report, and sends them to a signing service in an independent trust domain. The signed seal is printed in the report, and its digest is stored in an append-only anchor store. The verifier, such as a court expert, needs only the evidence file, the public key and the anchor record.

4.2 Custody entries and the hash chain:

Each entry carries a sequence number s, a UTC timestamp t, the Evidence ID, an event name e and a payload p. The event names follow the lifecycle already used by the tool (upload, hash_generated, analysis_started, per-module completion, fusion_complete, heatmap_generated and report_generated). The entry hash commits to its predecessor:

$$h_i = H(C(s_i, t_i, id, e_i, p_i) \| h_{i-1}) \quad (1)$$

where H is SHA-256 [19], C is a deterministic canonical JSON encoding (sorted keys, no whitespace, ASCII only) and h_{-1} is a fixed all-zero value. Because the payload of the hash_generated event contains the file digest, the digest itself becomes part of the chain.

4.3 Merkle commitment:

The entry hashes are the leaves of a binary Merkle tree. Following the prefix convention of [14] we separate leaves from interior nodes (Fig. 2):

$$leaf = H(0x00 \| h_i); \; node = H(0x01 \| L \| R) \quad (2)$$

An odd node at any level is promoted unchanged. The root summarises the entire log, and an audit path of $\lceil \log_2 n \rceil$ hashes proves that one entry belongs to it. This allows an investigator to disclose a single event, for example the fusion result, without handing over unrelated entries.

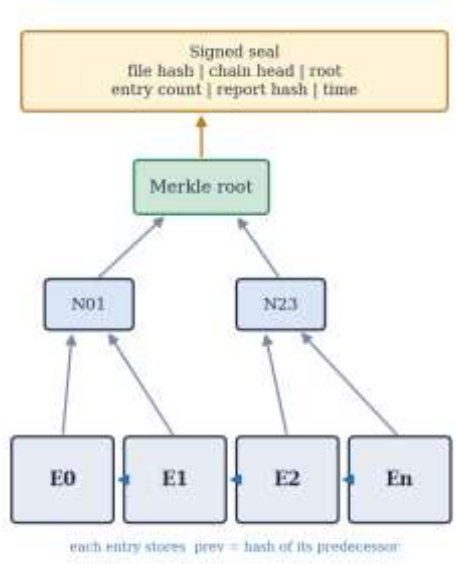


Fig. 2. Hash chain, Merkle tree and signed seal for one evidence item (E0 to En are custody entries).

4.4 Signed seal:

After the report body has been generated, the seal builder assembles the record S = (evidence ID, file SHA-256, entry count, chain head, Merkle root, report SHA-256, sealing time, algorithm). The signing service returns an Ed25519 signature [20], [21]:

$$\sigma = Sign(sk, C(S)) \quad (3)$$

The seal and signature are printed in the report as an Integrity Seal block. Later custody events, such as a report download or an evidence search, are still appended to the chain after the sealed prefix; the verifier checks that the sealed prefix is intact and treats the remaining entries as post-seal activity.

4.5 Anchor store:

The digest of the canonical seal is written to a store that the application server can append to but cannot modify, for example an object-lock bucket, a notary service or an RFC 3161 token [17]. The anchor defeats the attacker who replaces the entire evidence record, log and seal included, with a consistent forgery.

4.6 Verification procedure:

A verifier runs six checks. **Chain:** every entry hash and every previous-hash pointer recompute. **Signature:** σ verifies under the published key. **Prefix:** the entry at the sealed position hashes to the sealed head and the Merkle root of the sealed prefix equals the sealed root. **Report:** the digest of the report body equals the sealed value. **File:** the seal names the same evidence ID and file digest as the evidence in hand. **Anchor:** the seal digest equals the anchored record. The evidence is accepted only if all six pass.

4.7 Integration with the existing tool:

The layer needs few changes to the data model of [2]: a custody_entry table keyed by (evidence_id, seq) with columns for timestamp, event, payload, previous hash and entry hash; a seal column in the report table; two read endpoints, /verify/{evidence_id} and /proof/{evidence_id}/{seq}; and an Integrity Seal block in the ReportLab template. The Search Evidence page can show the result of the verification beside the

Evidence ID. The signing service is a separate process whose private key is never present on the application host.

