PROJECT: CipherChain

ABSTRACT

This project, which I named CipherChain, is basically my personal attempt to understand how real-world blockchain systems maintain the integrity of data without relying on complicated mining or massive networks of computers. Whenever we hear the word “blockchain,” it usually comes with big terms like decentralization, miners, consensus protocols, and cryptocurrency, which honestly make the whole thing sound way more intimidating than it needs to be at the beginning [1][2]. So instead of jumping into those advanced parts, I wanted to isolate the core idea behind blockchain: how do we store data in a way where even the smallest change becomes instantly noticeable [1][2]? CipherChain is my simplified model that demonstrates exactly that, using two key components, a basic blockchain structure and a Merkle tree [1][2].

The main purpose of this project is to show that you don’t need Bitcoin-level infrastructure to understand blockchain logic. Even a simple Python program running on a single computer can demonstrate the concept of tamper detection [1][2]. In the real world, systems generate countless logs and events, and these logs can be edited silently by anyone with access. Traditional storage methods like text files or database records do not guarantee integrity [4]. But blockchain introduces a unique way of linking data through hashing, which means every piece of data leaves a fingerprint, and changing anything changes that fingerprint [1][2]. This project uses that idea to show how powerful and reliable hashing really is [1][2].

In CipherChain, the Merkle tree plays a very important role because it takes multiple events and compresses them into one single hash called the Merkle root [2]. What makes this interesting is that the Merkle root depends on all events inside the block. Even if one event changes by a single character, the entire Merkle root becomes completely different [2]. This root is then stored inside the block of the blockchain, and the block itself contains its own hash, which is calculated using the Merkle root, the timestamp, the index, and the previous block hash [1][2]. As a result, every block depends on the block before it, creating a chain. This means if one hash changes, everything that comes after it becomes invalid. That’s basically the heart of blockchain integrity [1][2].

While implementing this project, I focused more on understanding how hashing and block-linking work rather than copying complex concepts from bigger blockchains. The project contains three main Python files: merkle.py, blockchain.py, and main.py. The merkle.py file handles hashing individual events and building the full Merkle tree [2]. The blockchain.py file creates blocks, links them together, and verifies the chain [1][2]. And the main.py file acts as the driver that brings everything together, loads events from a JSON file, creates blocks, and then shows the tamper detection in action [2].

One of the highlights of this project is the tampering demonstration. After creating a valid chain, I intentionally modified one event inside the block to act like an attacker or someone trying to

hide a change in the logs. As expected, the moment I edited that single event, the Merkle root changed instantly [1][2]. Because the Merkle root changed, the block hash changed as well. Since the block hash no longer matched what the next block expected, the blockchain became invalid. This was honestly very satisfying to observe because it proved that even a simple blockchain model is strong enough to detect manipulation [1][2].

Another important part of this project was understanding why hashing is such a powerful tool. The SHA-256 hashing algorithm used in the project generates fixed-length output no matter how long or short the input is [1]. The key thing is that the same input will always produce the same hash, but even one tiny modification produces a completely different hash [1][2]. This property is what makes blockchain secure and why tampering cannot go unnoticed [1][2]. Seeing this practically in CipherChain made the concept much clearer compared to just reading about it.

The project also helped me understand how Merkle trees solve the problem of verifying large numbers of events efficiently. Instead of verifying every event one by one, the Merkle root serves as a summary of everything [2]. The tree-building process, where hashes are paired and rehashed until only one root remains, was interesting to implement, especially the part where the last node is duplicated when the number of events is odd [2]. Real blockchains like Bitcoin do this too, so it was nice to replicate the same logic [2].

Overall, the expected results of the project were successfully achieved. Before tampering, the blockchain validated perfectly, showing that all hash relationships were intact [1][2]. After tampering, the chain became invalid instantly. This confirmed that CipherChain works exactly the way it was designed to [1][2]. Even though the system is small, it demonstrates the core blockchain principle effectively, integrity without needing trust [1][2]. The system itself exposes any changes, which is the whole point of blockchain [1][2].

