Università degli Studi di Messina

VERIFYING DATA INTEGRITY WITH A PRIVATE BLOCKCHAIN: A COURSE SCHEDULE USE CASE

Professor: Massimo Villari
Project Advisor: Mario Colosi

NOVEMBER 17, 2025

UNIVERSITY OF MESSINA
Student: Hamidy Adeebullah
Matricola: 539745

Table of Contents

Abstraction

This project tackles the problem of guaranteeing data integrity by developing a minimal, privacy-preserving blockchain system. The goal is to demonstrate how blockchain mechanisms can enhance traditional hashing techniques by adding structure, provenance, and tamper-evidence to a data integrity workflow. In this implementation, the blockchain does not serve as a public or decentralized ledger; instead, it functions as a lightweight, server-managed integrity chain that records a verifiable history of changes to a dataset.

The server-side application, built using the Flask framework, oversees two key files: *data.json*, which contains the primary dataset, and *blockchain.json*, which stores the integrity ledger. Whenever the dataset is modified, the system generates a SHA-256 hash of the updated data. This hash is written into a newly created block that includes essential metadata such as a timestamp and the hash of the previous block, thereby forming a linked, append-only structure. Each new block strengthens the chain's immutability by ensuring that any attempt to manipulate historical data would require recalculating and altering all subsequent blocks—an operation that is computationally evident even in this simplified setting.

A separate client application (*client.py*) interacts with the server to retrieve both the dataset and its corresponding blockchain. The client performs independent verification by recalculating hashes and validating the chain's continuity. If the ledger is intact and the data hash matches the most recent block, the client can confidently confirm that the dataset has not been tampered with.

To illustrate the approach in a meaningful context, the project uses a university course schedule as its practical example dataset. This scenario highlights how even small, locally managed systems can benefit from blockchain-based integrity tracking. The result is a clear, secure, and demonstrably robust validation framework that showcases how blockchain principles can extend traditional hashing methods to provide stronger guarantees of data authenticity and tamper detection.

Introduction

The primary objective of this project was to integrate a basic, private blockchain implementation into a data integrity workflow to demonstrate how blockchain principles can enhance traditional verification methods. While conventional hashing allows data to be validated at a single point in time, it does not inherently preserve a trustworthy history of changes. This project addresses that limitation by employing a blockchain-style immutable ledger, enabling the system to detect tampering not only with the current dataset but also with its historical states. By chaining each new hash to the previous one, the blockchain provides a verifiable timeline of modifications, making any unauthorized manipulation immediately evident.

To contextualize the system, the project applies this approach to a simple, practical scenario: a university course schedule. Although small in scale, this data is highly sensitive to integrity issues—any unauthorized modification to a lesson time, lecturer, or room allocation could cause widespread confusion. This use case illustrates how even routine administrative data can benefit from stronger integrity safeguards.

The solution is built using a Python Flask server (*server.py*) that serves as the authoritative source for both the data and the blockchain ledger. The server maintains two core files:

- data.json, which stores the current version of the course schedule, and
- blockchain.json, which acts as the append-only integrity ledger.

When an authorized update is received through the server's API, the system writes the new data to *data.json*, computes its SHA-256 hash, and appends a new block to the ledger. Each block includes metadata such as a timestamp, the computed hash, and a reference to the previous block's hash, ensuring continuity and tamper-evidence across the entire chain.

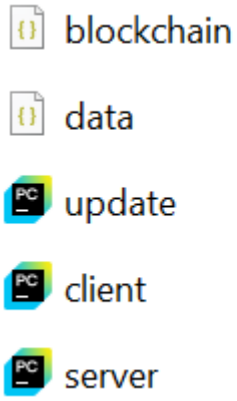
To provide independent verification, a standalone Python client (*client.py*) retrieves both the current dataset and the full blockchain from the server. The client recalculates the expected hashes, checks the validity of the chain, and confirms that the latest block accurately reflects the data being served. If any block or data value has been altered, the client's verification process will detect the discrepancy.

Overall, this project demonstrates how a minimal private blockchain can complement traditional hashing techniques, offering a more robust, transparent, and temporally aware approach to data integrity validation.

1. System Architecture and Core Concept

Component Overview

The system is built around a set of key files and scripts that work together to maintain and verify data integrity using a private blockchain approach. Each component plays a specific role in the overall workflow, and together they form a complete pipeline for updating, recording, storing, and validating changes to the protected dataset.



