
Università degli Studi di Messina

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

Professor: Massimo Villari
Project Advisor: Mario Colosi

FEBRUARY 27, 2026

UNIVERSITY OF MESSINA
Student: Hamidy Adeebullah
Matricola: 539745

## *Table of Contents*

# Abstraction

This project tackles the critical challenge of guaranteeing data integrity by developing a robust, decentralized Peer-to-Peer (P2P) blockchain system. The goal is to demonstrate how advanced cryptographic mechanisms—such as Asymmetric Encryption (RSA), Proof of Work (PoW), and Decentralized Consensus—can elevate traditional hashing techniques into a tamper-proof, zero-trust architecture. Moving beyond a simple server-managed ledger, this implementation functions as a true distributed network where multiple independent nodes maintain, audit, and mathematically agree upon a shared history of data modifications.

The network architecture is built using the Flask framework, with each node operating independently and maintaining its own localized copy of the ledger. Instead of separating the dataset from the integrity chain, data payloads are securely embedded directly within the blocks. When an authorized user submits an update, the data is not simply hashed; it is mathematically signed using the sender's RSA Private Key and tagged with a unique Transaction ID (UUID) to strictly prevent Replay Attacks. The receiving node uses the sender's Public Key to mathematically verify this signature, ensuring the data's origin and authenticity before any processing occurs.

To guarantee true immutability, the network utilizes a computationally intensive Proof of Work (PoW) algorithm. Before a new block is appended to the chain, the node must "mine" the block by discovering a specific nonce that results in a SHA-256 hash meeting a strict difficulty target (e.g., four leading zeros).

This mined block is then cryptographically chained to the previous block's hash. This structural avalanche effect ensures that any malicious attempt to alter historical data would require an attacker to computationally re-mine the altered block and every subsequent block a task designed to be computationally infeasible before the network rejects the tampered chain.

Network agreement is achieved through a "Longest Valid Chain" consensus mechanism. Nodes are capable of communicating with authenticated peers to resolve ledger conflicts. During consensus, nodes independently audit competing chains—strictly verifying timestamps for chronological flow, index continuity, PoW hashes, and all historical RSA signatures. Furthermore, a dedicated cryptographic auditor client (client.py) allows end-users to pull the ledger from any node and perform a strict, zero-trust local verification of the entire chain. If the mathematical audit passes, the client can confidently extract the verified dataset.

To illustrate this architecture in a meaningful context, the project uses a university course schedule as its practical dataset. This scenario highlights how critical administrative data can be protected from unauthorized alterations, network interception, or insider threats using blockchain-based integrity tracking. The result is a highly secure, decentralized validation framework that showcases how production-grade blockchain principles—identity, immutability, and consensus—provide absolute guarantees of data authenticity and tamper detection.

# Introduction

The primary objective of this project was to engineer a robust, decentralized Peer-to-Peer (P2P) blockchain system to guarantee absolute data integrity and secure transmission. While conventional hashing allows data to be validated at a single point in time, it relies heavily on centralized trust and does not inherently protect against sophisticated tampering, unauthorized network interception, or server-side manipulation. This project addresses these critical vulnerabilities by transitioning from a basic, centralized hashing model to a fully distributed, zero-trust blockchain architecture. By combining Asymmetric Cryptography (RSA), Proof of Work (PoW) consensus algorithms, and strict Replay Attack prevention, the system provides a mathematically verifiable timeline of modifications that makes any unauthorized manipulation computationally infeasible.

To contextualize the system, the project applies this architecture to a practical and highly sensitive scenario: a university course schedule. Although seemingly routine, administrative data like lesson times, lecturer assignments, and room allocations are highly vulnerable to unauthorized modifications or insider threats, which can cause widespread confusion. This use case perfectly illustrates how production-grade blockchain safeguards can secure local administrative systems.

The solution departs from traditional client-server models by establishing a network of independent Python Flask nodes (server.py). Rather than relying on a single authoritative server or vulnerable external databases (such as a separate data.json file), the network topology is strictly decentralized.