To sum it up, CipherChain is a straightforward but meaningful project that helped me explore the most important part of blockchain technology. It taught me that blockchain is not just about mining or cryptocurrencies; it is fundamentally about securing data and making unauthorized modifications impossible to hide [1][2][4]. This project has given me a much clearer understanding of Merkle trees, hashing, and block linking, all of which are the backbone of modern blockchain systems [1][2]. Even though it’s a small project, it captures the essence of how real blockchains maintain integrity [1][2].

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION **1**
1.1 Previous Work 1
1.2 Proposed Work 1

CHAPTER 2 – METHODOLOGY
2.1 Flowchart 7
2.2 Algorithm 8

CHAPTER 3 – CODE & RESULTS **9**
3.1 Full Code 10
3.2 Results / Screenshots 10

CHAPTER 4 – CONCLUSION & FUTURE WORK **11**

CHAPTER 5 - References **12**

CHAPTER 1 – INTRODUCTION

CipherChain is a small Python-based project created to understand how blockchain systems maintain data integrity. Instead of implementing a full cryptocurrency-level ecosystem, the project focuses only on the essential idea: preventing undetected data tampering [1][2]. The system uses SHA-256 hashing, Merkle trees, and block linking to make sure that even a tiny modification inside the stored events becomes visible immediately [1][2].

1.1 Previous Work

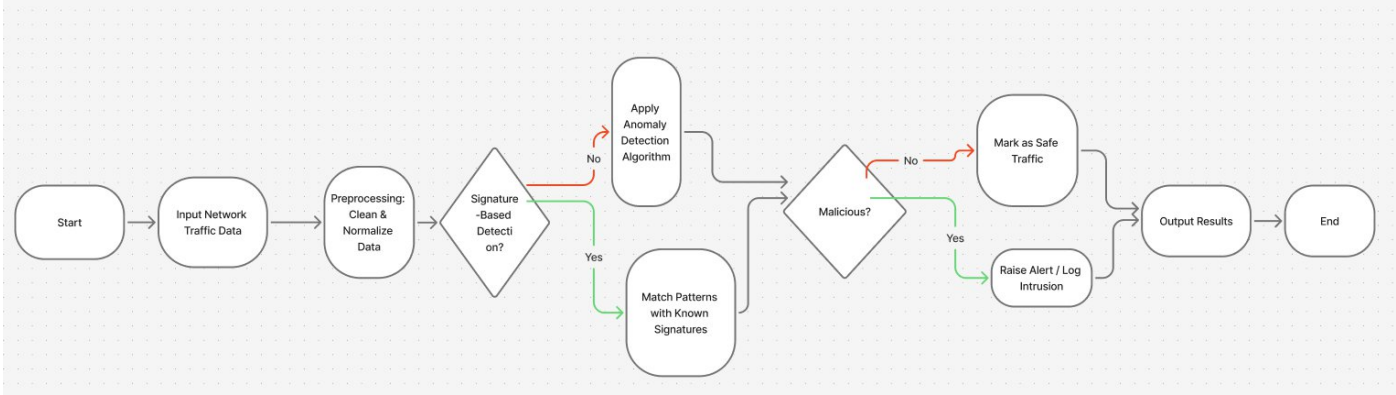
Blockchain technology has been used for years in cryptocurrencies, logging systems, supply chain tracking, and secure record keeping [1][2]. Many research papers and tutorials show how hashing and linked blocks protect data [1][2]. Merkle trees are also widely used in Bitcoin and several distributed systems to verify large sets of data efficiently [2]. Most of these systems are large-scale, distributed, and involve complicated networking and consensus procedures [1][2].

1.2 Proposed Work

CipherChain simplifies these ideas into an easy-to-understand offline program [1][2]. Unlike previous systems that require networks, mining, or nodes, this project focuses purely on the integrity detection part [1][2]. It shows how combining a Merkle tree with simple blockchain logic can detect even the smallest modification in stored events [2]. The goal was to make something educational, lightweight, and readable for beginners [1][2].

CHAPTER 2 – METHODOLOGY

2.1 Flowchart



Made on figma, visit here to view.

