Implementation and Security Analysis of a PoW-Based Cryptocurrency Protocol

Sehong Park
University of miami

December 9, 2025

Abstract

This report provides a comprehensive analysis of cryptocurrency systems, ranging from the historical evolution of digital cash to a technical audit of a proprietary Python-based blockchain implementation. We examine the theoretical foundations of the Proof of Work (PoW) consensus mechanism, analyzing its role in achieving distributed consensus and preventing double-spending. Furthermore, this document details the architectural design of a custom blockchain protocol, highlighting its account-based model, Merkle tree implementation, and Ed25519 cryptography. A portion of this report is dedicated to a security audit, verifying implemented countermeasures against Denial of Service (DoS) and Replay Attacks through simulation. Additionally, we demonstrate the system's stability via a dynamic difficulty adjustment mechanism and analyze its resilience against chain reorganization scenarios.

Contents

1 Introduction

1.1 Background and Motivation

The inception of Bitcoin was driven by the inherent limitations of the trust-based model underpinning traditional financial institutions. The current fiat system relies heavily on centralized intermediaries, such as banks, to mediate transactions and validate funds. This centralization introduces not only a single point of failure but also a problematic concentration of authority; intermediaries possess the power to censor transactions, freeze assets, or impose arbitrary restrictions, effectively limiting individuals' financial sovereignty. Furthermore, if these institutions lose public trust or fail to maintain their integrity due to insolvency, cyber-attacks, or geopolitical instability, the utility of the currency is jeopardized. Consequently, there was a growing demand for a decentralized digital currency that functions independently of any central authority.

However, creating such a currency posed a significant technical challenge known as the "double-spending problem." In his seminal paper, Satoshi Nakamoto [3] resolved this issue by combining cryptographic proof with a peer-to-peer network, enabling a robust, censorship-resistant digital currency that has since gained widespread adoption.

1.2 Scope of Study

Building upon the motivations outlined above, this report conducts a **technical study and implementation analysis** of a blockchain prototype. This project focuses on simulating the core components of a cryptocurrency to understand their architectural interplay in a simplified environment. Specifically, we constructed a Python-based toy model to examine the following pillars:

- **Consensus & Minting**: Simulating how distributed nodes agree on the ledger state via **Proof-of-Work (PoW)** and adjusting mining difficulty dynamically to maintain block time stability.
- **Identity Management**: Implementing user custody using **Ed25519** (Edwards-curve Digital Signature Algorithm). This scheme was chosen for the prototype to demonstrate modern alternatives to the traditional ECDSA used in earlier systems.
- **State Transition**: Utilizing an **Account-based Model** (similar to Ethereum [4]) rather than the UTXO model, to simplify state management and balance lookups in a simulation setting.
- **Economic Constraints**: Testing a custom logic for **Adaptive Precision**, which restricts transaction decimal places based on mining difficulty, and implementing a basic **Fee Market** where transactions are prioritized by fee density.
- **Data Integrity**: Recording transaction history using Merkle Trees and ensuring temporal validity to prevent time-based attacks.

A key architectural distinction of this study is its **hybrid design**: while it adopts the Proof-of-Work consensus mechanism identical to Bitcoin [3], it employs an Account-based Model for ledger updates. This combination allows for a focused analysis of state transitions within a probabilistic consensus framework.

2 Theoretical Background

This section outlines the cryptographic primitives and architectural concepts that form the foundation of the implemented protocol.

2.1 Cryptographic Primitives

The security of the system relies on robust, standard cryptographic algorithms. We utilize **SHA-256** and **RIPEMD-160** for hashing operations to ensure collision resistance and data integrity, adhering to the standards set by established cryptocurrencies.

For digital signatures, our implementation diverges from traditional systems (which often use RSA or secp256k1-based ECDSA) by employing the **Ed25519** signature scheme. This is a variant of the Edwards-curve Digital Signature Algorithm (EdDSA) utilizing Twisted Edwards curves.