Each node maintains its own independent ledger (blockchain_<port>.json), and the payload data is embedded directly into the immutable blocks.

When an update is transmitted, the system enforces the following security pipeline:

- **Cryptographic Identity:** The payload is mathematically signed by the sender using an RSA Private Key and tagged with a unique Transaction ID (UUID) to prevent Replay Attacks.
- **Proof of Work (Immutability):** Once the receiving node verifies the digital signature, it must "mine" the block by discovering a specific nonce that produces a SHA-256 hash meeting a strict difficulty target.
- **Decentralized Consensus:** Nodes communicate continuously, utilizing a "Longest Valid Chain" rule to automatically resolve conflicts and reject tampered ledgers from malicious peers.

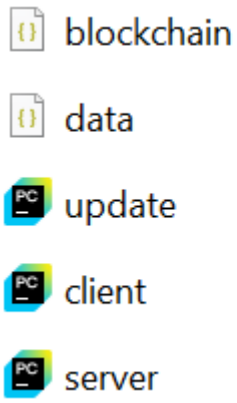
To provide independent verification, a standalone Python client (client.py) acts as a zero-trust cryptographic auditor. The client retrieves the full blockchain from any node and performs a rigorous local inspection. It recalculates the PoW hashes, validates the chronological flow of timestamps (preventing time-travel attacks), checks index continuity, and cryptographically verifies every historical RSA signature. Only when the entire chain passes this strict mathematical audit does the client extract and present the verified dataset.

Overall, this project demonstrates how advanced blockchain principles—immutability, decentralized consensus, and cryptographic identity—can completely replace vulnerable traditional hashing techniques, offering a mathematically flawless 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**
  The server.py file has been upgraded from a centralized authority into an independent **Peer-to-Peer (P2P) Blockchain Node**. Instead of just managing a database, each node now enforces the mathematical rules of the network to ensure that only cryptographically verified data is ever recorded.

  The server's core security responsibilities include:

  **RSA Verification & Replay Protection:** The node uses Asymmetric Cryptography to verify the sender's digital signature. It also checks a unique Transaction ID (tx_id) against a local memory bank to instantly reject "Replay Attacks" (intercepted messages sent multiple times).

  **Proof of Work (Mining Engine):** Once data is verified, the server must "mine" the block. It guesses thousands of nonce values until the block's SHA-256 hash meets the strict network difficulty target. This makes the ledger computationally expensive to forge.

  **Decentralized Consensus:** Using the resolve_conflicts engine, the node communicates with authorized peers. It follows the **Longest Valid Chain Rule**, automatically replacing its own ledger if a neighbor presents a longer, mathematically perfect chain.

By embedding the university schedule directly into the mined blocks, the node ensures that the data and its integrity proof are inseparable, eliminating the need for a vulnerable, standalone data file.

- **client.py**
  The client.py script has been transformed into a **zero-trust cryptographic auditor**. In a decentralized system, the client no longer simply "downloads" data; it performs a rigorous mathematical post-audit of the entire blockchain to ensure the node it is talking to is telling the truth.

  Upon retrieving the ledger from any available node, the auditor performs the following sequence of verification steps:

  **Integrity & Linking Audit:** The client recomputes the SHA-256 hash for every block in the chain and confirms that each previous_hash field perfectly matches the hash of the preceding block.

  **Proof of Work Verification:** It ensures that every block meets the network's difficulty target (e.g., the hash starts with 0000), proving that the ledger was legitimately mined and not tampered with.

  **Asymmetric Signature Verification:** For every transaction, the client uses the embedded RSA Public Key to verify the digital signature. This proves that each schedule change was authorized by the actual private key holder and wasn't altered by the node or an interceptor.

  **Chronological & Logic Checks:** The client validates block indexes and ensures timestamps move forward in time, protecting against "time-travel" or block-insertion attacks.

