A System for Verifiable Integrity in Public Document Archives

CS-3610 Course Project-cum-Assignment
Swapnanil Mukherjee

1 Introduction and Problem Definition

Public service applications, such as government portals for land registries, court filings, legislative archives, or public records, operate on a high-trust model. A citizen accesses the portal and downloads a document, trusting that the server is providing the authentic, correct version. This "trust-the-server" model is simple but brittle. The integrity of the entire public record hinges on the absolute, perpetual security of the server and the unwavering honesty of its administrators.

This trust is frequently violated in practice. Malicious actors, both internal and external, can and do tamper with official records. In India, there are numerous legal cases concerning the use of forged documents, such as fraudulent mark sheets, to obtain government employment, demonstrating a clear motive to alter official records [1, 2]. These forgeries often involve sophisticated collusion, sometimes with the "involvement of insiders" who facilitate the fraud [1]. Similarly, passport applications and other official documents have been the subject of forgery, leading to criminal cases [3].

The threat is not limited to modification; it also includes unauthorized destruction. The U.S. National Archives, for example, maintains logs of "Unauthorized Disposition of Federal Records." These logs detail numerous cases where official records were "unlawfully destroyed," "prematurely shredded," or "missing" from official systems, sometimes due to insider actions like a departed political appointee [4].

When a server is compromised or an administrator acts maliciously, a document can be silently altered, or worse, deleted entirely. The public has no mechanism to detect this breach of trust. A user who downloads a tampered document has no way to verify its authenticity. A researcher looking for a public record that has been silently deleted will simply find no result, assuming it never existed.

This project addresses this gap. The central question is: **How can an end-user independently verify that a document obtained from a public portal is authentic, untampered, and that its existence is being accurately reported?**

The objective is to design a system where the server's claims of integrity are not taken on faith but are publicly verifiable through cryptographic proof. This system must provide end-to-end security that ensures document integrity and accountability, primarily against the threat of a malicious insider.

2 Threat Model

To build a robust solution, it is essential to define the threats the system faces formally. The threat model concerns an adversary who wishes to modify, forge, or delete public records without detection.

2.1 Threat Actors and Adversaries

1. **Malicious Insider:** This is the most dangerous adversary.
    - **Identity:** A system administrator, database administrator, or a government employee with privileged, trusted access to the Public Records Server (PRS).
    - **Motive:** Varies widely, from financial gain (e.g., accepting a bribe to alter a land deed), ideological reasons, personal grievances, or coercion.
    - **Capability:** Full, legitimate root-level or administrative access. This actor has legitimate authorization to read, write, modify, and delete any file on the server. They can also modify or erase local audit logs (e.g., `syslog`, `auth.log`) to cover their tracks.
2. **External Attacker:**
    - **Identity:** A remote hacker, criminal organization, or state-sponsored actor.
    - **Motive:** Financial fraud, personal gain or grievance, political destabilization (e.g., by tampering with proposed legislation), or vandalism.
    - **Capability:** Gained privileged access by exploiting a software vulnerability (e.g., SQL injection, identity spoofing, phishing etc.). For this model, we assume this attacker has successfully escalated privileges and obtained capabilities equivalent to the Malicious Insider.

2.2 Potential Threats (Attack Vectors)

The adversaries will attempt the following attacks:

- **Unauthorized Document Modification:** The adversary modifies the content of an existing document. For example, changing a name on a court filing, altering a dollar amount in a public contract, or modifying a clause in a proposed law.
- **Silent Document Deletion:** The adversary deletes a document from the public-facing server. This is the most insidious attack, as the document (e.g., an inconvenient court record, evidence of a promise) simply ceases to exist.
- **Log Tampering:** The adversary modifies the server's internal logs to erase evidence of the actions taken in the two attacks described above. This attack renders traditional server-side logging insufficient for auditing.
- **Unauthorized Document Upload:** The adversary uploads a new, fraudulent document and attempts to make it appear legitimate, for instance, by back-dating its timestamp. This is an extension of the first attack.

2.3 Trust Assumptions

The proposed system must operate with a "zero trust" (or minimal trust) policy toward the server itself. This model explicitly defines what cannot be trusted and what must be trusted.

- **The Server is Untrusted:** We assume the server (PRS) is potentially compromised or malicious. Its claims of "file X is authentic" or "file Y does not exist" cannot be taken to be truth.

