toy-blockchain

Masoud Heidary
Couse: Cybersecurity ECE @ University of Houston
Apr 16, 2025

A simplified blockchain project inspired by Bitcoin, built for a Cybersecurity course at the University of Houston.

About the Project

This C++ implementation demonstrates the key elements of a blockchain: - **Proof-of-Work (PoW) - SHA-256 Hashing - Merkle Tree Root - Immutable chain structure - Fixed 10 transactions per block**

The goal is to showcase blockchain principles that support **data integrity**, **tamper resistance**, and **trustless verification**, all of which are critical in cybersecurity.

How to Build & Run

To compile and run the code, make sure OpenSSL development libraries are installed:

```
sudo apt install libssl-dev
g++ -o main main.cpp -lssl -lcrypto && ./main
```

Abstraction

This project demonstrates a simplified implementation of a blockchain system in C++, incorporating essential components such as transactions, blocks, Merkle trees, proof-of-work mining, and chain validation. The goal is to provide a hands-on understanding of blockchain data structures, cryptographic integrity, and consensus mechanisms without relying on external blockchain frameworks.

Each block in the chain contains multiple transactions and is mined based on a defined difficulty level requiring a hash with a specified number of leading zeros. The simulation validates the core principle of blockchain immutability: once a block is added to the chain, any modification to its contents invalidates the block and all subsequent blocks, unless they are re-mined — a computationally expensive process. By modifying a single transaction in an intermediate block, we highlight the cascading failure in hash linkage, Merkle root integrity, and overall chain validation.

This work serves as an educational tool to illustrate how blockchain maintains data integrity, resists tampering, and enforces trust in decentralized systems through cryptographic proofs.

Introduction

This project was developed as part of the Cybersecurity course at the University of Houston to explore the core cryptographic principles that underpin modern blockchain systems. The objective was to build a simplified, educational version of a blockchain in C++, inspired by the foundational ideas of Bitcoin.

The implementation focuses on three key areas critical to blockchain integrity and security:
- Cryptographic Hashing (SHA-256): Ensures data immutability and block linkage. - Merkle Trees: Efficiently represent and verify the integrity of transactions within a block. - Proof-of-Work (PoW): Simulates computational difficulty to secure new block creation.

Each block in the chain is constructed with a fixed number of ten transactions, mimicking real-world batching mechanisms, and contains the block's hash, previous block hash, and a Merkle root of its transactions. By mining blocks and linking them cryptographically, the project illustrates how any change to historical data renders the chain invalid — a core security feature of blockchain technology.

From a cybersecurity perspective, this project reinforces how blockchain resists tampering, supports data integrity, and builds distributed trust without relying on central authorities. While this is a simplified version, it successfully demonstrates the underlying mechanisms that make blockchain a powerful tool in secure, decentralized systems.

Blockchain Fundamentals (Background)

In this section, we explore the core building blocks of blockchain technology. These components form the cryptographic and structural foundation that makes blockchain secure, immutable, and relevant to the field of cybersecurity.

What is Blockchain?

A blockchain is a distributed, append-only data structure that stores information in a chain of blocks. Each block contains a batch of data (typically transactions), along with metadata that links it cryptographically to the previous block. This chain-like architecture ensures that any modification to an earlier block affects all subsequent blocks, making tampering computationally infeasible.

Blockchains are decentralized — no single party has control over the entire chain — and rely on consensus mechanisms to agree on the current state of data. This structure makes blockchains inherently resistant to data modification, which is why they are widely used in secure, trustless environments like cryptocurrencies, digital voting, and supply chain tracking.