5. ALGORITHM:

Step 1: Start

Step 2: Append a custody event

1. Build the entry (s, t, id, e, p) and set prev to the hash of the last entry, or to the zero hash for the first.
2. Compute h = H(C(entry) ‖ prev) and store the entry with h.

Step 3: Seal the evidence record

3. After report_generated, compute the Merkle root of all entry hashes.
4. Compute the SHA-256 digest of the report body.
5. Assemble S and obtain σ from the signing service.
6. Embed S and σ in the report and write the digest of S to the anchor store.

Step 4: Verify

7. Recompute every entry hash and check the previous-hash pointers.
8. Check σ against the public key.
9. Check that the sealed head and root match the first n entries of the log.
10. Recompute the report digest and compare it with the seal.
11. Compare the evidence ID and file digest in the seal with the evidence in hand.
12. Compare the digest of S with the anchored value. Accept only if all checks succeed.

Step 5: End

6. SECURITY ANALYSIS:

G1 and G2. Changing, inserting, removing or swapping an entry alters its hash, which no longer matches the pointer stored in its successor. Collision resistance of SHA-256 prevents the attacker from finding a different entry with the same hash. A chain alone, however, has two blind spots. Any prefix of a valid chain is itself valid, so the last entries can be cut off, and since the construction involves no secret, an attacker with write access can alter an entry and recompute every later hash. These two cases are the reason for the signed seal. The seal records the number of entries and the chain head, so a shortened or recomputed log no longer agrees with it, and the attacker cannot produce a fresh signature.

G3. The report digest is part of the signed record. Flipping a single bit of the PDF changes the digest and invalidates the match.

G4. The seal names the evidence ID and the file digest. A genuine seal copied from another case fails the file check and the anchor lookup.

G5. Verification uses only public data and the evidence file. Storing the Merkle root in the application database, in contrast, adds nothing against an attacker with database access, because the attacker simply stores the new root as well. The experiments in Section 8 confirm this.

Residual risks. Entries that were never sealed, for instance a session that crashed before report generation, are protected only by the chain. Forged events can still be appended after the last seal until a new seal or anchor covers them. Theft of the signing key would allow re-sealing, which is why the key is kept off the application host and why rotation and dual control are listed as future work.

7. EXPERIMENTAL SETUP:

7.1 Prototype and environment:

The layer was implemented as a standalone Python 3.12 module of roughly two hundred lines that mirrors the event names and report fields of the existing tool. SHA-256 comes from the standard hashlib library, Ed25519 from the cryptography package, and the sample report was produced with ReportLab. All measurements were taken on a single virtual CPU (Intel Xeon, 2.10 GHz) with 3 GB of memory, with a fixed random seed of 2026. The signing service and the anchor store were simulated in-process, so network latency is not included. The source code and experiment scripts can be obtained from the authors on request.

7.2 Workload:

We generated 1,000 synthetic evidence sessions: 334 image sessions with 8 events, 333 audio sessions with 9 events and 333 video sessions with 10 events. Each session carries random actors and scores and a file digest in its hash_generated event, and each was sealed with a minimal PDF report.

7.3 Configurations and attacks:

Four verifiers were compared (Table I). Nine attacks were then applied once to every session (Table II), giving 1,000 trials per attack, plus a control with no modification (A0). For C2 the attacker is root-aware: wherever the log is changed, the stored root is refreshed to match.

TABLE I. VERIFIER CONFIGURATIONS

ID	Description
C0	Plain JSON trail (baseline of [2]): checks the expected event sequence, sequence numbers and non-decreasing timestamps
C1	Hash chain only
C2	Hash chain plus Merkle root stored in the same database (unsigned)
C3	Proposed: chain, signed seal, report hash, file binding and external anchor

TABLE II. ATTACK SCENARIOS

ID	Attack applied to a sealed session
A1	Change one payload value in a random entry
A2	Shift the timestamp of a random entry by up to ±5 s
A3	Delete a random entry (not the last)
A4	Insert a forged entry at a random position
A5	Swap two adjacent entries
A6	Truncate the last 1 to 3 entries
A7	Edit an entry, then recompute all later hashes and the stored root
A8	Flip one random bit in the report PDF
A9	Replace the seal with a genuine seal from another session

7.4 Metrics:

Detection rate is the number of rejected sessions divided by the number of tampered sessions. The false rejection rate is measured on the untouched control. Latency figures are medians of repeated runs measured with a monotonic clock: seven repetitions for append and chain verification, three for the 100,000-entry case, and 300 for signature operations.