- **Server Administrators are Untrusted:** We assume any administrator with access to the server is a potential adversary.
- **The Client is Trusted:** We trust the user's local machine and the verification software (the "client") to perform calculations correctly (e.g., hash a file).
- **Dedicated Cryptographic Hardware is Trusted:** When using dedicated security infrastructure such as the Hardware Security Module (HSM) or Trusted Platform Module (TPM) for cryptographic operations and key storage, we can assume its integrity and the inability of an attacker to exploit such infrastructure.
- **Cryptography is Trusted:** We trust the correctness of standard, cryptographic algorithms, specifically SHA-256 (a cryptographic hash function) and public-key signature algorithms (e.g., ECDSA).
- **The Witnesses are Partially Trusted:** As will be detailed in the solution, the system will rely on a set of independent, public "witness" logs. The trust assumption is that **at least one** of these witnesses is honest and will not collude with the PRS or any adversary. The system's security fails only if *all* witnesses collude with the server simultaneously.

2.4 Verifiability

The goal of the solution is to provide **public verifiability** by achieving the following properties:

1. **Integrity (File Modification):** Any user must be able to cryptographically prove that a downloaded document is identical to the version that was originally uploaded to the server.
2. **Tamper-Evidence (for Deletion):** It must be computationally infeasible for an adversary to delete a document without leaving behind cryptographic evidence that the deletion occurred. The system log must be append-only.
3. **Accountability & Non-Repudiation (for Upload):** A user who submits a document (e.g., a lawyer filing a court document) must receive an immediate, undeniable cryptographic receipt (a Signed Upload Receipt) from the server. This receipt allows the uploader to later prove that their document was submitted, even if the server later denies it.
4. **Public Auditability:** Any third party, without needing special credentials, must be able to audit the entire history of actions (uploads, deletions) and verify that the server's logs are consistent and untampered.

3 Proposed Solution Architecture

The proposed solution moves the system from an implicit trust model to an explicit verification model. The core idea is that the server (PRS) must not be the sole arbiter of its own integrity. Instead, it is forced to create a continuous, publicly-auditable, and cryptographically-secured record of all its actions.

This is achieved by logging all state-changing events (uploads, modifications, deletions) and periodically committing a cryptographic summary of these events to a set of independent, public "witness" servers. These servers can be independent audit organizations or neutral but trusted third parties. Any user can then perform a client-side verification to check a document's integrity against this immutable public record.

3.1 Core Components

- **Public Records Server (PRS):** The central application (simulated as a Python Flask server) that handles file uploads, downloads, and user requests. It is responsible for generating event logs, creating batches, and publishing them. This provides functionality to authorised users to `upload`, `modify`, or `delete` files and documents through a standard operating procedure which adheres to the security architecture defined below.
- **Event Log:** An in-memory list (a Python `list`) maintained by the PRS. It temporarily stores event objects (as dictionaries-like structures) as they occur, before they are processed into a batch.
- **Object Store:** A directory on the server's file system (e.g., `/uploads/`) where the raw files are stored.
- **Server Keys:** A pair of cryptographic keys (e.g., ECDSA) for signing. The private key (`private_key.pem`) is stored securely on the PRS by means of a dedicated security module such as the TPM on Intel/AMD processors. The public key (`public_key.pem`) is distributed to clients for verification.
- **Federated Witness Logs (FWLs):** A set of independent, append-only public logs. For this project, they are simulated as three or more `git` repositories. Git is used because its data structure is an inherent Merkle tree, making it append-only by default; rewriting public history is both detectable and difficult.
- **Client Verifier:** A user-side script that downloads a file and all associated proofs from the PRS, then independently runs a series of cryptographic checks to confirm its authenticity.

3.2 Task 1: The Upload and Receipt (SUR) Generation

This process provides immediate accountability for the *uploader* and addresses Security Goal 3 (Accountability & Non-Repudiation).

When a user (e.g., attorney or court clerk) uploads a document, the server must provide an immediate, non-repudiable receipt.

1. A user `POST`s a file to the PRS.
2. The PRS receives the file and calculates its hash: `file_hash = SHA256(file)`.
3. The file is saved to the Object Store, indexed by its `file_hash`.
4. The PRS generates an Event object (implemented as a JSON-like structure):
   `{"action": "upload", "file_hash": "...", "timestamp": "...", "user_id": "..."}`
5. This event is appended to the PRS's in-memory **Event Log**.
6. The PRS then creates a **Signed Upload Receipt (SUR)**.
   - **Definition:** The SUR is a cryptographically signed promise. It is a data structure containing the `event` data, signed using the server's private key (`K_PRS_Sign`).
   - **Generation:** `signature = Sign(event, K_PRS_Sign)`
7. The PRS returns the `(event, signature)` pair to the uploader.