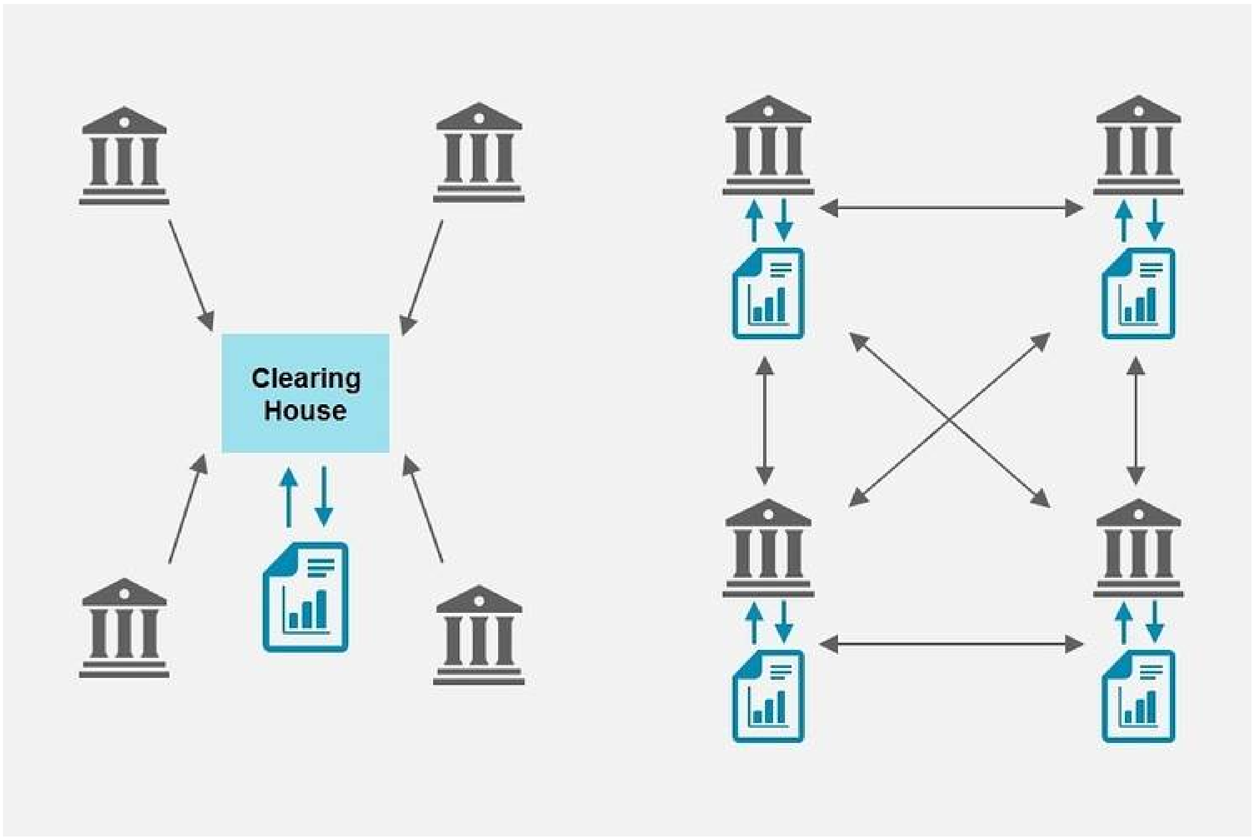

Figure 1: blockchain is decentralized

What Is a Merkle Root?

A Merkle root is a single hash that summarizes all transactions in a block by recursively hashing them in pairs until a single digest remains. This is done using a Merkle Tree, a binary tree of hashes where: - The leaves are the individual hashes of each transaction. - Each non-leaf node is the hash of its two child nodes. - The top-most node is the Merkle root.

Merkle roots allow efficient and secure verification of large data sets. In blockchain: - They allow verification of individual transactions without revealing the full data set. - They provide integrity guarantees: if even one transaction changes, the Merkle root changes.

This efficiency is particularly important in lightweight clients (e.g., SPV nodes in Bitcoin) that do not download the entire blockchain but still want to verify transaction inclusion.