8. RESULTS AND DISCUSSION:

8.1 Detection:

Table III and Fig. 3 report the outcome. The complete design (C3) rejected all 9,000 tampered sessions and accepted all 1,000 untouched ones. The hash chain (C1) detected every local modification, deletion, insertion and reordering (A1 to A5), but it did not detect tail truncation (A6) or the recomputation attack (A7), and it does not cover the report or the seal at all. Storing the Merkle root beside the log (C2) behaved exactly like C1 against the root-aware attacker, which supports the argument in Section 6. The plain trail (C0) caught structural changes through its event-sequence check but never noticed an edited value (A1), caught an edited timestamp only when the edit broke the ordering (728 of 1,000), and was blind to A7 to A9.

A caution on interpretation is in order. Results for A1 to A5 under C1 to C3 follow directly from the properties of the hash function, so the campaign verifies the implementation rather than estimating an uncertain probability. The informative outcomes are the cases where simpler configurations fail, namely A6 to A9, because they show which components of the design carry weight.

TABLE III. SESSIONS REJECTED OUT OF 1,000 (A0: UNTOUCHED CONTROL)

Attack	C0	C1	C2	C3
A0 (control)	0	0	0	0
A1 edit value	0	1000	1000	1000
A2 edit time	728	1000	1000	1000
A3 delete	1000	1000	1000	1000
A4 insert	1000	1000	1000	1000
A5 reorder	1000	1000	1000	1000
A6 truncate	1000	0	0	1000
A7 rewrite	0	0	0	1000
A8 report bit	0	0	0	1000
A9 seal swap	0	0	0	1000

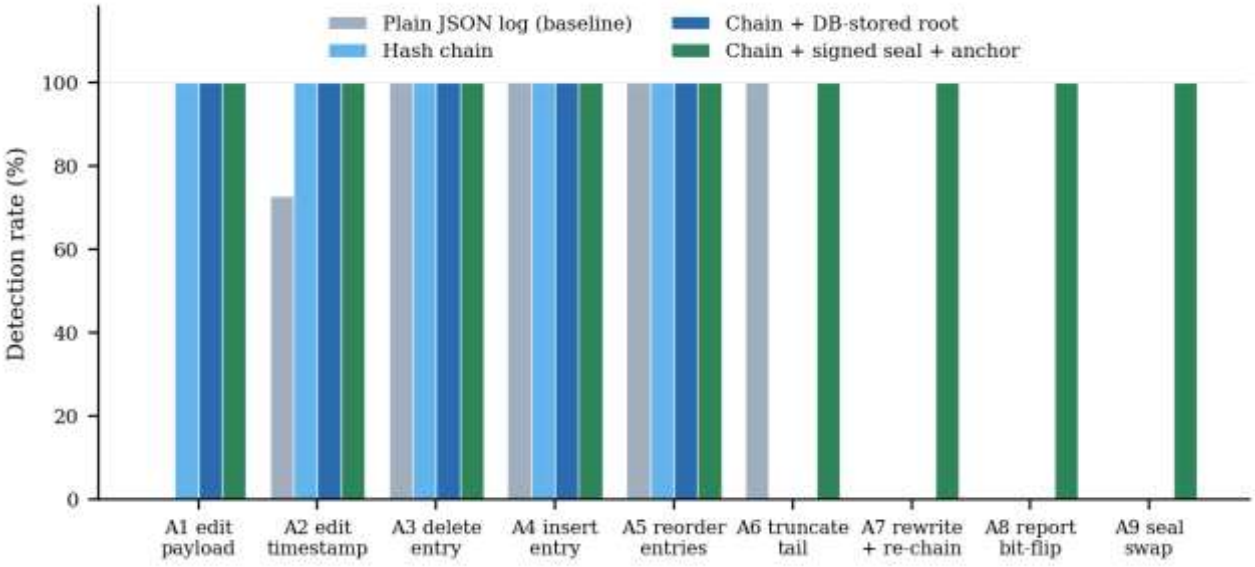


Fig. 3. Detection rate per attack scenario and verifier configuration (1,000 trials each).