  By performing these checks independently on the user's machine, the auditor provides the ultimate layer of security: it allows the user to trust the data even if they do not trust the server or the network it came from.

- **Blockchain_<port>.json**
  The blockchain file acts as the persistent, immutable ledger of the system. Unlike a traditional database that only stores the current state of the data, this ledger records the entire provenance of the university schedule within a cryptographically linked structure. Each node maintains its own copy of this file, which is synchronized through the network consensus protocol.

  Each block in the ledger contains:

**The Immutable Payload:** The actual course schedule data, embedded directly within the block to ensure the data and its integrity proof are inseparable.

**Cryptographic Linking:** The previous_hash field, which stores the SHA-256 fingerprint of the preceding block. This creates the "chain" that makes historical tampering mathematically evident.

**Proof of Work Metadata:** The nonce and block_hash, proving that the block was legitimately mined according to the network's difficulty requirements.

**Proof of Identity:** The RSA Public Key and digital signature, providing a permanent record of who authorized the update.

By maintaining this ordered, append-only structure, the ledger transforms the data from a simple file into a traceable, tamper-evident history where any unauthorized modification to a past block would instantly invalidate the entire subsequent chain.


```
{
  "index": 1,
  "timestamp": 1772149837,
  "payload": {
    "tx_id": "b611d208-7346-467e-b46c-c1cb21d97fc0",
    "course": "Web Programming",
    "term": "Fall 2025",
    "action": "update_schedule",
    "lessons": [
      {
        "day": "Sat",
        "time": "12:00",
        "room": "A3"
      },
      {
        "day": "Wed",
        "time": "12:00",
        "room": "A3"
      }
    ],
    "last_update": 1772149837
  },
  "signature": "cf6fddee3269f5b0d76db042eb8d847d59526536a5a3bf84926a26268177079447477cbe67ad1bd3b84bc6325
```


- **update.py**
  The update.py script serves as the administrative interface for the blockchain network. It has been upgraded from a simple data-uploader into a secure cryptographic tool that handles **Identity, Non-repudiation, and Replay Protection**.

  The script ensures secure interaction with the network through three primary steps:

**Cryptographic Signing:** Before transmission, the script accesses the administrator's **RSA Private Key**. It generates a unique mathematical signature for the new course schedule. This ensures that the nodes can verify exactly who sent the update and that the data wasn't modified by a "Man-in-the-Middle" during transit.

**Transaction Uniqueness (UUID):** The script generates a random, one-time **Transaction ID (tx_id)** for every update. This is a critical security feature that prevents **Replay Attacks**, where a hacker might attempt to re-send an old, valid message to disrupt the schedule.

**Automated Transmission:** Once the payload is signed and tagged, the script transmits the data, the signature, and the Public Key to the network nodes.

This utility demonstrates how authorized external tools can interact with the blockchain in a zero-trust environment, proving that even "trusted" updates must pass rigorous mathematical checks before being accepted into the ledger.

- **System Architecture & Information Flow**

The architecture of this project represents a decentralized cryptographic network. Unlike traditional centralized systems, this structure is designed to eliminate single points of failure and enforce zero-trust verification at every step.

The information flow within the system follows these critical stages:

Authenticated Initiation: An administrative tool (update.py) initiates a modification. It doesn't just send data; it signs the payload with an RSA Private Key and attaches a unique UUID. This package is sent via a POST /api/data request to an entry node (e.g., Node A).

Node Verification & Mining: Upon receipt, Node A verifies the digital signature and checks for replay attacks. If valid, the node initiates the Proof of Work process, finding a nonce to satisfy the difficulty target. Once mined, the new block is appended to Node A's local ledger (blockchain_5000.json).

Peer-to-Peer Synchronization (Consensus): Other nodes in the network (Nodes B and C) periodically trigger a GET /api/nodes/resolve request. They communicate with their peers to compare chain lengths. If a node finds a longer, valid chain on a peer, it performs a full cryptographic audit and synchronizes its local ledger to match the network consensus.