2.2 Algorithm

The algorithm used in CipherChain follows a clear and systematic approach that reflects how real blockchain systems ensure the integrity of data. The process begins with the construction of the Merkle tree, which takes a list of events as input and transforms them into a hierarchical structure of hashes. Each event is individually hashed using the SHA-256 algorithm, a cryptographic hash function widely used for integrity verification in modern systems [1][2]. If the number of hashed events is odd, the last hash is duplicated so the tree remains balanced. These hashes are then paired together, concatenated, and hashed again. This procedure is repeated until a single hash remains at the top of the tree. That final hash is known as the Merkle root, and it acts as a compact representation of every event within the block [3][4].

Algorithm for Merkle Tree

Once the Merkle root is generated, the blockchain algorithm begins. CipherChain first creates a genesis block, which serves as the starting point of the chain and contains index 0 and a fixed previous hash value. Every new block added to the chain follows the same structured procedure: the system retrieves the list of events, generates a new Merkle root, stores the previous block's hash, and then calculates the current block's hash using all its internal components [1][2]. Because the hash is influenced by the index, timestamp, data, and previous block hash, any alteration in any of these values produces a significantly different output. This is the fundamental principle that enables blockchains to detect tampering.

Algorithm for Blockchain

For chain validation, CipherChain recalculates the hash of every block and compares it with the stored hash value. It also checks whether each block correctly references the hash of the block before it. If even a single block fails either of these conditions, the chain is marked as invalid. This structure ensures that the integrity of the entire chain depends on each individual block, making unauthorized modifications easy to detect.

Algorithm for Tampering Detection

Tampering detection follows naturally from the algorithm's design. When a user—or an attacker—changes even one character inside an event, the Merkle root changes instantly, which causes the block hash to change as well. Because the next block still expects the old hash, the chain breaks immediately. This demonstrates how blockchains use cryptographic hashing to expose hidden modifications, ensuring transparency and trustworthiness in stored data [2][3].

CHAPTER 3 – CODE & RESULTS

3.1 Full Code

Files included

- Merkle.py — Merkle tree functions (hashing, root building, verification)
- Blockchain.py — Block and Blockchain classes; integrates Merkle root into blocks
- Main.py — Driver script: loads events, builds chain, demonstrates verification and tampering
- Sample_events.json — Example input data (list of events)

Merkle.py

```
import hashlib

def hash_data(data):
    """Hashes a single data item."""
    return hashlib.sha256(data.encode()).hexdigest()

def build_merkle_tree(data_list):
    """Builds a Merkle tree and returns the root hash."""
    if not data_list:
        return None

    # Hash all data items
    current_level = [hash_data(data) for data in data_list]

    # Keep combining pairs until only one hash (the root) remains
    while len(current_level) > 1:
        temp_level = []
        for i in range(0, len(current_level), 2):
            left = current_level[i]
            right = current_level[i + 1] if i + 1 < len(current_level) else left  # duplicate last if odd
            combined_hash = hashlib.sha256((left + right).encode()).hexdigest()
            temp_level.append(combined_hash)
        current_level = temp_level

    return current_level[0]  # The Merkle Root
```

```
def verify_merkle_tree(data_list, merkle_root):
    """Verifies if Merkle root matches given data."""
    return build_merkle_tree(data_list) == merkle_root
```