8.2 Overhead:

Table IV summarises the costs. Chaining raises the append cost from 0.35 µs to 5.26 µs per entry and grows each stored entry from about 185 to 333 bytes. Sealing a ten-event session takes the ten appends (about 53 µs), one Merkle build (about 12 µs) and one signature (about 55 µs), roughly 0.12 ms in total. Against the 3.2 s average that we reported for analysing an image on GPU hardware [2], this is about 0.004 percent. Complete verification of a sealed ten-event session, including the signature, takes about 0.19 ms.

Chain verification grows linearly, at about 5 µs per entry, and reaches half a second for 100,000 entries (Fig. 4). Per-evidence logs are short, so this figure matters mainly for system-wide logs, where a verifier can start from the last checkpoint it has already trusted. An audit path for one entry in a log of 100,000 contains 17 hashes and is checked in about 13 µs.

TABLE IV. MEASURED COSTS (MEDIAN, SINGLE vCPU)

Operation	Result
Append entry, plain / chained	0.35 µs / 5.26 µs
Stored entry size, plain / chained	185 B / 333 B
Merkle build, n = 10 / 1,000 / 100,000	0.012 / 1.06 / 120 ms
Chain check, n = 10 / 1,000 / 100,000	0.042 / 4.5 / 500 ms
Audit path length, same n	4 / 10 / 17 hashes
Audit path check, same n	3.3 / 6.9 / 12.5 µs
Ed25519 sign / verify	54.7 µs / 113.1 µs
Full sealed verification (10 events)	0.19 ms
Seal size (canonical JSON)	608 B

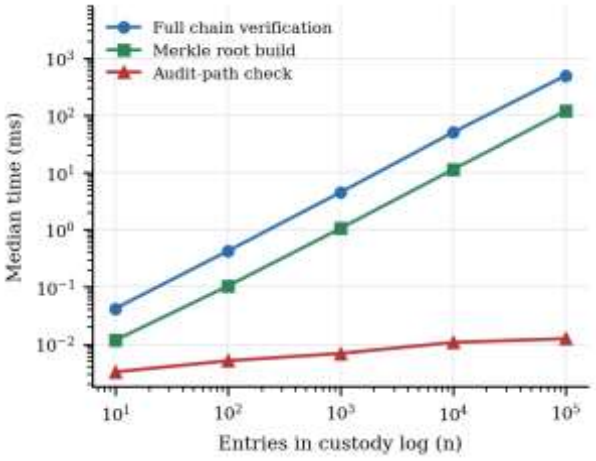


Fig. 4. Median time against log length n (log–log scale). Chain verification is linear; an audit-path check is nearly flat.

8.3 Prototype outputs:

To illustrate the result, a session of the existing tool was replayed through the prototype: the evidence ID, file digest, timestamps, verdict and heatmap are those of a validation run of the existing tool on 7 March 2026, and the custody events between them were generated by the prototype. Fig. 6 shows the resulting report. Beneath the original sections it now carries an Integrity Seal block with the entry count, chain head, Merkle root, sealing time, signature and the fingerprint of the verification key. Fig. 7 shows the verifier on five cases: the untouched record is accepted, and an edited score, a single flipped bit in the report, an edited entry with recomputed hashes, and a removed final entry are each rejected by a different combination of checks.

8.4 Existing tool for comparison:

Fig. 5 shows the corresponding screens of the first-generation tool [2]. The result page and the downloaded report display the Evidence ID, hash and verdict, but nothing on either allows a reader to establish whether those values were altered after the analysis. The seal adds that missing evidence without changing how the investigator uses the tool.