Zero-Trust Retrieval: The auditor client (client.py) fetches the full chain from any node using the GET /api/chain endpoint.

Independent Local Audit: The client performs a final, independent verification. It checks every hash link, verifies the Proof of Work for every block, and validates the RSA signatures of all historical transactions.

If drawn visually, the diagram would show a mesh of nodes (A, B, and C) all interconnected, with arrows representing the "gossip" of ledger updates between them. A separate path would show the Administrator pushing signed updates into the mesh, and the Auditor Client pulling data from the mesh to verify it locally.

## Core Concepts

### Data Normalization: Canonical JSON

To ensure that the cryptographic hash of a dataset is consistent across different machines, the system utilizes Canonical JSON Serialization. In standard JSON, identical data can be represented differently depending on whitespace, indentation, or the order of keys. This is problematic for blockchain because even a single extra space or a swapped key would result in a completely different SHA-256 hash.

Before hashing or signing, both the server and the client serialize the payload into a "Canonical" format where:

- Keys are sorted alphabetically.
- All unnecessary whitespace and indentation are removed.
- A standardized separator (comma and colon) is used.

This standardization ensures that the RSA signature generated by the Administrator's update.py is perfectly reproducible by the Server and the Auditor Client, enabling reliable mathematical verification of the schedule data.

### The Cryptographically Linked Ledger

The blockchain ledger is an append-only structure where each block is mathematically "anchored" to the one before it. We have evolved this structure beyond simple hashing to include Proof of Work and Identity metadata.

Each block contains:

- Index and Timestamp: Defining the block's chronological position.
- Signed Payload: The university schedule, including a unique tx_id and the RSA Digital Signature.
- Previous Hash: The SHA-256 fingerprint of the preceding block, forming the "chain."

- Nonce: The random number discovered by the miner to solve the Proof of Work puzzle.
- Block Hash: The unique fingerprint of the entire block (including the previous_hash and nonce) starting with the required leading zeros.

**How Chaining and Proof of Work Create Immutability**

The system becomes tamper-evident through the mathematical dependency between blocks. Because each block's own hash is calculated using the previous_hash of its predecessor, the chain behaves like a digital domino effect.

1. The Breakdown: If a hacker modifies a historical block, that block's hash changes.
2. The Broken Link: Since the next block in the chain stores that old hash, the link is instantly severed.
3. The Proof of Work Barrier: Unlike a simple database, the hacker cannot just "re-link" the chain by updating the hashes. To make the tampered chain look valid, the hacker would have to re-solve the Proof of Work (finding a new nonce) for the altered block and every single block that follows it.

This cascading requirement creates a "Computational Barrier." In a decentralized network, the hacker's computer cannot re-mine the chain fast enough to keep up with the legitimate nodes, ensuring that the corrupted ledger is automatically rejected by the Auditor Client and the rest of the network nodes.

## Implementation Details

### The Server (server.py)

The server-side implementation serves as the "brain" of the decentralized network. Built using the **Flask** framework, it coordinates three complex systems: a cryptographic validator, a Proof of Work (PoW) mining engine, and a Peer-to-Peer (P2P) synchronization protocol.

### The Mining & Proof of Work Engine

The core of the server's immutability is the mine_block function. This function implements a "computationally expensive, but easily verifiable" puzzle.