- **Efficiency & Security:** Ed25519 offers high-speed verification and signing operations while mitigating common side-channel attacks associated with standard ECDSA implementations [5].
- **Key Size:** It utilizes compact 32-byte public keys compared to the significantly larger keys required by RSA. This reduction is critical for minimizing blockchain storage requirements and facilitating decentralization.

Furthermore, to enhance user experience and prevent transcription errors, addresses are encoded using **Base58Check** encoding, which incorporates a checksum derived from double-SHA256 hashing.

2.2 Blockchain Architecture and Data Structure

The system architecture adopts an **Account-based Model**, similar to Ethereum [4], rather than the Unspent Transaction Output (UTXO) model used by Bitcoin.

- **State Management:** The global state consists of a direct mapping of addresses to account balances and nonces. This design simplifies the implementation of balance lookups and facilitates more complex logic compared to the stateless nature of UTXO.
- **Merkle Tree:** To ensure data integrity within blocks, transactions are organized into a Merkle Tree. The **Merkle Root** is included in the block header, allowing lightweight clients to verify the inclusion of specific transactions without downloading the entire block body [3].

2.3 Consensus Mechanism: Proof-of-Work

To resolve the Byzantine Generals Problem and establish a unified history without a central authority, the system adopts a **Proof-of-Work (PoW)** consensus mechanism.

1. **Mining Process:** Miners must find a probabilistic solution, known as a *nonce*, such that the hash of the block header satisfies a specific difficulty target:

$$H(Block\_Header||Nonce) < Target \tag{1}$$

   where H is the SHA-256 cryptographic hash function. The target value is inversely proportional to the mining difficulty. Finding such a nonce is computationally expensive and requires brute-force trial and error, ensuring that block creation incurs a tangible cost.

2. **Longest Chain Rule:** In the event of a network partition or propagation latency (a fork), nodes implicitly agree to adopt the chain with the greatest cumulative computational work, commonly referred to as the *Longest Chain*. This rule provides probabilistic finality; as long as the majority of CPU power is controlled by honest nodes, rewriting the transaction history becomes computationally infeasible [3].

2.4 The Double-Spending Problem

A fundamental challenge in digital currency systems is the *double-spending problem*, where a single digital token is spent more than once. Unlike physical cash, digital data can be effortlessly duplicated.

In traditional centralized financial systems, a trusted intermediary (e.g., a bank) prevents this by maintaining a central ledger and validating each transaction serially. However, in a decentralized network lacking such an authority, consensus must be achieved among untrusted peers.

To resolve this, our protocol relies on the combination of Proof-of-Work and the Longest Chain Rule. If a malicious actor attempts to double-spend by broadcasting conflicting transactions, the network will inevitably fork. Under the consensus rules, only the transaction included in the chain with the most accumulated work is considered valid. Consequently, once a transaction is buried under sufficient subsequent blocks, it becomes immutable, effectively neutralizing the threat of double-spending.

3 System Architecture

This section details the architectural design of the implemented cryptocurrency system. The system is built using an object-oriented approach in Python, modularizing distinct responsibilities such as identity management, state maintenance, and consensus enforcement.

3.1 Core Components and Class Structure

The protocol is composed of distinct classes that interact to maintain the distributed ledger.

- **Wallet:** Responsible for key management and identity. It generates Ed25519 key pairs and derives the Base58Check-encoded address.
- **Transaction (AccountTx):** Represents a state transition request. It contains a sender, recipient, amount, fee, and a strictly increasing nonce to prevent replay attacks.
- **Block:** A container data structure that aggregates validated transactions. It includes a block header containing the Merkle root, timestamp, and PoW nonce.
- **StateManager:** Manages the current state of the ledger (World State). It maintains a mapping of addresses to balances and nonces.
- **TransactionValidator:** A dedicated module responsible for enforcing consensus rules. It verifies signatures, checks balance sufficiency, ensures strictly increasing nonces, and validates adaptive precision constraints.
- **MempoolManager:** Manages unconfirmed transactions. It implements DoS protection by enforcing a maximum pool size and expiring old transactions.
- **PoWChain:** The central controller that orchestrates consensus. It utilizes the Validator and MempoolManager to mine new blocks and resolve forks using the Longest Chain Rule.