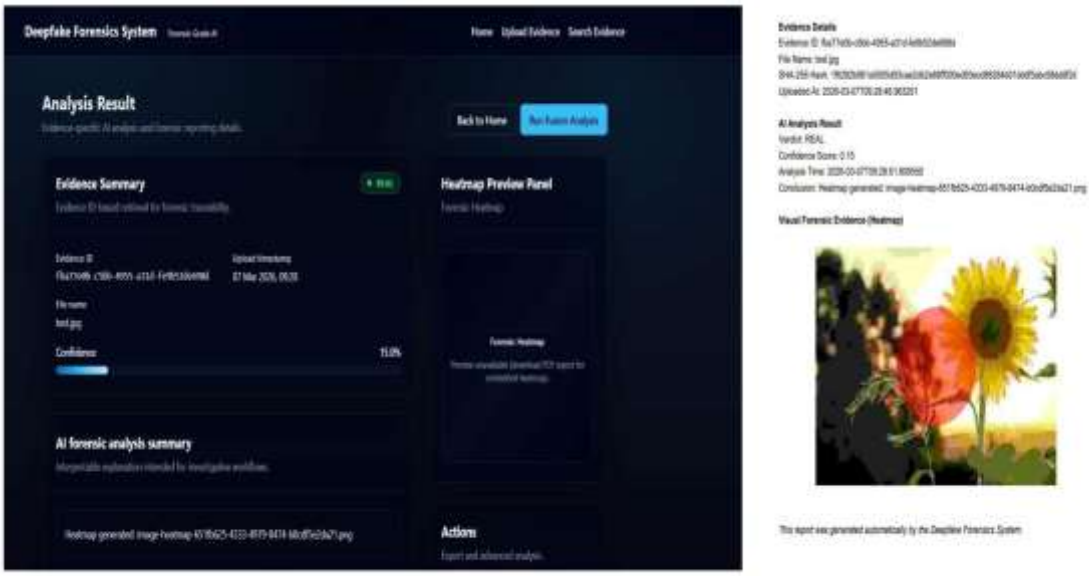


Fig. 5. Screens of the existing tool [2]: (a) analysis result page and (b) downloaded report, which carry the Evidence ID and hash but no verifiable seal.



Fig. 6. Report produced by the prototype for the replayed session, with the Integrity Seal block appended.



Fig. 7. Output of the six-check verifier on five cases (untouched, value edit, report bit-flip, recomputed chain, removed last entry).

8.5 Design comparison:

Table V places the configurations beside designs that rely on a public ledger. The first four rows were measured here; the ledger row reflects the design principle and was not evaluated. A ledger anchor can provide the same independence as our anchor store but requires a network of nodes, whereas the proposed design needs only a signing service and an append-only record.

TABLE V. DESIGN COMPARISON

Design	Edits	Tail cut	Independent	Infrastructure
Plain log (C0)	Partial	Template	No	None
Hash chain (C1)	Yes	No	No	None
Chain + DB root (C2)	Yes	No	No	None
Ledger-anchored (design only, cf. [15])	Yes	Yes	Yes	Ledger nodes
Proposed (C3)	Yes	Yes	Yes	Signer, anchor

8.6 Discussion and limitations:

Three observations stand out. First, an unsigned digest or Merkle root held inside the system under protection provides no defence against an attacker with write access; independence of the signer and the anchor is what matters. Second, the cost is small compared with the analysis itself, which removes any performance argument against sealing every session. Third, the same mechanism supports selective disclosure through audit paths.

The study has limits. The sessions are synthetic, and a real deployment will add network latency to signing and anchoring, and concurrency to appends, neither of which was measured. Timings come from Python on one virtual CPU. The layer was evaluated as a standalone prototype and has not yet been exposed through the production API. No user study with forensic examiners was performed, and whether a court admits an item of evidence is decided by the court, so the layer supports admissibility arguments but cannot guarantee them.

9. FUTURE SCOPE:

Batch anchoring. Publishing a daily Merkle root of all seals to an RFC 3161 authority or a permissioned ledger would extend protection to entries appended after the last seal.

Model provenance. Adding the digests of the model weights and of the software build to the seal would let a report state which detector version produced a verdict.

Key management. Threshold or dual-control signing, key rotation and a transparency record of published keys would reduce the consequences of a key compromise.

Operational evaluation. Measuring latency, concurrency and failure handling of the signing and anchoring services, including the behaviour when the signer is unavailable, is the next engineering step.

Media provenance. Verifying and chaining C2PA manifests [18] of submitted media would join pre-submission provenance to the custody record.