The uploader now possesses an undeniable, signed receipt. They can hold the server accountable, as they can publicly prove the server received this event at this time. This addresses the threat of silent document deletion by ensuring every upload has a corresponding, signed event which the uploader can use to claim their upload.

3.3 Task 2: Batching and Public Commits

This process, run periodically (e.g., every 10 minutes), creates the public, tamper-evident record. It addresses Security Goals of tamper-proofing and public auditability.

1. A scheduled task on the PRS takes the entire in-memory **Event Log** (containing all events from the last 10 minutes).
2. It constructs a **Merkle Tree** from this list of events.
    - **Definition:** A Merkle Tree (or hash tree) is a binary tree where each leaf node is a hash of an individual event, and each non-leaf node is a hash of its two child nodes.
    - **Purpose:** This structure allows a single, compact hash—the **Merkle Root**—to represent the entire, ordered set of events. It also enables efficient "inclusion proofs" (Task 3).
3. The PRS calculates the `Merkle_Root` of the event tree by hashing each event (Step 2, Task 1) and computing the tree bottom-up.
4. The PRS signs this root: `Batch_Signature = Sign(Merkle_Root, K_PRS_Sign)`.
5. The PRS commits a new "batch file" to **all** FWLs. This file contains:
    - The `Merkle_Root`.
    - The `Batch_Signature`.
    - The full list of `events` in the batch (for public auditability).
6. The PRS transfers this commit to the remote FWL repositories using a secure, encrypted channel (e.g. TLS).
7. Upon successful push, the PRS *clears* its in-memory Event Log.

This process creates the immutable public record. Since the FWLs are independent and append-only, the PRS cannot modify or erase this history, and it is publicly available.

3.4 Task 3: Client-Side Verification

This process is performed by any user wishing to download a document and verify its integrity and authenticity.

1. The client (user) requests a document from the PRS using its `file_hash`.
2. The PRS retrieves the `file` from the Object Store.
3. The PRS searches the (now public) batch history to find the `event` corresponding to the `file_hash`.

4. The PRS generates a **Merkle Proof** for **that** event.
    - **Definition:** A Merkle Proof (or inclusion proof) is the minimal list of sibling hashes from the tree that allows the client to recalculate the Merkle Root, using only their event hash.
5. The PRS returns the following package to the client:
    - The `file` itself.
    - The `event` data.
    - The `Merkle_Proof`.
    - The signed batch header: `(Merkle_Root, Batch_Signature)`.
6. The client-side verifier script automatically performs a 4-step check:
    (a) **Check 1 (File Integrity):** Verify that `SHA256(file) == event.file_hash`. This proves the file content matches the event log.
    (b) **Check 2 (Event Inclusion):** Use the `event` and `Merkle_Proof` to recalculate a root. Verify that `Recalculated_Root == Merkle_Root`. This proves the event was part of the claimed batch.
    (c) **Check 3 (Batch Authenticity):** Verify the batch signature: `Verify_Signature(Merkle_Root, Batch_Signature, K_PRS_Public)`. This proves the PRS vouched for this entire batch.
    (d) **Check 4 (Public Witness):** Contact one or more of the public FWLs (e.g., `git pull` from the repository) and verify that the `Merkle_Root` exists in their public commit history. This proves the batch wasn't fabricated for the client, but is part of the permanent public record.

If all four checks pass, the user has high cryptographic assurance that the file is authentic and its existence is publicly witnessed.

3.5 Threat Mitigation

This architecture directly mitigates the threats defined in the threat model:

- **T1 (Unauthorized Modification):** Mitigated by **Check 1** of the client verifier. If an adversary modifies a file, its hash computed by the client will no longer match the `file_hash` in the signed, public event log. The verification will fail instantly.
- **T2 (Silent Deletion):** Mitigated by the event log structure. To delete a file, the adversary must do one of two things:
    1. **Honest Deletion:** Create a {`"action": "delete", ...`} event. This is logged and publicly witnessed, providing a transparent audit trail of the deletion.
    2. **Silent Deletion:** Try to delete the file from without a trail (dishonestly). The file's original `"upload"` event still exists in the public FWLs. A user (or auditor) can now prove that the server is failing to provide a file that it publicly promised to store, providing cryptographic evidence of malfeasance.