- **server.py**
  This file contains the Flask-based server application that serves as the central authority for both the data itself and the blockchain ledger. It exposes API endpoints that allow external tools or users to interact with the system. When a request is made to retrieve data, the server responds with the contents of the dataset. When an authorized request arrives to update the data, the server handles the modification, rewrites the data file, generates the appropriate hash, and appends a new block to the ledger. This makes the server responsible for all controlled updates and ensures that no unauthorized process can write directly to the protected files.

- **client.py**
  This script acts as an independent verifier, separate from the server. It sends requests to the server to obtain both the dataset and the full blockchain. After receiving them, it performs a series of verification steps on its own machine, without relying on the server's integrity. The client recomputes hashes, checks the internal structure of the chain, confirms the relationships between blocks, and ensures that the most recent block correctly represents the current data. This separation provides an additional layer of trust: even if the server were compromised, the client could detect discrepancies.

- **data.json**
  This file represents the actual dataset being protected. In this project, it contains the course schedule used in the example scenario. This file is rewritten

whenever legitimate updates occur. Because this information is expected to remain consistent and trustworthy, it is essential that any unauthorized changes become detectable, which is the purpose of connecting it to the blockchain.

- **blockchain.json**
  This file stores the ledger that records the full history of the data's integrity. Each entry in the ledger is a block containing the hash of the data at the time it was updated, along with related information that connects it to previous blocks. By maintaining an ordered list of these blocks, the system creates a traceable, tamper-evident history that reflects every authorized modification made to the dataset.


```
[
  {
    "index": 0,
    "timestamp": 1762268713,
    "data_hash": "GENESIS",
    "previous_hash": "0000000000000000000000000000000000000000000000000000000000000000",
    "block_hash": "1f2ade380b89f1ea7d46e345a42bdda757027066103c0ca149355397d81685e6"
  },
  {
    "index": 1,
    "timestamp": 1762275734,
    "data_hash": "bed33ec41744b2dbb806a1dd236e106a193d05fc1ffe174ddef57e04e7b4d255",
    "previous_hash": "1f2ade380b89f1ea7d46e345a42bdda757027066103c0ca149355397d81685e6",
    "block_hash": "38c30eb72da3f036a0011b36104f1f6ce90de6bb327ba270b87ca67e7a06ae5b"
  },
  {
    "index": 2,
    "timestamp": 1762276031,
    "data_hash": "9051c5ab013805ff76048c9ea63db1ce560b545ac6e3019460a099783a516f47",
    "previous_hash": "38c30eb72da3f036a0011b36104f1f6ce90de6bb327ba270b87ca67e7a06ae5b",
    "block_hash": "ca03d558979ba902f363f23cccf48fce11be6693c59a3b2df6bd018932064460"
  }
]
```


- **update.py**
  This utility script exists to demonstrate how updates to the data can be performed programmatically. It sends a request to the server's API, triggering the same process that would occur if an administrator manually changed the data. This script helps show how external tools can interact with the system in a controlled manner.

Architecture Diagram (Description Only)

The intended architecture diagram for this project would visually represent the way the components communicate and how information flows between them. Although no graphical diagram is shown here, the structure can be described through the following relationships:

- An administrative tool such as update.py initiates a data modification by sending a POST /api/data request to the server.
- The server receives this request, updates data.json, and writes a new block into blockchain.json to record the updated data hash.
- The server stores both files locally, keeping the official versions of the data and the ledger.
- The client application uses two read-only endpoints, GET /api/data and GET /api/chain, to fetch the current state of both files.
- Once it obtains this information, the client performs its own verification procedure to ensure the blockchain is valid and that the dataset corresponds to the most recent block.

If drawn visually, arrows would show the update path from the administrative tool to the server and arrows pointing from the server to the client for data retrieval. A final arrow inside the client would show the verification process happening locally.

Core Concepts

Canonical Hashing

To ensure that the hash of the dataset is consistent and repeatable, the system uses canonical JSON formatting when generating hashes. This means that before hashing, the JSON data is serialized in a way that always produces the same string representation. Keys are sorted, unnecessary whitespace is removed, and the resulting text is fully standardized. This step is essential because JSON data can be identical even when the formatting differs. Without canonicalization, the same data could produce different hashes depending on spacing, indentation, or key order. By using a canonical JSON encoder, both the server and the client produce the exact same hash for the same data, enabling accurate and reliable comparisons.