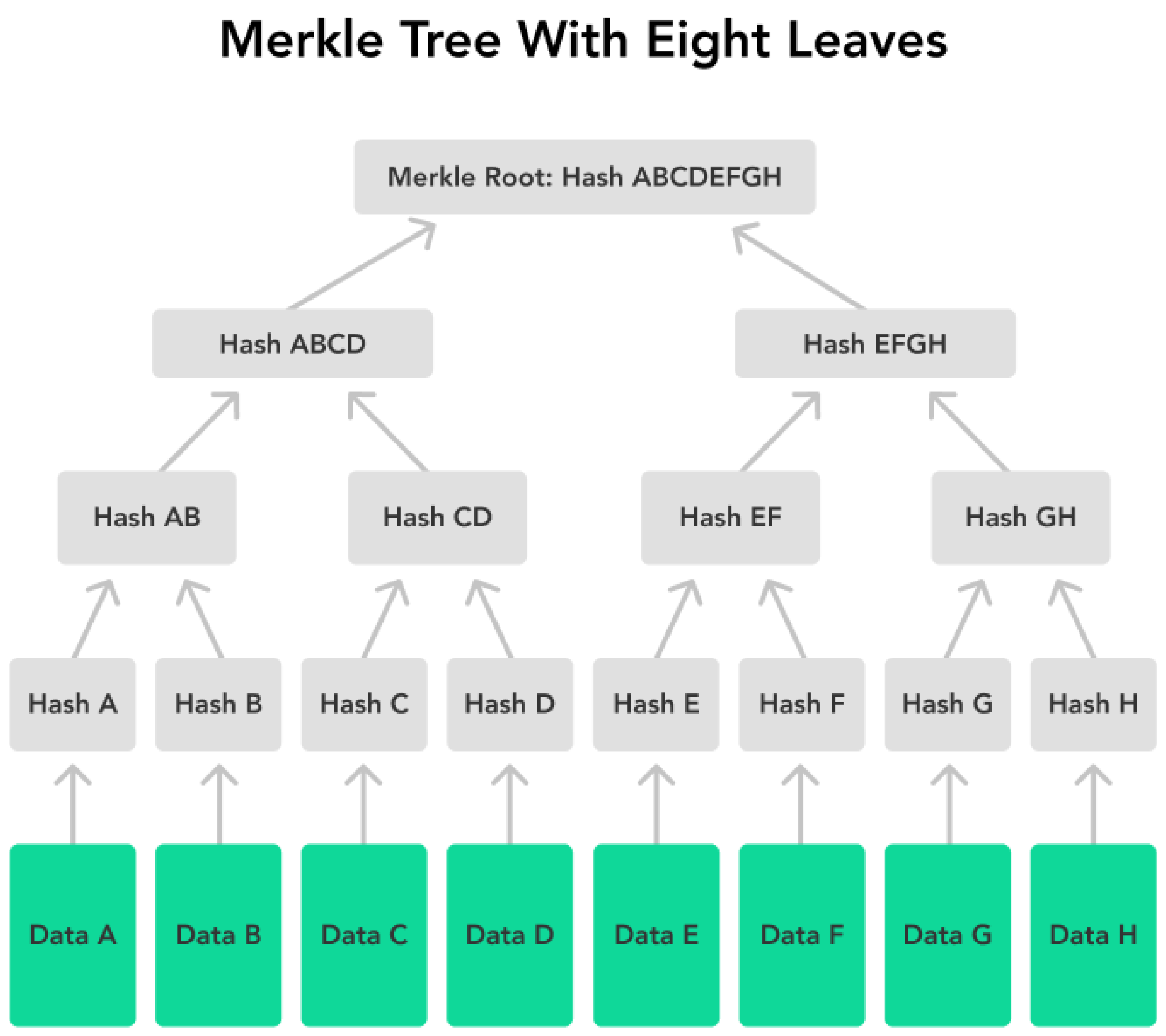

Figure 2: merkle root with eight leaves

Hashing & SHA-256

Hashing is the process of converting arbitrary data into a fixed-size string of characters using a mathematical function called a hash function. Hashes are:

- Deterministic: the same input always gives the same output.
- Non-reversible: you cannot recover the input from the output.

- Collision-resistant: it's hard to find two different inputs with the same output.