- **T3 (Log Tampering):** Mitigated by the **FWLs**. The adversary can tamper with any local server logs, but these are irrelevant. The *source of truth* is the set of public, external, append-only FWL which are multiple in number.
- **T4 (Forged Document Upload):** Mitigated by the **SUR**. An adversary cannot forge a back-dated document, as they would need to create a signed SUR for it. Furthermore, any new upload will create a new `upload` event, which will be logged in the *current* batch, not a past one.
- **T5 (History-Revision):** Mitigated by the combined **Merkle Tree** and **FWL** structure. An adversary cannot change a past event (e.g., in Batch 5) without changing its hash, which would change the Merkle Root of Batch 5. This would require re-signing and re-committing Batch 5, which would also change the history of all subsequent batches. This breaks the append-only nature of the FWLs and would be immediately obvious to any public auditor.

This architecture, while effective for its stated goals, relies on a performance-security trade-off. The 10-minute batching window, during which events are held in memory before being committed, introduces a vulnerability. This limitation, along with others, has been discussed in the later sections. Following that, a theoretical solution to mitigate them will is the final section.

4 Limitations and Vulnerabilities

The proposed architecture is designed and optimized for simplicity and efficiency, makes specific trade-offs that leave it vulnerable to a highly sohisticated and technically capable adversary.

4.1 The Batching Window Vulnerability

This is the most significant security flaw in the proposed simplified design.

- The system uses a "batching" process, where events are collected in an in-memory `container` as a batch, for a given period (e.g., 10 minutes) before being cryptographically secured and published. This batching window, from the moment an event occurs to the moment it is publicly committed, is the vulnerability.
- **Core Vulnerability:** The `events` list is a mutable data structure (e.g. a Python `list`) that resides in the server's main memory (RAM). This list is controlled by the same operating system (and thus, the same root user) that we have defined in our threat model as the **Malicious Insider**.
- **Attack Scenario:** An adversary with root access to the PRS can exploit this window to perform several integrity-violating attacks.
  1. The adversary gains root-level access to the PRS.
  2. They monitor the server's memory for the `events` list.
  3. A legitimate user uploads a sensitive document, `evidence.pdf`. The PRS generates the event {`"action": "upload", "file_hash": "abc...", ...`} and appends it to the in-memory `events` list. The uploader receives a Signed Upload Receipt (SUR) for this event.

4. **Before** the batch timer expires, the adversary strikes. They can choose from several attacks:
   - **Attack A (Event Deletion):** The adversary simply removes the `evidence.pdf` event from the in-memory `list`. When the batching process runs, it will build a Merkle Tree of all *other* events, but not this one. The `evidence.pdf` event is never publicly witnessed. The server can later claim the user's SUR is fraudulent, as it cannot be found in any public batch. If the uploader is also the adversary, they can simply purge the SUR, and there would remain no record of the upload.
   - **Attack B (Event Tampering):** The adversary modifies the event in-memory. For example, they change `"action": "upload"` to `"action": "delete"`, or they modify the `"user_id"` to frame another user. This tampered event is then included in the batch, creating a false public record.

This vulnerability exists because the cryptographic signature (in the SUR) only proves the event was *created*, not that it was *added to the list* that will be used to build the batch.

4.2 Private Key Security

The security of the entire protocol is greatly dependent on the security of the private key of the PRS.

- The architecture design stores the server's private signing key, `K_PRS_Sign`, as a file on the server's file system (e.g., `private_key.pem`).
- **Vulnerability:** The Malicious Insider, by definition, has root access and can therefore read this file.
- **Attack Scenario (Catastrophic Failure):**
  1. The adversary steals the `private_key.pem` file.
  2. The entire trust model of the system collapses. The adversary can now:
     - Forge SURs for any file, including back-dated, fraudulent ones.
     - Create entirely new, malicious Merkle batches.
     - Sign these malicious batches with the stolen private key.
     - Push these fraudulent batches to the Federated Witness Logs (FWLs).
  3. A client verifier has no way to detect this. The signatures will be cryptographically valid. The adversary can rewrite history with impunity, and the system itself will provide "proof" that the new, fraudulent history is correct.

However, this threat is **minimal** as long as the security processor of the system's CPU (e.g. TPM 2.0) is not compromised, which in reality is practically infeasible.

4.3 Collusion of Witnesses

- The security of the FWLs relies on a "1-of-N" trust model, meaning we trust that at least one of the N witnesses is honest and will not collude with the adversary.
- **Vulnerability:** If an adversary (e.g., a powerful state actor) can compromise the PRS *and* all N of the witness servers, the system fails.