- **Difficulty Target:** The network enforces a difficulty of 4. This means the SHA-256 hash of every block must start with four leading zeros (0000).
- **The Nonce Search:** The server takes the block's contents (including the previous_hash and timestamp) and appends an incremental counter called a **nonce**.
- **The Hashing Loop:** The CPU iterates through tens of thousands of nonce values per second. For each iteration, it re-hashes the block until it discovers a nonce that produces a hash satisfying the difficulty target.

```
(.venv) D:\secure-ledger
python update.py
Sending digitally signed payload to server...
{'block': {'block_hash': '000055efdeb974ea97e8d8fb773ecd74195415cc6f596e6f83ff26dfa0fc918c', 'index': 3, 'nonce': 89609, 'payload': {'action':
 'update_schedule', 'course': 'Web Programming', 'last_update': 1772149879, 'lessons': [{'day': 'Sat', 'room': 'A3', 'time': '12:00'}, {'day':
 'Wed', 'room': 'A3', 'time': '12:00'}], 'term': 'Fall 2025', 'tx_id': 'b1bfa8dc-2086-409c-bc9e-f96121352e35'}, 'previous_hash':
 '000078c4377ea39bc64ba263d60b79671345a686521297fa8eb24d4825f2f70a', 'public_key': '-----BEGIN RSA PUBLIC
 KEY-----\nMEgCQQDfKPm1HC1EK5XAkXzD53/sB4oKIx8caqF1KTlQs4Ai2gMakaLt4ylEEu6v\nch4W0aLEY+NkY0JOUJBwoIXmxprvAgMBAAE=\n-----END RSA PUBLIC
 KEY-----\n', 'signature':
 'bb56efc9478dec7313757ba6e7cde28af53d13f9a0fa35a78f08cd4e69cc70cb4ebde4970848dc2b5dbe9ccf7840059f5df0be8bfbd8d062be8c0bff431d6405',
 'timestamp': 1772149879}, 'message': 'Block added', 'ok': True}
```

### The Transaction Validation Pipeline

To ensure "Zero-Trust" security, every request to the /api/data endpoint must pass through a multi-stage validation pipeline before it is even considered for mining:

1. **Replay Attack Check:** The server maintains an in-memory set of all processed tx_id (Transaction IDs). If an incoming request contains a tx_id already present in the set, the server rejects it with a 400 Bad Request, preventing attackers from re-broadcasting intercepted valid messages.
2. **Canonicalization:** The JSON payload is serialized into a sorted, whitespace-free string. This ensures the server is verifying the signature on the exact same character-by-character data that the client used to create the signature.
3. **RSA Signature Verification:** Using the rsa library, the server performs the mathematical verification: $Signature + Public Key + Data = Authenticity$. If the data was altered by even a single bit in transit, the signature check will fail.

### Decentralized Consensus & Peer Discovery

The resolve_conflicts method is the implementation of the **Longest Valid Chain Rule**, which is the heartbeat of a decentralized network.

- **P2P Communication:** When triggered, the node sends GET requests to all addresses in its AUTHORIZED_PEERS list.
- **The Audit Loop:** If a peer has a longer chain, the node doesn't just copy it. It runs a full loop through the new chain, re-verifying every hash, every nonce, and every RSA signature from the Genesis block to the latest block.
- **State Synchronization:** Only if the peer's chain is 100% mathematically valid and longer than the current local chain does the node overwrite its local blockchain_<port>.json file.