In this project, we use SHA-256, a member of the Secure Hash Algorithm 2 family, producing a 256-bit (32-byte) hash. SHA-256 is:

- Widely used in blockchain, especially Bitcoin.
- Considered secure (as of today) against known pre-image and collision attacks.
- Critical for linking blocks, hashing transactions, and computing Merkle roots.

Each block's identity is based on its SHA-256 hash, which includes key metadata like the previous hash, Merkle root, timestamp, and nonce. Even a one-bit change in the block's data causes the resulting hash to change drastically — this is known as the avalanche effect.

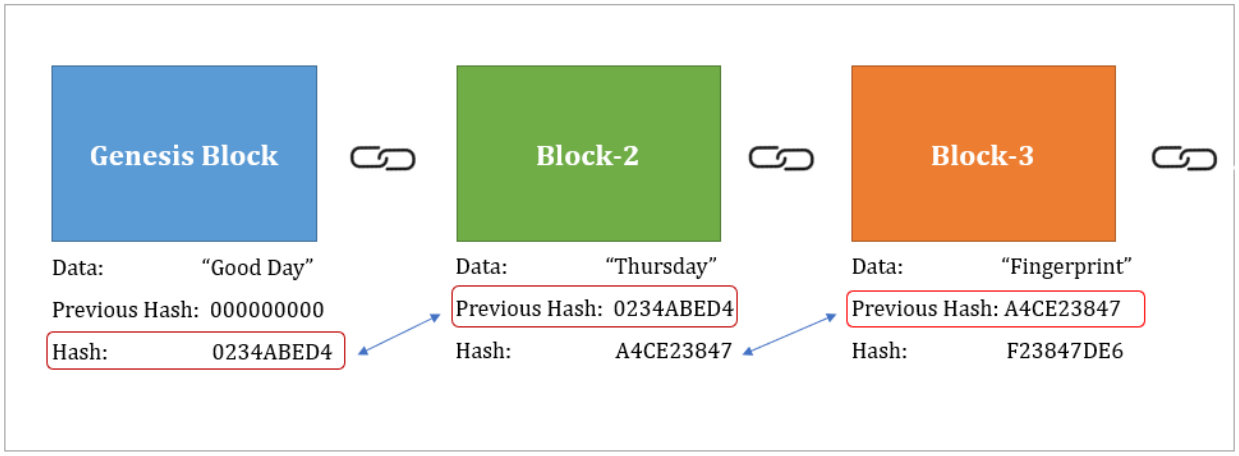

Figure 3: Hash importance in Block Chain

Proof-of-Work (PoW)

Proof-of-Work is a consensus algorithm that requires participants (called miners) to perform a certain amount of computational work to add a new block to the chain. This "work" typically involves finding a nonce value such that the resulting hash of the block's header starts with a certain number of leading zero bits (difficulty).

This ensures: - Blocks cannot be added instantly; time and energy are required. - Altering a block would require re-mining it and all subsequent blocks. - The system is resistant to Sybil attacks, where malicious actors try to flood the network with fake identities.

In our project, PoW is implemented by brute-forcing the nonce value until the SHA-256 hash of the block starts with N zeros (e.g., `0000 ...`), simulating real-world mining difficulty.

Importance of Immutability in Cybersecurity

In cybersecurity, immutability — the guarantee that once data is written it cannot be altered — is a highly valued property. Blockchain achieves immutability through a combination of cryptographic hashing, Merkle trees, and consensus algorithms like PoW.

This immutability offers several security advantages:

- Tamper detection: Any unauthorized change in historical data breaks the chain.
- Auditability: All actions and changes are permanently recorded and traceable.
- Trustless environments: Participants do not need to trust each other; trust is placed in the protocol.