3.2 Transaction Lifecycle and Data Flow

The data flow within the system follows a standard verify-then-commit pattern. The lifecycle of a transaction is described as follows:

1. **Creation & Signing:** A user creates an `AccountTx` and signs it using their private key via the `Wallet` class.

2. **Mempool Admission:** The transaction is submitted to the node. The `MempoolManager` performs preliminary checks (e.g., balance sufficiency) and adds it to the pending pool.

3. **Prioritization:** Upon mining, the `PoWChain` selects transactions from the Mempool. Crucially, transactions are sorted by **fee density**, creating a fee market where higher-fee transactions are prioritized.

4. **Execution & Mining:** Selected transactions are tentatively applied to a temporary state. If valid, they are packed into a `Block`, and the PoW puzzle is solved.

5. **Propagation & Consensus:** The valid block is appended to the chain, and the global state (‘State-Manager‘) is updated permanently.

3.3 Hybrid State Model

A distinguishing feature of this architecture is its hybrid nature. While it utilizes the **Proof-of-Work** consensus mechanism typical of Bitcoin, it employs an **Account-based State Model**. This design choice decouples the consensus logic from the state transition logic, allowing for simpler balance lookups compared to the UTXO model, which requires iterating through unspent outputs.

4 Technical Analysis of Implementation

This section provides a detailed analysis of the Python-based implementation, focusing on the core classes that constitute the blockchain architecture.

4.1 Wallet: Identity and Key Management

The `Wallet` class serves as the user’s interface to the network, handling cryptographic identity without storing state. Unlike traditional banking apps, the wallet does not store the balance locally. Instead, the balance is derived from the distributed ledger state to ensure trustlessness.

We employ the **Ed25519** curve for digital signatures due to its performance benefits over standard ECDSA. The address generation process follows a multi-step hashing mechanism to ensure security and brevity:

1. **Key Generation:** Derive the public key from the Ed25519 private key.

2. **Hashing:** Apply SHA-256 followed by RIPEMD-160 to the public key.

3. **Encoding:** Append a checksum and encode using Base58Check.

```
# From Wallet Class
sha = hashlib.sha256(pub_bytes).digest()
hash160 = hashlib.new('ripemd160', sha).digest()
payload = ADDRESS_VERSION_BYTE + hash160
checksum = hashlib.sha256(hashlib.sha256(payload).digest()).digest()[0:4]
return base58.b58encode(payload + checksum).decode()
```

Listing 1: Address Generation Logic

Signature Verification

The system verifies transaction authenticity using the sender’s public key. This ensures that only the owner of the private key can authorize a transfer.

```python
def verify_signature(signature: bytes, message: bytes, public_key_hex: str) ->
    bool:
    try:
        pub_bytes = bytes.fromhex(public_key_hex)
        loaded_pub_key = ed25519.Ed25519PublicKey.from_public_bytes(pub_bytes)
        loaded_pub_key.verify(signature, message)
        return True
    except Exception:
        return False
```

Listing 2: Ed25519 Signature Verification

4.2 Transaction Models

The protocol distinguishes between two transaction types: `CoinbaseTx` for minting new currency and `AccountTx` for peer-to-peer transfers.