However, in reality these FWLs are large independent security organizations. Compromising them all at the same time would be incredibly challenging and practically infeasible. Therefore, the risk from this threat is also **minimal**.

4.4 Denial of Service (DoS) via Log Bloat

- The system is open to any authorized user uploading files.
- **Vulnerability:** An attacker can write a simple script to upload millions of 1-byte files. Each upload, no matter how small, generates a unique event.
- **Attack Scenario:** If the attacker can somehow manage to gain credentials of a (regular, non-root) authorized user of the system, they can flood and overload the system, creating massive event logs. This bloats the Merkle trees and potentially choke their transmission to the FWLs. The batches become huge in size, making it computationally infeasible for the system to function as expected, thus bringing it down. Then the malicious user can tamper with the system's contents at will.

The risk posed by this threat is at **high** because it is quite possible for a sophisticated adversary to steal user credentials and gain access to a PRS, and then launch a Denial-of-Service attack. But this can be somewhat mitigated by using standard mechanisms against DoS attacks such as rate-limiting, activity pattern monitoring and alerts, etc.

4.5 On the Privacy of the System

The system is designed for *transparency*, not *privacy*. The event logs, which are pushed to the public FWLs contain the action-event logs and are ultimately made public. This metadata includes information like `user_id`, `file_hash`, and `timestamp`. This reveals who uploaded or deleted what file, and when. While this is a "feature" for a fully public archive, it makes this architecture unsuitable for any system that requires user anonymity or confidentiality of access patterns.

This architecture, by design, is **not intended** for applications requiring high privacy and confidentiality and should not be used as such.

5 Mitigating the Batching Vulnerability

The critical "Batching Window" vulnerability (L1) and "Key Theft" vulnerability (L2) are both symptoms of the same root cause: the server's software (and thus its root user) has access to its own most sensitive assets (the in-memory event list and the private key).

This section proposes a theoretical recommendation for a production-grade system to mitigate these flaws. **This architecture is not implemented directly** due to its engineering complexity, but has been simulated with simplistic assumptions and programming primitives.

A production-grade solution mitigates this by moving these assets from software to hardware. This is achieved using a **secure co-processor**, such as a Hardware Security Module (HSM) or a Trusted Platform Module (TPM). A TPM is more accessible and cost-efficient than an HSM and is available on most modern commercial processors.

5.1 Architecture Modification

A secure co-processor (which will be referred to as an HSM for simplicity) is a dedicated hardware device that is physically and logically separate from the server's main CPU. It is designed to perform cryptographic operations and protect secret data, even from the server's own root user.

The architecture is modified in two ways:

1. The `K_PRS_Sign` (private key) is moved *inside* the HSM. It is generated inside the HSM and marked as "non-exportable." The server can only send data *to* the HSM and request a signature; it can never read the key.
2. The HSM is required to store one additional piece of data: `Latest_Event_Hash`. This is a single SHA-256 hash that represents the "head" of the event chain, securing the *sequence* of events.

5.2 Revised Task 1: Instantaneous Event Commit

The batching vulnerability is eliminated by processing each event instantaneously, securing both its *content* and its *place in history*.

1. A user uploads a file. The PRS generates the `Event_N` (the `event` object, e.g., {`"action": "upload", ...`}).
2. The PRS makes a single, atomic API call to the HSM:
   `(Chain_Hash, Signature) = HSM.ChainAndSign(Event_N)`
3. **Inside the HSM (Atomic Operation):** This operation must be atomic, meaning it cannot be interrupted.
   (a) **Secure the Sequence:** The HSM retrieves its *internally-stored* `Latest_Event_Hash` (let's call it `Hash_N-1`).
   (b) It calculates the new chain head:
   `Hash_N = SHA256( Event_N + Hash_N-1 )`
   (c) It atomically overwrites its internal `Latest_Event_Hash` with `Hash_N`.
   (d) **Secure the Content:** It retrieves its *internally-stored* private key, `K_PRS_Sign`.
   (e) It signs the *event data itself*: `Signature_N = Sign(Event_N, K_PRS_Sign)`.
   (f) It returns the `(Hash_N, Signature_N)` pair to the PRS.
4. **PRS Action:** The PRS returns the `(Event_N, Signature_N)` to the uploader as their **Signed Upload Receipt (SUR)**. The uploader can now independently verify this signature

5.3 Revised Task 2: Batch Commit

This is now a publishing operation for scalability and public verification, not a security-critical one.