In the context of this project, the blockchain serves as a miniature secure ledger where transaction history cannot be forged without redoing significant computational work, aligning with the core principles of confidentiality, integrity,

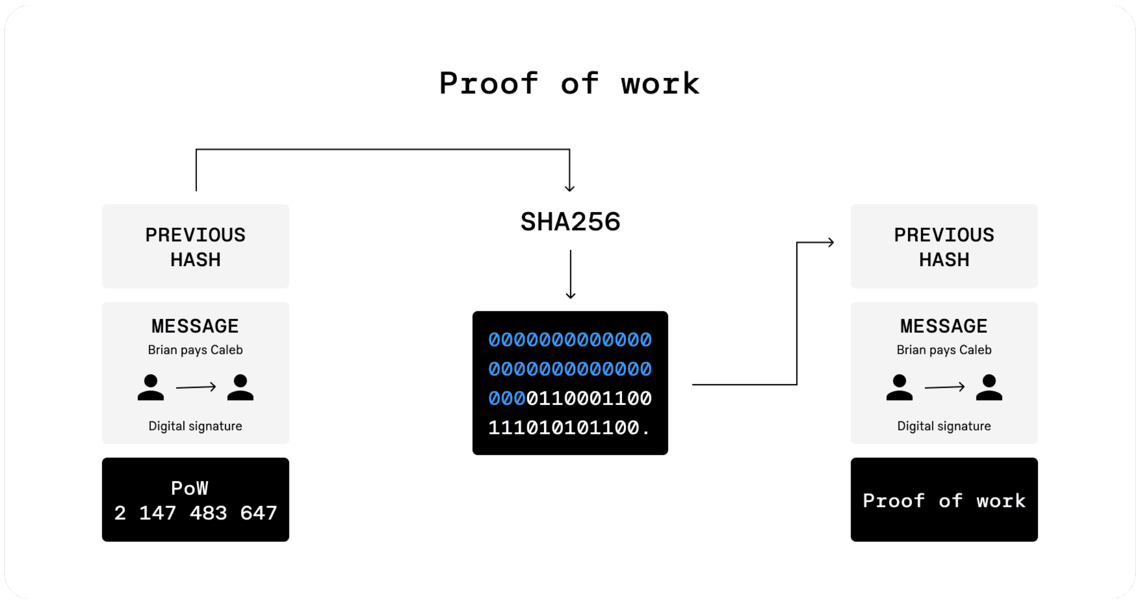

Figure 4: proof of work

and availability (CIA) in cybersecurity.

Implementation Details: Simple Blockchain in C++

This section details the core implementation of the blockchain project developed for the Cybersecurity course at the University of Houston. The goal was to implement a Bitcoin-like blockchain with fundamental features such as transactions, mining, block validation, Merkle roots, and immutability enforcement.

Data Structure

Transaction

Each transaction contains a sender, a receiver, and a transfer amount:

```cpp
struct Transaction {
    std::string sender;
    std::string receiver;
    double amount;
};
```

Block

A block contains a list of transactions, timestamp, previous block's hash, Merkle root, nonce, and current block hash:

```cpp
class Block {
    int index;
    time_t timestamp;
    std::vector<Transaction> transactions;
    std::string last_hash;
    std::string markle_root;
    std::string block_hash;
    long nonce;
};
```

Blockchain

A chain of validated blocks:

```cpp
class BlockChain {
    std::vector<Block> blocks;
};
```

Block Construction

Each block contains exactly 10 transactions.

The block must be mined such that the hash starts with `n` leading zeroes, determined by `MINE_DIFFICULTY`.

Transactions are randomly generated from a pool of names and amounts:

```cpp
Transaction get_random_transaction(const std::vector<std::string>& names, int max_amount);
```

Proof-of-Work Mechanism

Blocks are mined using a simple brute-force method to satisfy difficulty:

```cpp
double mine_block() {
    do {
        nonce++;
        block_hash = get_block_hash();
    } while (!is_hash_lead());
}
```

`is_hash_lead()` checks if the hash starts with the correct number of leading zeros.