Coinbase Transaction

This special transaction type is created by the system to reward miners. It has no sender and serves as the source of all currency supply.

```python
class CoinbaseTx:
    recipient: str
    amount: int
    def txhash(self, hashfunc: Callable = hashlib.sha256) -> str:
        return hash_json({"type": "coinbase", "to": self.recipient, "amt": self.
    amount}, hashfunc)
```

Listing 3: Coinbase Transaction Structure

Account Transaction

The `AccountTx` class encapsulates value transfers between users. Unlike the UTXO model used in Bitcoin, this system uses an account model where a `nonce` field protects against replay attacks.

```python
def message(self) -> dict:
    return {
        "type": "account", "sender": self.sender, "recipient": self.recipient,
        "amount": self.amount, "fee": self.fee, "nonce": self.nonce,
        "timestamp": int(self.timestamp)
    }
```

Listing 4: Account Transaction Message

4.3 Block Structure and Merkle Integrity

The `Block` class acts as a container for validated transactions. To ensure data integrity and allow for efficient verification, transactions are summarized using a **Merkle Tree**. The Merkle Root is stored in the block header, creating a tamper-evident link between the transactions and the block hash.

```python
def merkle_root(hashes: List[str], hashfunc: Callable = hashlib.sha256) -> str:
    if not hashes: return hashfunc(b"").hexdigest()
    layer = hashes[:]
    while len(layer) > 1:
        if len(layer) % 2 == 1: layer.append(layer[-1])
        nxt = []
        for i in range(0, len(layer), 2):
            concat = hashfunc((layer[i] + layer[i+1]).encode()).hexdigest()
            nxt.append(concat)
        layer = nxt
```

```
    return layer[0]
```

Listing 5: Merkle Root Calculation

4.4 State Management and Mempool

The `StateManager` maintains the "World State," tracking balances and nonces for every address. It tightly integrates with the `MempoolManager`, which handles unconfirmed transactions.

Transaction Validation Logic

Before applying a transaction, the validator enforces strict rules: non-negative amounts, valid timestamps, correct signatures, and sufficient funds.

```
class TransactionValidator:
    def validate_and_apply(self, tx: AccountTx, balances: Dict, nonces: Dict,
                           current_difficulty: int = 1) -> bool:
        # 1. Basic Checks
        if tx.amount <= 0 or tx.fee < 0: return False
        if not tx.is_timestamp_valid(): return False
        if not tx.verify_signature(): return False

        # 2. Adaptive Precision Check (New Feature)
        # Checks if decimal places <= current_difficulty
        if not self.check_precision(tx.amount, current_difficulty):
            return False

        # 3. Strict Nonce Ordering (Replay Protection)
        curr_nonce = nonces.get(tx.sender, 0)
        if tx.nonce != curr_nonce: return False

        # 4. Balance Check
        req = tx.amount + tx.fee
        if balances.get(tx.sender, Decimal("0")) < req: return False

        # 5. State Transition
        balances[tx.sender] -= req
        balances[tx.recipient] = balances.get(tx.recipient, Decimal("0")) + tx.
    amount
        nonces[tx.sender] = curr_nonce + 1
        return True
```

Listing 6: Transaction Validator Logic (Updated)

Mempool Management

The Mempool acts as a waiting room for transactions. To prevent resource exhaustion (DoS attacks), it enforces a maximum size. Additionally, transactions that remain unmined for over 100 blocks are automatically expired to maintain network hygiene.

```
def add_transaction(self, tx: AccountTx, current_height: int) -> bool:
    if len(self.pending) >= self.config.max_pending_size: return False
    self.pending.append(tx)
    self.added_height[tx.txhash(self.hashfunc)] = current_height
    return True
```

Listing 7: Mempool Ingestion

4.5 PoW Consensus Engine

The `PoWChain` class orchestrates the entire system. It connects identity (Wallet), value (Transactions), and storage (Blocks) through the Proof-of-Work consensus algorithm.