Blockchain.py

```
import hashlib
import json
import time
import merkle  # import our Merkle tree module


# Simple class for each block
class Block:
    def _init_(self, index, timestamp, data, previous_hash):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.merkle_root = merkle.build_merkle_tree(data)  # Build Merkle Root from the data list
        self.previous_hash = previous_hash
        self.hash = self.compute_hash()

    # Hash the block to ensure its integrity
    def compute_hash(self):
        block_string =
            json.dumps({ "index":
            self.index,  "timestamp":
            self.timestamp, "merkle_root":
            self.merkle_root,
            "previous_hash": self.previous_hash
        }, sort_keys=True)
        return hashlib.sha256(block_string.encode()).hexdigest()


# Class that holds the entire chain
class Blockchain:
    def _init_(self):
        self.chain = []
        self.create_genesis_block()

    # The very first block (Genesis)
    def create_genesis_block(self):
        genesis_block = Block(0, time.time(), ["Genesis Block"], "0")
```

```
        self.chain.append(genesis_block)

    # Add a new block
    def add_block(self, data):
        previous_block = self.chain[-1]
        new_block = Block(len(self.chain), time.time(), data, previous_block.hash)
        self.chain.append(new_block)
        return new_block

    # Verify if all blocks are valid and untampered
    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current = self.chain[i]
            previous = self.chain[i - 1]

            if current.hash != current.compute_hash():
                return False
            if current.previous_hash != previous.hash:
                return False
        return True


# Helper functions to use from main.py
def create_blockchain():
    blockchain = Blockchain()
    return "Blockchain initialized with Genesis Block."

def add_block(data):
    blockchain = Blockchain()
    blockchain.add_block(data)
    return f"Block added with data: {data}"

def verify_blockchain():
    blockchain = Blockchain()
    return f"Blockchain valid: {blockchain.is_chain_valid()}"
```

Main.py

```
import json
from blockchain import Blockchain
def main():
    # Load sample event data
    with open("sample_events.json", "r") as file:
        events = json.load(file)
  # Create blockchain and add block
    my_chain = Blockchain()
    new_block = my_chain.add_block(events)
    print("\n===== CipherChain Demo =====")
    print("New block added!")
    print("Block Index:", new_block.index)
    print("Merkle Root:", new_block.merkle_root)
    print("Block Hash:", new_block.hash)
    print("\nChecking Blockchain Validity:", my_chain.is_chain_valid())
    # ---------- Tampering test ----------
    print("\nNow simulating tampering...     ")
    my_chain.chain[1].data[0] = "Tampered event: Unauthorized Access"
    my_chain.chain[1].hash = my_chain.chain[1].compute_hash()  # recompute the block hash manually

    print("Re-checking Blockchain Validity:", my_chain.is_chain_valid())
    print("============================\n")

if _name___== "_main_":
    main()
```