Blockchain Validation

The `is_valid()` method in Block ensures:

- Merkle root matches transactions.
- Hash is correctly calculated.
- Hash starts with required leading zeros:

```cpp
bool is_valid() const {
    if (markle_root != get_markle_root())
        return false;
    if (block_hash != get_block_hash())
        return false;
    if (!is_hash_lead())
        return false;
    return true;
}
```

The blockchain is also validated as a whole:

```cpp
bool BlockChain::is_valid() const {
    for ( ... ) {
        if (!blocks[i].is_valid() || blocks[i].last_hash != blocks[i - 1].block_hash)
```

```
            return false;
        }
    return true;
}
```

Result & Evaluation

We conducted a simulation of a simple blockchain network by creating and mining a chain of 10 blocks, each containing 10 transactions. The mining difficulty was set such that a valid block hash required a prefix of five leading zeros. The following key points summarize the behavior and performance:

Transaction Generation:

You can also manually create a specific transaction by directly providing the sender, receiver, and amount:

```
Transaction t {"Alice", "Bob", 5};
t.print();
```

this will output:

```
Alice → Bob: 5
```

One Random Transaction

Random transactions between predefined users (e.g., Alice, Bob, Eve, etc.) are generated for each block. Each transaction includes:

- A random sender
- A random receiver
- A random amount (up to a defined max_tx_amount)

This is handled using a helper function `get_random_transaction( … )`, which randomly selects sender/receiver names from a list and assigns a random amount.

Example of generating one random transaction:

```
const int max_tx_amount = 10;
std::vector<std::string> names = {
    "Alice", "Bob", "Charlie", "Dave", "Eve",
    "Frank", "Grace", "Heidi", "Ivan", "Judy"
};

Transaction rt = get_random_transaction(names, max_tx_amount);
rt.print();
```

Sample output:

```
Eve → Grace: 9.73
```

List of Random Transactions for one Block

Example of generating a list of Transactions for one Block:

```
std::vector<Transaction> tx_list = random_tx_lst(names, max_tx_amount, TRANSACTION_BLOCK_SIZE);

for (const auto &tx: tx_list) {
    tx.print();
}
```

Sample output:

```
trasaction [0]: Eve → Grace: 9.73
trasaction [1]: Bob → Grace: 4.84
trasaction [2]: Eve → Heidi: 7.2
trasaction [3]: Frank → Ivan: 5.2
trasaction [4]: Bob → Grace: 8.22
trasaction [5]: Eve → Alice: 9.82
trasaction [6]: Dave → Eve: 8.13
trasaction [7]: Bob → Ivan: 5.27
trasaction [8]: Dave → Alice: 2.81
trasaction [9]: Bob → Grace: 7.8
```

Each of these is randomly created during the block formation phase, simulating real-world transaction traffic in a toy blockchain environment.

Mining Time:

Once the genesis (first) block is mined, additional blocks can be mined and appended to the chain. Each block contains a new list of randomly generated transactions, a reference to the hash of the previous block, and a newly computed valid hash that satisfies the mining difficulty.

The process follows these steps: 1. Generate a list of random transactions. 2. Construct a new block, using: - The current index. - The transaction list. - The previous block's hash (to preserve the chain). 3. Mine the block: this means finding a valid hash that starts with a specified number of zeros (according to the difficulty). 4. Add the block to the chain.

```

std::cout << "adding 9 more Blocks to the BlockChain" << std::endl;

for (int index = 1; index < 10; index++) {
    std::vector<Transaction> tx_list;

    Block block {
        index,
        random_tx_lst(names, max_tx_amount, TRANSACTION_BLOCK_SIZE),
        chain.get_last_hash(), // previous hash for linking
    };

    double exetime = block.mine_block();  // perform Proof of Work
    chain.add_block(block);               // append to blockchain

    std::cout << " - Block [" << index << "] mined in " << exetime << "s" << std::endl;
}
coutln();
```

Code Example

- `random_tx_lst( ... )`: creates a list of `TRANSACTION_BLOCK_SIZE` random transactions.
- `chain.get_last_hash()`: fetches the hash of the last block to maintain integrity.
- `mine_block()`: attempts different nonces until a valid hash is found (based on the mining difficulty).
- `chain.add_block( ... )`: safely appends the mined block to the blockchain.