4.5.1 Mining and Fee Market

Mining incentivizes participation by rewarding nodes with a block reward plus transaction fees. Our implementation specifically includes a **Fee Market**: the miner sorts pending transactions by fee (descending) to maximize profit.

```python
def mine_block(self, miner_wallet) -> Block:
    current_height = len(self.chain)

    # 1. Calculate Next Difficulty FIRST
    # This ensures transactions are validated against the rule of the *new* block
    next_diff = self._get_next_difficulty()

    temp_bal, temp_nonce = self.state.copy_state()
    selected_txs = []

    # 2. Select Transactions from Mempool (Fee Market)
    candidates = sorted(self.mempool.pending, key=lambda x: x.fee, reverse=True)

    for tx in candidates:
        if len(selected_txs) >= self.block_capacity: break

        # Pass 'next_diff' to validator
        if self.validator.validate_and_apply(tx, temp_bal, temp_nonce, next_diff):
            selected_txs.append(tx)

    # 3. Solve Proof-of-Work
    block = Block(..., difficulty=next_diff)
    target = "0" * next_diff
    while not block.blockhash(self.hashfunc).startswith(target):
        block.nonce += 1

    return block
```

Listing 8: Mining Process with Difficulty Look-ahead

4.5.2 Difficulty Adjustment Algorithm

To maintain a stable block generation time despite fluctuating network hash rates, the system implements a dynamic difficulty adjustment mechanism. The `_get_next_difficulty` method is triggered every `ADJUSTMENT_INTERVAL` (set to 5 blocks).

- **Target Time:** The protocol aims for a block time of 1.0 second.
- **Logic:** It compares the time taken to mine the last 5 blocks against the expected time.

$$ActualTime = Timestamp_{Last} - Timestamp_{Last-5} \tag{2}$$

- **Adjustment:**
  - If $ActualTime < TargetTime \times \frac{Interval}{2}$: The difficulty increases by 1 (Mining is too fast).
  - If $ActualTime > TargetTime \times Interval \times 2$: The difficulty decreases by 1 (Mining is too slow).

This simple negative feedback loop ensures the blockchain maintains a consistent heartbeat.

[Image of blockchain fork resolution process]

```python
def attempt_reorg(self, candidate_chain: List[Block]) -> bool:
    # 1. Check Length
    if len(candidate_chain) <= len(self.chain): return False

    # 2. State Replay (Verify all txs in new chain)
    temp_state = StateManager()
    for i, block in enumerate(candidate_chain):
        # ... (Genesis & Header Validation) ...

        # Transaction Execution on Temp State
        for tx in block.transactions[1:]:
             # Adaptive Precision Check included here
            if not self.validator.validate_and_apply(tx, temp_state.balances,
                                                     temp_state.nonces, block.
    difficulty):
                return False

    # 3. Swap Chain
    self.chain = copy.deepcopy(candidate_chain)
    self.state = temp_state
    return True
```

Listing 9: Chain Reorganization Logic

4.5.3 Adaptive Precision

A unique feature of this implementation is **Adaptive Precision**. This mechanism dynamically adjusts the **minimum atomic unit** of the currency based on the network's mining difficulty (D).

The protocol enforces that the number of decimal places in a transaction amount cannot exceed the current difficulty. Mathematically, the smallest divisible unit becomes 10^{-D}.

1. **Implementation:** The system converts the transaction amount to a string and parses the fractional part.
2. **Constraint:** If the length of the fractional part exceeds D, the transaction is rejected as invalid.

For instance, if the current difficulty is **3**, the minimum allowed unit is **0.001**.

- **Valid:** `0.123` (3 decimal places ≤ 3)
- **Invalid:** `0.1234` (4 decimal places > 3)

This effectively prevents "dust spam" (transactions with insignificant value) when the network security is low, allowing finer granularity only as the network matures (i.e., as difficulty increases).

```python
def check_precision(self, amount: Decimal, difficulty: int) -> bool:
    s = str(amount)
    # Integer values are always valid
    if '.' not in s: return True

    # Extract fractional part
    fractional_part = s.split('.')[1]
    decimals = len(fractional_part)

    # Rule: Decimal places must not exceed current difficulty
    # e.g., If Diff=3, 0.123 is OK, 0.1234 is Invalid.
    return decimals <= difficulty
```

Listing 10: Adaptive Precision Implementation

5 Implemented Security Countermeasures

The analyzed codebase includes proactive security features designed to mitigate common attack vectors in distributed systems. We present the actual implementation logic used to prevent Denial of Service (DoS), Replay Attacks, and Time-Warp attacks.