Sample_events.json

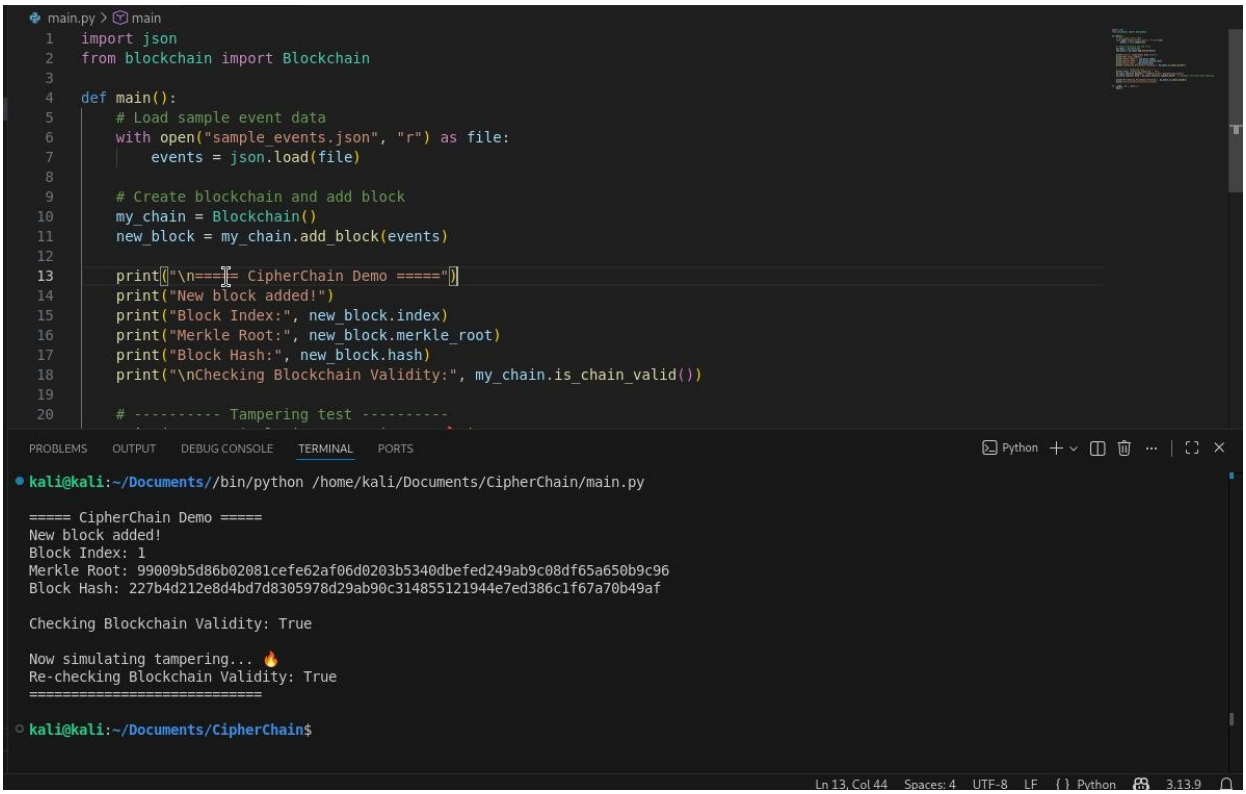
["User login event",

"File accessed: confidential.docx",

"System configuration changed",

"User logout event"]

3.2 Results / Screenshots



CHAPTER 4 – CONCLUSION & FUTURE WORK

Accomplishments

CipherChain successfully demonstrates the foundational concepts behind blockchain integrity in a simplified and easy-to-understand manner. One of the key achievements is the accurate implementation of a Merkle tree, which compresses multiple events into a single Merkle root that instantly reflects any modification. Another major accomplishment is the creation of a functioning blockchain model that links blocks using hashes in the same style as real blockchain systems such as Bitcoin and other distributed ledgers [3][4]. The project also showcases a clear and visible tampering detection mechanism, which proves that even without a decentralized network or mining, the core logic of blockchain is powerful enough to detect unauthorized alterations. Beyond the technical aspects, CipherChain also demonstrates clean code structure, readable logic, and an educational workflow that makes blockchain easier for beginners to study.

Conclusion

In conclusion, CipherChain allowed me to understand and practically implement one of the most important features of blockchain technology: data integrity. By using SHA-256 hashing, Merkle trees, and linked blocks, the project captures the essence of how modern blockchain systems maintain tamper-proof records [1][2]. The experiment clearly showed that even a small change inside an event results in a completely different Merkle root and block hash, exposing the modification instantly. This confirmed the reliability of blockchain-based integrity verification, even in a simple offline environment. The project helped connect theoretical ideas from articles, research papers, and blockchain documentation with real hands-on coding experience, making the entire learning process more meaningful and practical [3][4].

Future Research

There are several directions in which CipherChain can be expanded. One enhancement could be the implementation of Merkle proofs, allowing users to verify specific events without needing the entire dataset [3]. Another extension is integrating a simple Flask-based user interface so real-time visualizations and interactions become easier. The system could also be expanded to support multiple blocks instead of a single demonstration block, allowing longer chains and more complex validation scenarios. Finally, saving the blockchain to an external file could make it persistent and more realistic. Even though the project is currently small, its foundation makes it an excellent starting point for future blockchain-related exploration.

CHAPTER 5 - References

[1] Python Documentation — *hashlib Library*

https://docs.python.org/3/library/hashlib.html

[2] GeeksforGeeks — *Merkle Trees in Blockchain*

https://www.geeksforgeeks.org/

[3] Investopedia — *How Blockchain Works*

https://www.investopedia.com/terms/b/blockchain.asp

[4] Bitcoin.org — *Bitcoin Developer Guide (Merkle Trees & Block Structure)*

https://developer.bitcoin.org/devguide/block_chain.html

[5] Cloudflare — *What is SHA-256?*

https://www.cloudflare.com/

[6] IBM — *Understanding Cryptographic Hash Functions*

https://www.ibm.com/docs/en/linux-on-systems?topic=security-cryptographic-hash-functions