The Hash-Chained Ledger

The blockchain ledger stored in blockchain.json is a simple but functional implementation of a hash-chained structure. It begins with an initial Genesis block, which acts as the base of the chain. Every block that follows includes a set of fields:

- The index, representing the block's position in the list.
- A timestamp, marking when the block was created.
- The data_hash, which is the SHA-256 hash of the entire data.json file at that moment.
- The previous_hash, which stores the hash of the block directly before it.
- The block_hash, which is the SHA-256 hash of the block's own contents, including the previous hash.

These fields ensure that each block is linked to the one before it. Because the block_hash depends not only on the block's own fields but also on its reference to the previous block, the entire chain becomes a connected sequence where each block reinforces the validity of the one before it.

How Chaining Creates Immutability

The system becomes tamper-evident because of how each block refers to the hash of the previous block. If someone were to alter any part of an older block—for example, modifying the data_hash inside Block 1—the block_hash for that block would change as well. This new block_hash would no longer match the previous_hash stored inside Block 2. As a result, Block 2 would become invalid. Every block after Block 2 would then also be invalid because their chains of dependency would be broken.

This cascading effect means that even a small unauthorized modification becomes immediately detectable simply by checking the hashes in sequence. The client's verification function specifically looks for this type of mismatch. By verifying each block one by one, recomputing hashes, and ensuring that the final data matches the last block, the client confirms whether the chain remains intact. If any block has been altered, the client identifies the problem without needing to trust the server.

2. Implementation Details

The Server (server.py)

The server functions as the central authority that manages both the main dataset and the private blockchain ledger. It is responsible for receiving updates, storing the live data, and ensuring that every modification is permanently recorded in the chain. To support this workflow, the server exposes four primary API endpoints that external applications, scripts, or clients can interact with.

The GET /api/data endpoint provides read-only access to the current contents of the data.json file. Whenever a user or client requests this endpoint, the server simply loads the data file from storage and returns the exact JSON structure that represents the latest version of the course schedule or other protected information.

```python
@app.route("/api/data", methods=["GET"])  # get current dataset
def api_get_data():
    ensure_chain_initialized()
    data = load_data()
    return jsonify(data)
```

The POST /api/data endpoint handles the full update process, which includes both changing the data and recording that change in the blockchain. When new content is submitted to this endpoint, the server begins by receiving and validating the JSON payload provided by the requester. Before saving anything, the server automatically attaches a new last_update timestamp to the data, marking the exact moment the modification occurred. Once the timestamp is added, the updated data is written to data.json, replacing the previous version.

```python
@app.route("/api/data", methods=["POST"])  # update dataset and append a block
def api_update_data():
    ensure_chain_initialized()
    try:
        payload = request.get_json(force=True)
    except Exception:
        return jsonify({"error": "invalid_json"}), 400

    if not isinstance(payload, dict):
        return jsonify({"error": "payload_must_be_object"}), 400

    # Update "last_update" for visibility
    payload["last_update"] = int(time.time())
    save_data(payload)

    # Record integrity
    dh = compute_data_hash(payload)
    new_block = append_block(dh)
    return jsonify({"ok": True, "block": new_block})
```

Immediately after saving the updated data, the server computes a canonical hash of the new dataset. This hashing process uses a standardized form of JSON representation to ensure that identical data always produces the same hash value. After computing the hash, the server appends a new block to the blockchain ledger. This block contains the freshly computed data hash along with essential metadata such as the timestamp and reference to the previous block. In this way, every data update creates a permanent record within blockchain.json.