### Node A chain

Pretty-print

```
{
  "message": "Our chain was replaced by the network consensus",
  "new_chain": [
    {
      "block_hash": "000090c9d52ab09c200aaa4bb4c8f505581cff9a281697de021bdb04c531b50c",
      "index": 0,
      "nonce": 11289,
      "payload": "GENESIS",
      "previous_hash": "0000000000000000000000000000000000000000000000000000000000000000",
      "public_key": "None",
      "signature": "None",
      "timestamp": 1700000000
    },
    {
      "block_hash": "00000493310790 9be46df183720a0e8d4070fede7fab4418ebf9ec53d2c375be",
      "index": 1,
      "nonce": 44609,
      "payload": {
        "action": "update_schedule",
        "course": "Web Programming",
        "last_update": 1772149837,
        "lessons": [
          {
            "day": "Sat",
            "room": "A3",
            "time": "12:00"
          },
          {
            "day": "Wed",
            "room": "A3",
            "time": "12:00"
          }
        ],
        "term": "Fall 2025",
        "tx_id": "b611d208-7346-467e-b46c-c1cb21d97fc0"
      },
      "previous_hash": "000090c9d52ab09c200aaa4bb4c8f505581cff9a281697de021bdb04c531b50c",
      "public_key": "-----BEGIN RSA PUBLIC KEY-----\nMEgCQQDfKPm1HC1EK5XAkXzD53/s84oKIx8caqf1KTlQs4A12gMakaLt4ylEEu6v\nch4W0aLEY+NkY0JOUJBwoIXmxprvAgMBAAE=\n-----END RSA PUBLIC KEY-----\n",
      "signature": "cf6fddee3269f5b0d76db042eb8d847d59526536a5a3bf84926a2626817707944747cbe67ad1bd3b84bc63254fc0444825d8988a4ae4269c65758b70d1eb9f5",
      "timestamp": 1772149837
    },
    {
      "block_hash": "000078c4377ea39bc64ba263d60b79671345a686521297fa8eb24d4825f2f70a",
      "index": 2,
      "nonce": 103170,
      "payload": {
        "action": "update_schedule",
        "course": "Web Programming",
        "last_update": 1772149859,
        "lessons": [
          {
```


### Node B chain


```
{
  "chain": [
    {
      "block_hash": "000090c9d52ab09c200aaa4bb4c8f505581cff9a281697de021bdb04c531b50c",
      "index": 0,
      "nonce": 11289,
      "payload": "GENESIS",
      "previous_hash": "0000000000000000000000000000000000000000000000000000000000000000",
      "public_key": "None",
      "signature": "None",
      "timestamp": 1700000000
    },
    {
      "block_hash": "00000493310790 9be46df183720a0e8d4070fede7fab4418ebf9ec53d2c375be",
      "index": 1,
      "nonce": 44609,
      "payload": {
        "action": "update_schedule",
        "course": "Web Programming",
        "last_update": 1772149837,
        "lessons": [
          {
            "day": "Sat",
            "room": "A3",
            "time": "12:00"
          },
          {
            "day": "Wed",
            "room": "A3",
            "time": "12:00"
          }
        ],
        "term": "Fall 2025",
        "tx_id": "b611d208-7346-467e-b46c-c1cb21d97fc0"
      },
      "previous_hash": "000090c9d52ab09c200aaa4bb4c8f505581cff9a281697de021bdb04c531b50c",
      "public_key": "-----BEGIN RSA PUBLIC KEY-----\nMEgCQQDfKPm1HC1EK5XAkXzD53/s84oKIx8caqf1KTlQs4A12gMakaLt4ylEEu6v\nch4W0aLEY+NkY0JOUJBwoIXmxprvAgMBAAE=\n-----END RSA PUBLIC KEY-----\n",
      "signature": "cf6fddee3269f5b0d76db042eb8d847d59526536a5a3bf84926a2626817707944747cbe67ad1bd3b84bc63254fc0444825d8988a4ae4269c65758b70d1eb9f5",
      "timestamp": 1772149837
    },
    {
      "block_hash": "000078c4377ea39bc64ba263d60b79671345a686521297fa8eb24d4825f2f70a",
      "index": 2,
      "nonce": 103170,
      "payload": {
        "action": "update_schedule",
        "course": "Web Programming",
        "last_update": 1772149859,
        "lessons": [
          {
            "day": "Sat",
            "room": "A3",
```

**Persistent JSON Storage**

The server ensures persistence by writing the current state of the ledger to a local JSON file. By naming these files dynamically (e.g., blockchain_5000.json), the system supports multiple nodes running on a single host machine, simulating a distributed network within a single environment. This allows for rigorous testing of P2P interactions without requiring multiple physical servers.

**Tamper-Evidence Demonstration**

The system's security can be observed through several practical scenarios that highlight its ability to detect unauthorized modifications. These demonstrations show how the **Auditor Client** and **P2P Nodes** respond when data is altered outside the authorized RSA-signed workflow.