5.1 DoS Mitigation via Mempool Governance

To prevent "spam attacks" where malicious actors flood the network, the `MempoolConfig` enforces strict resource limits. The `MempoolManager` implements the following governance rules:

- **Bounded Memory:** `max_pending_size` prevents memory exhaustion.
- **Rate Limiting:** `max_per_sender` ensures fair access.
- **Economic Cost:** `min_fee` requires attackers to burn resources to fill the mempool.

Prevention Code: The system checks the current size of the mempool before accepting any new transaction. If the pool is full, the transaction is immediately rejected to save RAM.

```
class MempoolManager:
    def add_transaction(self, tx: AccountTx, current_height: int) -> bool:
        # 1. Bounded Memory Check (Anti-Flooding)
        if len(self.pending) >= self.config.max_pending_size:
            return False # Reject immediately to save RAM

        # 2. Add to Pending Pool
        self.pending.append(tx)
        self.added_height[tx.txhash(self.hashfunc)] = current_height
        return True
```

Listing 11: Mempool Capacity Enforcement (DoS Protection)

5.2 Replay Attack Prevention

In an account-based model, a valid signed message could theoretically be intercepted and re-broadcast by an attacker to drain the sender's funds repeatedly. We mitigate this using a strictly increasing **Nonce** (Number used once).

Prevention Code: The `TransactionValidator` compares the transaction's nonce against the stored nonce in the `StateManager`. They must match exactly.

```
# From TransactionValidator.validate_and_apply
curr_nonce = nonces.get(tx.sender, 0)

# The nonce must be exactly current_nonce (no gaps, no repeats)
if tx.nonce != curr_nonce:
    return False

# After execution, nonce is incremented
nonces[tx.sender] = curr_nonce + 1
```

Listing 12: Nonce Verification Logic

5.3 Temporal Integrity (Time-Warp Attack Prevention)

Miners might attempt to manipulate block timestamps to artificially lower the difficulty or mine blocks faster than allowed. The protocol enforces strict boundaries to maintain temporal integrity:

- `MAX_FUTURE_DRIFT` (2 hours): Blocks with timestamps too far in the future are rejected.
- `Chronological Ordering`: A block must have a timestamp greater than its predecessor.

Prevention Code: The `PoWChain` rejects any block that violates these temporal constraints.

```python
# From PoWChain.validate_block
def validate_block(self, block: Block, previous_block: Block) -> bool:
    # 1. Future Drift Check (Limit: 2 hours)
    if block.timestamp > time.time() + MAX_FUTURE_DRIFT:
        return False

    # 2. Chronological Check
    if block.timestamp <= previous_block.timestamp:
        return False

    return True
```

Listing 13: Timestamp Validation Logic

6 Simulation and Results

To validate the theoretical architecture, we conducted a series of simulations using the Python prototype. The simulation covers three primary scenarios: standard operations, adversarial attempts (double spending), and consensus resolution (forks).

6.1 Case 1: Standard Transactions and Mining

In this scenario, we verified the basic lifecycle of the cryptocurrency: wallet creation, transaction signing, and block mining.

Setup:

- Two nodes (Node A, Node B) and three wallets (Alice, Bob, Miner).
- Alice sends transactions to Bob with varying fees.
- The miner attempts to mine a block including these transactions.

Result: The simulation successfully demonstrated the **Fee Market** mechanism. As implemented in the `mine_block` function, the miner prioritized transactions with higher fees.

- Transaction 1 (Fee: 0.01) was delayed.
- Transaction 2 (Fee: 5.00) was included in the immediate next block.

This confirms that the economic incentive model functions correctly, prioritizing users willing to pay higher fees for block space.