1. Every 10 minutes, the PRS finalizes the `current_batch.log` file.
2. It builds a **Merkle Tree**. The leaves of this tree are a hash of each event: `leaf = SHA256(Event_N)`.
3. It calculates the `Merkle_Root` for this batch.

4. It gets the *last* chain hash from its log, `Final_Chain_Hash`, which was generated by the HSM.
5. It retrieves the hash of the *previous* batch's header, `Previous_Batch_Header_Hash`.
6. It creates a `Batch_Header` object:

```
{
    "merkle_root": "...",
    "final_chain_hash": "...",
    "previous_batch_header_hash": "...",
    "timestamp": "..."
}
```

7. The PRS requests the HSM to sign this `Batch_Header` (or just the `merkle_root` and `final_chain_hash` for efficiency).
8. The PRS publishes the `(Signed_Batch_Header, all_events_in_batch)` to the FWLs.

5.4 Security Analysis of the Revised Model

This combined hardware-software model successfully mitigates the critical vulnerabilities.

- **Batching Window Vulnerability Mitigated:** The vulnerability is gone. An event is atomically bound to the HSM's hash chain in nanoseconds. The `(Event, Hash)` pair is also immediately persisted to disk. An adversary cannot tamper with this on-disk log without breaking the HSM-secured `Hash_N` sequence, which would be detected by any auditor.
- **Key Security Vulnerability Mitigated:** The private key never exists in software. The adversary *cannot* steal the key to forge past events.

5.5 Verifiability in the Revised Model

This architecture provides robust, multi-layered verifiability. Here is a clear, descriptive breakdown of the user-side verification process.

5.5.1 Detection of File Tampering (User Verification)

This is the check performed by a user on every download. It is now a 4-step process.

1. **User Action:** The user requests to download `document.pdf`.
2. **Server Response:** The server provides a "verification package":
    - The file content: `document.pdf`
    - The event: `Event_N` (the `event` object)
    - The event signature: `Signature_N` (the SUR)
    - The inclusion proof: `Merkle_Proof`
    - The batch header: `Signed_Batch_Header`
3. **Attack:** A malicious admin has modified the file to `document_tampered.pdf`.

4. **Client-Side Verification:** The user's tool performs four checks in sequence. Each check proves a different fact.

   - **Check 1: Content Integrity**
     - **What it does:** `SHA256(downloaded_file) == Event_N.file_hash`
     - **What it prove:** The downloaded file is the same file (bit-for-bit) that the server originally logged in this event. It has not been tampered with.
     - The attacker modifies the original document. The client downloads this and calculates its hash (e.g., "xyz..."). The `Event_N` (which is signed and cannot be changed) contains the *original* hash ("abc..."). The check fails: "xyz..." != "abc...".
   - **Check 2: Event Authenticity**
     - **What it does:** `Verify(Event_N, Signature_N, K_PRS_Public)`
     - **What it proves:** The event data itself (`Event_N`) is authentic and has not been faked."
     - This check prevents an attacker from just creating a new, fake event to match their tampered file. They cannot forge the `Signature_N` from the HSM. This is the check that makes the SUR a non-repudiable receipt for the original uploader.
   - **Check 3: Batch Inclusion**
     - `MerkleVerify(SHA256(Event_N), Merkle_Proof) == Signed_Batch_Header.merkle_root`
     - **What it proves:** This event (`Event_N`) was officially part of a specific batch of events (publicly available), summarized by this `merkle_root`. This prevents the server from lying by omission.
   - **Check 4: Public Witness**
     - **What it does:** The tool contacts the public FWLs and checks that the `Signed_Batch_Header` (which contains the `merkle_root` from Check 3) is publicly listed.
     - **What it proves:** This entire batch of events, which the requested (downloaded) file is part of, was publicly published and recorded by independent third parties. It is now part of the permanent, immutable public record.

5.5.2 Detection of Silent File Deletion

This check is performed by an auditor or a user who believes a file is missing that is supposed to be served by the PRS but is not.

1. **Attack:** A malicious admin deletes `document.pdf` from the Object Store but does not create a "delete" event.
2. **Auditor Action:** The auditor downloads the *entire* event log from the FWLs.
3. **Auditor Verification:**
   (a) The auditor replays the entire event chain, verifying all HSM-generated chain hashes (`Hash_N`) and all batch header signatures. This proves the log is an unbroken, authentic history.
   (b) The auditor finds the {`"action": "upload", "file_hash": "abc...", ...`} event, proving the file *was* stored.