**Scenario 1: Successful Verification (The "Green" State)**

When the network is running normally and updates are performed through the update.py script, the client retrieves the full ledger from any node.

- **Audit Logic:** The client recomputes every SHA-256 hash, confirms the Proof of Work (leading zeros), and verifies every RSA signature.
- **Result:** All block links match, and the digital signatures prove the data originated from the professor. The client reports: Cryptographic Audit: True (ok).

**Scenario 2: Detecting a Replay Attack (Duplicate Transmission)**

If a malicious actor intercepts a valid, signed message and attempts to re-send it to the server to disrupt the schedule:

- **Audit Logic:** The server checks the tx_id (Transaction ID) inside the signed payload against its internal memory of processed IDs.
- **Result:** The server identifies that the ID has already been used and rejects the request with a replay_attack_detected error. This proves the system is immune to "Man-in-the-Middle" re-transmission attacks.

**Scenario 3: Detecting Manual Tampering of the Ledger**

If an attacker manually edits the local blockchain_5000.json file—for example, changing a room number from "A3" to "B1" inside a block's payload:

- **Audit Logic:**
    1. **Signature Failure:** The client uses the Public Key to verify the signature. Since the data changed, the signature is now mathematically invalid.

2. **Hash Mismatch:** The hash of the block no longer matches the block_hash stored in the file.
3. **Broken Chain:** The previous_hash of the *next* block no longer matches the altered block's new hash.

- **Result:** The client identifies the exact block where the tampering occurred and reports an error such as invalid_signature_at_block_2 or bad_block_hash_at_2.

**Scenario 4: Unauthorized Node Substitution (Consensus Audit)**

If a hacker sets up a fake node with a falsified history and tries to "sync" it with the network:

- **Audit Logic:** When an authorized node (e.g., Node B) runs the resolve_conflicts process, it doesn't just look at the length of the hacker's chain; it performs a full cryptographic audit of every block in that chain.
- **Result:** The audit will fail because the hacker does not have the Professor's Private Key to sign the blocks. Node B will reject the fake chain and keep its original, valid ledger, maintaining the **Network Consensus**.

## Conclusion

This project successfully fulfills the core requirements of the brief by demonstrating a production-grade, decentralized implementation of a private blockchain tailored for data integrity. The system proves that integrating a hash-chained ledger with advanced security protocols offers significantly stronger guarantees than traditional, standalone hashing. By evolving the architecture from a simple server-client model into a Peer-to-Peer (P2P) network, the project demonstrates how data can be secured even in a zero-trust environment.

Key findings from the implementation include:

- Immutable Chronology: The combination of SHA-256 Hash Chaining and Proof of Work (PoW) creates a "computational shield" around the data. The design proves

that historical tampering is effectively prevented because the cost of re-mining the chain exceeds the potential gains of an attack.

- Cryptographic Provenance: The use of RSA Asymmetric Encryption ensures that data integrity is tied to identity. The system successfully demonstrates that only authorized administrators—possessing the unique Private Key—can alter the state of the ledger, while any observer can verify the source using the Public Key.
- Network Resilience: Through the Longest Valid Chain Consensus algorithm, the project shows that a distributed network can maintain a single version of the truth. Even if one node is compromised or goes offline, the remaining nodes utilize mathematical audits to preserve the integrity of the university schedule.

The chosen use case—a university course schedule—proved to be an effective and approachable example for illustrating these complex concepts. Since course schedules are time-sensitive and administratively critical, they serve as a perfect real-world scenario for showcasing why data integrity matters. Any unauthorized modification to lesson times or room assignments is instantly flagged by the Auditor Client, preventing potential disruption.

Ultimately, the project clearly shows how each update can be recorded, traced, and independently verified. By implementing features like Replay Protection and Canonical Hashing, the system meets and exceeds the intended learning outcomes, providing a robust and demonstrably secure framework for administrative data validation.

## 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.