output for `#define MINE_DIFFICULTY 5`:

```
- Block [1] mined in 1s
- Block [2] mined in 5s
- Block [3] mined in 11s
- Block [4] mined in 3s
- Block [5] mined in 4s
- ...
```

output for `#define MINE_DIFFICULTY 6`:

```
- Block [1] mined in 120s
- Block [2] mined in 40s
- Block [3] mined in 18s
- Block [4] mined in 2s
- Block [5] mined in 11s
- ...
```

output for `#define MINE_DIFFICULTY 7`:

```
- Block [1] mined in 580s
- Block [2] mined in 308s
- Block [3] mined in 719s
- ...
```

Chain Sequence

Each block in the blockchain is cryptographically linked to the one before it using its hash. This creates a secure and verifiable sequence of data—any change in a previous block will invalidate the hashes of all subsequent blocks.

The sequence of hashes looks like this:

```
hain Hash sequence:
 - 0000000000000000000000000000000000000000000000000000000000000000 → 00000bb31d28042367d1a3d9b2b662bcc

 - 00000bb31d28042367d1a3d9b2b662bcc9c3bd0fcdcfa0dc05dc8d6a7cf13ca4 → 000005fc7d4b66716742cebb74f68f9fb

 - 000005fc7d4b66716742cebb74f68f9fbad45cf4d7059dd7eaed4a224dd63b40 → 00000ad2da75e0573f59fe3fd42d5a5cb
 -  ...
```

This chain of hashes ensures immutability:

- If any transaction in a block changes, the Merkle Root will change.
- That causes the block's hash to change.
- As a result, all following blocks will have incorrect previous_hash values, breaking the chain.

This is why blockchains are considered secure and tamper-evident.

Blockchain Manipulation Attempt

This section demonstrates why tampering with a single block in the blockchain invalidates the entire chain and is computationally infeasible to fix without significant effort.

Step 1: Replace a Transaction in Block[5]

```
chain.blocks[5].transactions[0] = get_random_transaction(names, max_tx_amount);
```

Result:

```
 - Block [5] Hash is an invalid Hash (as the content is altered)
 - Block [5] invalid MerkleRoot (as Transactions are altered)
```

- **Why it fails:** Each block contains a hash that depends on its content (including transactions).
- By changing a transaction, the content changes, making the stored block hash **no longer valid**.
- Also, the `Merkle Root`, which is a hash summary of all transactions, becomes invalid.

Step 2: Recalculate Merkle Root

```
chain.blocks[5].markle_root = chain.blocks[5].get_markle_root();
```

Result:

```
 - Block [5] Hash is an invalid Hash (as the content is altered)
```

- **Why it still fails:** Even though the Merkle Root is corrected, the overall `block hash` is still outdated and incorrect.
- The block's hash was originally mined to satisfy the difficulty (e.g., having 5 leading zeros), and recalculating the Merkle Root doesn't solve this.

Step 3: Remine Block[5]

```
chain.blocks[5].mine_block();
```

Result:

```
 - Block [6] is not following the Block[5] Hash
```

- **Why it fails now:** Even after re-mining `Block[5]` to get a valid hash, `Block[6]` still stores the old hash of `Block[5]` as its previous_hash.
- So `Block[6]` becomes invalid because it no longer links correctly to the new hash of `Block[5]`.

Final Consequence

- **Explanation:** You would have to re-mine every block after `Block[5]` to fix the entire chain.
- Given the mining difficulty and computational cost, this becomes practically impossible—this is exactly what gives blockchain its immutability and security.

references

- Wall Street Journal. (n.d.). CIO Explainer: What Is Blockchain? Retrieved from https://www.wsj.com/articles/BL-CIOB-8993
- River. (n.d.). Merkle Tree. Retrieved from https://river.com/learn/terms/m/merkle-tree/
- Metlabs. (n.d.). What is a Hash in Blockchain? Basic Concepts and Characteristics. Retrieved from https://metlabs.io/en/what-is-a-hash-in-blockchain/
- CoinLoan. (n.d.). What is Proof of Work (PoW)? Retrieved from https://coinloan.io/blog/what-is-proof-of-work-pow/
- Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved from https://bitcoin.org/bitcoin.pdf