(c) The auditor scans the *remainder* of the chain for a corresponding {`"action": "delete", "file_hash": "abc...", ...`} event.

4. **Detection:** The auditor finds no "delete" event. They now have cryptographic proof that the server's state is inconsistent with its own immutable log. The "file not found" error is proven to be a silent deletion.

5.5.3 Detection of File Replacement

This attack is a combination of modification and deletion.

1. **Attack:** A malicious admin wants to replace `document.pdf` (hash "abc...") with `malicious.pdf` (hash "xyz...").
2. **How the attacker must proceed:**
   - **Option A (Overwrite):** The attacker just overwrites the original file's content. This is a **File Modification attack and is detected by the user's client-side Check 1, as described above.
   - **Option B (Delete and Upload):** The attacker must: 1. Silently delete `document.pdf`. 2. Upload `malicious.pdf`, which creates a new, timestamped event: {`"action": "upload", "file_hash": "xyz...", ...`}.
3. **Detection:** When replaying the event chain, the auditor will see:
   (a) `... "timestamp": "T1", "action": "upload", "file_hash": "abc..."`
   (b) *(... other events ...)*
   (c) `... "timestamp": "T2", "action": "upload", "file_hash": "xyz..."`

The auditor can also see that `document.pdf` (hash "abc...") is no longer served by the PRS. This is simply a Silent Deletio (for "abc..."), which is detected as described above. The attacker cannot backdate the new upload ("xyz...") to time T1, as this would require breaking the HSM's secure hash chain. The replacement is obvious.

6 Implementation and Simulation

This section presents a minimal Python-based simulation of the proposed architecture that demonstrates multi-file handling, versioning, and event chaining across batches. The implementation uses the `cryptography` library for production-grade ECDSA signatures while abstracting hardware and network complexities.

6.1 File Structure and Components

The simulation consists of five Python modules:

- `utils.py`: Implements SHA-256 hashing, ECDSA P-384 signing/verification, and Merkle tree construction. Provides cryptographic primitives for the entire system.
- `hsm_sim.py`: Simulates a hardware security module that encapsulates the private signing key and maintains the `Latest_Event_Hash` chain head. The `chain_and_sign()` method performs atomic chaining and signing operations.

- `server.py`: Contains the `PublicRecordsServer` class that handles file uploads, deletions, batching, and witness publication. Maintains an in-memory event list as a list of 3-tuples `(event, chain_hash, signature)` and stores files in `uploads/` using content-addressed naming.
- `client.py`: Implements the `Verifier` class that performs four-step client-side verification and audits missing files against witness logs. Searches through all historical batches to locate events.
- `main.py`: Driver script that orchestrates a multi-file attack scenario, demonstrating event chaining and versioning.

6.2 Class and Function Definitions

The `PublicRecordsServer` class exposes the following methods:

- `upload(file_path)`: Calculates the file hash using SHA-256, stores content at `uploads/{hash}`, creates an event object, invokes the HSM simulator for signing and chaining, and returns a Signed Upload Receipt containing the event and signature.
- `delete(file_hash)`: Removes the file from object store and creates a signed delete event.
- `batch_and_publish()`: Constructs a Merkle tree from accumulated event hashes, retrieves the final chain hash from the HSM simulator, creates and signs a batch header, and publishes the complete package to three witness log files. The batch header includes the Merkle root, final chain hash, and hash of the previous batch header.
- `download(file_hash)`: Searches through all witness logs and all historical batches to locate the upload event for the requested file, then returns a verification package containing the file content, event, signature, and the batch that contains the event.

The `Verifier` class provides:

- `verify_download(package)`: Executes the four-check verification protocol: content integrity hash comparison, event signature validation, Merkle proof verification, and witness log existence check. The verification succeeds only if the file remains untampered since its upload batch.
- `audit_missing_file(file_hash)`: Searches witness logs across all batches to detect silent deletions by finding upload events without corresponding delete events.

6.3 Algorithmic Execution Walkthrough with Event Chain

The simulation demonstrates a realistic multi-document workflow with versioning and active attack detection:

1. **Multiple File Uploads**: Three distinct files (`doc1.txt`, `doc2.txt`, `doc3.txt`) are uploaded sequentially. Each upload generates a separate event signed by the HSM simulator. The event is appended to the server's in-memory event list. The HSM's `Latest_Event_Hash` updates with each operation, creating a cryptographic chain: `Hash_1` $\rightarrow Hash\_2 \rightarrow Hash\_3$.