Court-facing verifier. A stand-alone offline verifier that prints a plain-language verification certificate aligned with the electronic-evidence certificate of [4] would ease use by non-technical parties.

10. ACKNOWLEDGMENTS:

The authors thank the faculty of the Department of Computer Engineering at MIT ADT University, Pune, and in particular Prof. Avinash Utikar, for guidance and encouragement throughout this work. We also acknowledge the open-source projects used in the prototype and the first-generation tool,

including Python, the cryptography package, ReportLab, Matplotlib, FastAPI, PostgreSQL and React.

11. CONCLUSION:

A detector's verdict becomes evidence only when the record around it can be trusted. We showed that the first version of our tool leaves that record editable by anyone who controls its database, and we proposed a layer that fixes this with four ingredients: a hash chain over custody events, a Merkle commitment, an Ed25519-signed seal that covers the report, and an external anchor. The attack campaign explains the role of each ingredient. The chain catches local edits, the signed seal catches truncation and recomputation, the report digest catches document changes, and the anchor catches substituted seals. All 9,000 tampered sessions were rejected and all 1,000 untouched sessions were accepted, at a cost of about 0.12 ms per session. The next steps are integration into the production tool, anchoring to an external time authority and evaluation with practising forensic examiners.

REFERENCES:

[1] R. Chesney and D. Citron, "Deep fakes: A looming challenge for privacy, democracy, and national security," California Law Review, vol. 107, pp. 1753–1819, 2019.

[2] A. Utikar, A. Pund, S. Shinde, and A. Bhondave, "Deepfake Forensics Tool: AI-driven multi-modal media authentication with forensic evidence reporting," Int. J. Sci. Res. Eng. Manag. (IJSREM), vol. 10, no. 3, Mar. 2026, doi: 10.55041/IJSREM57455.

[3] ISO/IEC 27037:2012, Information technology — Security techniques — Guidelines for identification, collection, acquisition and preservation of digital evidence, ISO, 2012.

[4] Government of India, The Bharatiya Sakshya Adhiniyam, 2023 (Act No. 47 of 2023), s. 63 and the Schedule.

[5] R. Tolosana, R. Vera-Rodriguez, J. Fierrez, A. Morales, and J. Ortega-Garcia, "Deepfakes and beyond: A survey of face manipulation and fake detection," Information Fusion, vol. 64, pp. 131–148, 2020.

[6] A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, "FaceForensics++: Learning to detect manipulated facial images," in Proc. IEEE/CVF Int. Conf. Computer Vision (ICCV), 2019, pp. 1–11.

[7] B. Dolhansky et al., "The DeepFake Detection Challenge (DFDC) dataset," arXiv:2006.07397, 2020.

[8] L. Verdoliva, "Media forensics and deepfakes: An overview," IEEE J. Sel. Topics Signal Process., vol. 14, no. 5, pp. 910–932, Aug. 2020.

[9] R. R. Selvaraju et al., "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in Proc. IEEE Int. Conf. Computer Vision (ICCV), 2017, pp. 618–626.

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

[11] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," ACM Trans. Inf. Syst. Secur., vol. 2, no. 2, pp. 159–176, 1999.

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

[13] R. C. Merkle, "A digital signature based on a conventional encryption function," in Advances in Cryptology — CRYPTO '87, LNCS vol. 293, Springer, 1988, pp. 369–378.

[14] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," IETF RFC 6962, Jun. 2013.

[15] S. Bonomi, M. Casini, and C. Ciccotelli, "B-CoC: A blockchain-based chain of custody for evidences management in digital forensics," in Proc. Int. Conf. Blockchain Economics, Security and Protocols (Tokenomics 2019), OASIcs, vol. 71, 2020, pp. 12:1–12:15.

[16] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[17] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, "Internet X.509 public key infrastructure time-stamp protocol (TSP)," IETF RFC 3161, Aug. 2001.

[18] Coalition for Content Provenance and Authenticity, "C2PA technical specification." [Online]. Available: https://c2pa.org

[19] National Institute of Standards and Technology, "Secure hash standard (SHS)," FIPS PUB 180-4, Aug. 2015.

[20] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," J. Cryptographic Eng., vol. 2, no. 2, pp. 77–89, 2012.

[21] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (EdDSA)," IETF RFC 8032, Jan. 2017.