The GET /api/chain endpoint returns the entire blockchain ledger as it currently exists. Any client or external system can request this endpoint to obtain the full history of recorded data states.

```python
@app.route("/api/chain", methods=["GET"])  # get the full chain
def api_get_chain():
    ensure_chain_initialized()
    return jsonify(load_chain())
```

The GET /api/verify endpoint allows the server to validate its own data and chain. When called, the server runs an internal consistency check to confirm that the blockchain is valid and that the current data still corresponds to the hash stored in the latest block. The server then returns the result as a structured JSON message, indicating whether the system's integrity is intact.

```python
@app.route("/api/verify", methods=["GET"])  # verify chain + data match
def api_verify():
    ensure_chain_initialized()
    chain = load_chain()
    data = load_data()

    chain_ok = verify_chain_integrity(chain)
    if not chain_ok.get("ok"):
        return jsonify({"ok": False, "where": "chain", **chain_ok}), 200

    if len(chain) < 2:
        # Only genesis exists: treat as no recorded data yet
        return jsonify({"ok": True, "note": "no_data_blocks_yet"}), 200

    last = chain[-1]
    dh = compute_data_hash(data)
    if dh != last["data_hash"]:
        return jsonify({"ok": False, "where": "data_hash_mismatch", "expected": last["data_hash"], "got": dh}), 200

    return jsonify({"ok": True, "where": "all_good", "last_block": last}), 200
```

The Client (client.py)

The client acts as an independent verification tool, designed to operate without relying on the server's own statements about its integrity. Instead of trusting what the server claims through its internal verification endpoint, the client downloads both the data and the blockchain and performs all validation steps locally. This trustless approach ensures that tampering can be detected even if the server itself has been compromised.

```python
if __name__ == "__main__":
    data = requests.get(f"{BASE}/api/data").json()
    chain = requests.get(f"{BASE}/api/chain").json()


    ok, reason = verify_locally(data, chain)
    print("Local verification:", ok, reason)


# Cross-check server endpoint (should match)
    server_verify = requests.get(f"{BASE}/api/verify").json()
    print("Server verify:", server_verify)
```

The client begins its process by fetching the current dataset using the GET /api/data endpoint and retrieving the full blockchain ledger using the GET /api/chain endpoint. With both elements available, the client moves into a step-by-step verification sequence.

```python
def verify_locally(data, chain):  1 usage

    if not chain:
        return False, "empty_chain"
# Verify genesis
    gcore = {k: chain[0][k] for k in ["index", "timestamp", "data_hash", "previous_hash"]}
    if compute_block_hash(gcore) != chain[0].get("block_hash"):
        return False, "bad_genesis_hash"
# Verify links and block hashes
    for i in range(1, len(chain)):
        prev = chain[i - 1]
        curr = chain[i]
        if curr["previous_hash"] != prev["block_hash"]:
            return False, f"bad_link_at_{i}"
        core = {k: curr[k] for k in ["index", "timestamp", "data_hash", "previous_hash"]}
        if compute_block_hash(core) != curr.get("block_hash"):
            return False, f"bad_block_hash_at_{i}"
    if len(chain) >= 2:  # if we have blocks beyond genesis
        last = chain[-1]
        dh = compute_data_hash(data)
        if dh != last["data_hash"]:
            return False, "data_hash_mismatch"
    return True, "ok"
```

The first step is to verify the integrity of the Genesis block, the very first block in the chain. The client recomputes the hash of the Genesis block based on its stored contents and checks whether this computed value matches the block_hash recorded in the ledger. This step confirms that the foundation of the ledger has not been altered.

Next, the client proceeds to verify the rest of the chain. It iterates through all blocks from the second block onward. For each block, the client performs two essential checks. The first is the link check, which ensures that the previous_hash value stored within the block correctly matches the block_hash of the block that came before it. This verifies the continuity of the chain. The second is the block hash check, in which the client recomputes the block's own hash from its internal fields and compares it to the stored block_hash. Any discrepancy indicates tampering.

If all blocks pass these chain-level checks, the client finally verifies the live data itself. It computes the canonical hash of the data retrieved from the server and compares this value to the data_hash stored in the final block of the blockchain. If these two hashes match, it confirms that the current data.json file truly corresponds to the most recent block in the chain.

If every stage of the verification process succeeds, the client reports the result as True, "ok", indicating that both the blockchain and the data remain intact and unmodified.

Tamper-Evidence Demonstration

The system's behavior can be observed through several practical scenarios that highlight its ability to detect unauthorized modifications. These demonstrations show how the client responds when data or blocks are altered outside the proper update workflow.

Scenario: Successful Verification with Unmodified Files

When the server is running normally and none of the project files have been altered manually, the client is able to retrieve both data.json and blockchain.json in their

original, valid state. The hash of the current data—including its recorded last_update timestamp—will exactly match the data_hash stored in the final block of the chain. All block links remain valid, and all block hashes recompute correctly during verification. As a result, the client reports a successful outcome, confirming that both the chain and the data are fully consistent.