2. **Batch Publication**: The server packages all three events into **Batch 0**, constructs a Merkle tree from their hashes, and publishes the signed batch header along with all events to three witness logs. The batch header's `final_chain_hash` reflects the HSM state after the third event, anchoring the entire sequence.

3. **Document Update**: The first document is modified and re-uploaded, creating a **new event** with a different hash. This represents versioning—the original version remains in Batch 0, while the updated version appears as a separate upload event. The HSM chain extends to `Hash_4`, and the server's event list now contains only this new event.

4. **Second Batch Publication**: The update event is packaged into **Batch 1**. The batch header includes the previous batch's header hash, creating an inter-batch chain: `Batch 0 → Batch 1`. This links the entire history while maintaining efficient verification.

5. **Verification Across Batches**: When verifying the third file (which remains unchanged since Batch 0), the client searches through all historical batches and witnesses to locate its original upload event. The verification package includes the Batch 0 data, allowing the client to validate the file against its original event. Check 3 succeeds because the event is found in the Merkle tree of Batch 0, demonstrating that verification works across multiple batches.

6. **Tampering Attack Detection**: After successful verification, the simulation directly modifies the content of the third file in the `uploads/` directory, simulating an attacker tampering with stored data. When the client attempts re-verification, **Check 1 (Content Integrity)** immediately fails because the recalculated SHA-256 hash of the tampered content does not match the expected hash recorded in the Batch 0 event. The signature remains valid, but the file content is proven corrupted.

7. **Silent Deletion Attack Detection**: The simulation then silently deletes the second file from the object store without creating a corresponding delete event, simulating an insider attack or storage compromise. When the client attempts to download the file, the server returns `None` because the file is missing. The audit function searches all witness logs and confirms that an **upload event exists** for the file hash in Batch 0, but **no delete event** exists in any subsequent batch. This evidence proves the file was uploaded but vanished without authorization, distinguishing legitimate deletions from malicious ones.

8. **Chain Hash Progression**: The simulation outputs the chain hash after each event, showing the cryptographic accumulation: each hash depends on all previous events. The batch finalization captures this state, making the entire history tamper-evident. The final summary explicitly reports both attacks as *detected*, validating the system's security guarantees.

9. **Final State Preservation**: The object store contains all file versions (original and updated) except for the silently deleted file, demonstrating that the system preserves data and detects anomalies. Witness logs maintain the complete event history across both batches, providing immutable evidence for forensic analysis.

6.4 Cryptographic Correctness and Limitations

The simulation uses the `cryptography` library's ECDSA P-384 implementation with SHA-256 hashing. Merkle trees follow the standard binary construction with concatenated sibling hashes. The limitations of this simulation are that all operations happen locally, simplified Merkle proof generation, and absence of network latency or witness synchronization challenges.

The implementation successfully demonstrates that the architecture provides end-to-end integrity through cryptographic chaining, public verifiability via Merkle trees and witness logs, and accountability via non-repudiable event signatures, while having a realistic performance.

7 AI Use

AI help was sought in the completion of this assigment– particularly for brainstorming various ideas and background research for this idea (finding real world cases, issues, exploring blockchain based security etc.). AI help was also sought in typesetting this report and implementing some of the cryptographic operations in `hsm_sim.py`. However, all major ideas, architecture design, and protocol design are my own.

References

[1] TNN. (2025, Nov 13). *Teachers got govt jobs using fake mark sheets; MPSTF registers case against 8.* The Times of India. Retrieved from https://timesofindia.indiatimes.com/city/bhopal/teachers-got-govt-jobs-using-fake-mark-sheets-mpstf-registers-case-against-8/articleshow/125282997.cms

[2] The Hindu. (2023, May 21). *HC refuses relief to woman submitting forged documents for employment.* Retrieved from https://www.thehindu.com/news/cities/Delhi/hc-refuses-relief-to-woman-submitting-forged-documents-for-employment/article66877249.ece

[3] Indian Kanoon. (n.d.). *Search: "forged documents".* Retrieved from https://indiankanoon.org/search/?formInput=forged%20documents

[4] U.S. National Archives. (n.d.). *Unauthorized Disposition of Federal Records.* Retrieved from https://www.archives.gov/records-mgmt/resources/unauthorizeddispositionoffederalrecords

[5] CISA. (2022). *Insider Threat Mitigation Guide.* Cybersecurity and Infrastructure Security Agency. Retrieved from https://www.cisa.gov/sites/default/files/2022-11/Insider%20Threat%20Mitigation%20Guide_Final_508.pdf