Scenario: Detecting Unauthorized Modification of data.json

If someone manually alters the contents of data.json while the server is offline, the integrity of the system is broken. For example, changing a lesson's room value from "A3" to "B1" changes the structure of the dataset. When the server is restarted and the client performs verification, the client calculates a new hash for the modified data, which will not match the data_hash stored in the most recent block of the ledger. This mismatch is detected during the final verification step, causing the client to report a failure with a data_hash_mismatch status.

Scenario: Detecting Unauthorized Modification of blockchain.json

If an attacker manually edits the blockchain ledger—such as modifying the data_hash inside one of the earlier blocks—the chain becomes corrupted. When the client verifies the ledger, it recomputes the block hash for the altered block. The computed value will no longer match the stored block_hash, breaking the chain's integrity. Additionally, because each block depends on the one before it, the next block will also fail during its previous_hash comparison. This causes the client to flag the exact point in the chain where the inconsistency appears, reporting an error such as bad_link_at_2 or bad_block_hash_at_1 depending on which part was tampered with.

Conclusion

Summary of Findings

This project successfully fulfills the core requirements outlined in the brief by demonstrating a functioning, minimal implementation of a private blockchain used specifically for data integrity purposes. Through this system, it becomes clear how a hash-chained ledger can enhance traditional hashing methods by providing not only verification at a single point in time but also a chronological, tamper-evident history of changes. The design shows that even a lightweight blockchain can significantly strengthen integrity guarantees when combined with canonical hashing techniques.

The chosen use case—a university course schedule—proves to be an effective and approachable example for illustrating how the system works in practice. Because the course schedule contains information that must remain accurate, such as times and room assignments, it serves as a meaningful demonstration of why data integrity matters. Any unauthorized modification to such data could lead to confusion or disruption, making it an ideal scenario for showcasing the benefits of the implemented blockchain-based validation approach. Overall, the project clearly shows how each update can be recorded, traced, and independently verified, meeting the intended learning outcomes of the assignment.

Limitations

Although the system performs reliably within its intended scope, it does have several intrinsic limitations that stem from its minimal and centralized design. As described in the project's documentation, the implementation is deliberately simple, and this simplicity brings constraints that affect the overall security model.

The most significant limitation is the absence of any form of consensus mechanism. The project enhances a basic integrity check by attaching a small blockchain ledger, but the ledger operates entirely under the authority of a single server. Without mining, voting, or validation across multiple independent nodes, the blockchain does not provide decentralization, which is a key feature in real-world blockchain systems.

Because of this, the server becomes a single point of trust—and also a single point of failure. If an attacker gains full control over the server, they could modify both data.json and blockchain.json at the same time. After altering both files, the attacker could recompute all relevant hashes and regenerate the entire chain in a consistent state, thereby defeating the integrity checks. The client's verification process is only capable of detecting tampering that occurs in an uncoordinated manner, such as when an

attacker alters only the data or only the ledger. Coordinated changes made directly on the server fall outside the protection offered by this minimal implementation.

These limitations highlight the difference between a simple educational blockchain model and a full decentralized blockchain system used in production environments.

Future Enhancements

Several improvements could be made to overcome the limitations of the current system and extend its capabilities. One important direction for future development is the introduction of distributed consensus. By creating multiple nodes that each maintain their own copy of the chain, and by requiring agreement among them before blocks are accepted, the system would no longer rely on a single authority. A basic form of Proof-of-Work or Proof-of-Authority could be introduced to ensure that no single compromised machine could rewrite the entire ledger without detection.

Another enhancement would be to incorporate public-key cryptography into the update workflow. If all changes to the data had to be digitally signed using an administrator's private key, then only authorized users could create valid updates. Even if someone gained access to the server, they would still be unable to forge new blocks unless they also held the private key.

The project could also benefit from adopting Merkle trees as part of the hashing structure. Instead of hashing the entire dataset as one block of information, a Merkle tree would break the data into components, hash each part, and combine those hashes into a final root hash. This approach would allow for partial verification, lower computational cost for large datasets, and more flexible integrity checks.

Taken together, these future enhancements would provide greater decentralization, stronger authentication, and more efficient verification, significantly improving both the security and scalability of the system.