6.2 Case 2: Double Spending Prevention Simulation

We simulated a ”Replay Attack” scenario where a malicious user (Alice) attempts to broadcast two conflicting transactions using the same nonce.

```
# [Scenario] Alice's Nonce is 1. Sign and broadcast to different recipients simultaneously.

tx_A = AccountTx(alice.address, bob.address, Decimal("50"), fee=Decimal("1"), nonce=1)
tx_B = AccountTx(alice.address, attacker.address, Decimal("50"), fee=Decimal("1"), nonce=1)
tx_A.sign(alice)
tx_B.sign(alice)

print("Broadcasting Tx A (Alice -> Bob)...")
if sec_node.blockchain.add_transaction(tx_A):
    print(" -> Tx A accepted.")
else:
    print(" -> Tx A rejected.")

print("Broadcasting Tx B (Same Nonce)...")
# Should be rejected due to Mempool Collision check (PoWChain's add_transaction)
if sec_node.blockchain.add_transaction(tx_B):
    print(" FAIL: Double Spend Accepted (Vulnerability!).")
else:
    print(" SUCCESS: Double Spend Rejected (Mempool Collision).")
```

Listing 14: Double Spending Simulation Code

Result: The system correctly identified the duplicate nonce in `tx_double`. The console output confirmed:

```
-> SUCCESS: Double Spend Rejected (Mempool Collision).
```

This proves that the `StateManager` and `TransactionValidator` effectively maintain ledger consistency against malicious duplication attempts.

6.3 Case 3: Fork Resolution (Chain Reorganization)

This is the most critical test for a distributed system. We simulated a network partition where two nodes have different versions of the blockchain.

Scenario:

- **Node A** mines 3 blocks (Chain Height: 3).
- **Node B** mines only 1 block (Chain Height: 1).
- Node B connects to Node A and initiates synchronization using `attempt_reorg`.

Result: Node B recognized Node A's chain as the ”Longest Valid Chain”.

1. Node B downloaded Node A's blocks.
2. Node B replayed all transactions from the genesis block to ensure validity.
3. Node B replaced its local chain with Node A's chain.

6.4 Security Analysis Simulation

Beyond standard operations, we conducted adversarial simulations to verify the system's security features. The following scenarios were executed using the `SecurityNode` environment.

6.4.1 Case 4: Replay Attack Prevention

We simulated an attacker attempting to broadcast a previously confirmed transaction again.

- **Attack Vector:** The attacker captured a valid signed transaction (Nonce 0) from Alice to Bob and re-broadcast it after it was mined.
- **Result:** The `TransactionValidator` detected that Alice's current nonce had already incremented to 1. The replayed transaction (Nonce 0) was strictly rejected.
- **Console Output:** `SUCCESS: Replay Attack Rejected (Nonce mismatch).`

6.4.2 Case 5: DoS Mitigation via Mempool Limits

To test resilience against spam flooding, we reduced the mempool capacity to 5 slots and attempted to flood the node with 10 valid transactions simultaneously.

- **Attack Vector:** 10 distinct attacker wallets sent valid transactions to the node within a short timeframe.
- **Result:** The `MempoolManager` accepted the first 5 transactions. The subsequent 5 transactions were immediately rejected with `False` return values, preventing memory exhaustion.
- **Console Output:** `SUCCESS: DoS Protection Working. Accepted 5/10 transactions.`

This confirms that the resource governance logic effectively protects the node from basic flooding attacks.

7 Vulnerabilities and Limitations

While the current implementation successfully demonstrates the core mechanics of a blockchain, it operates as a simplified prototype ("Toy Model"). Several architectural limitations must be addressed for a production-grade environment.

7.1 Data Persistence (Volatile Memory)

Weakness: The current `StateManager` stores all account balances and nonces in Python dictionaries (RAM).

- If the node software terminates or crashes, the entire ledger state is lost.
- As the blockchain grows, the memory requirement scales linearly ($O(n)$), eventually exceeding available RAM.

Improvement: Integration with a key-value store database such as **LevelDB** or **RocksDB** is required to persist the state on disk and allow for efficient querying of historical data.

7.2 Network Layer Abstraction

Weakness: The simulation models P2P networking via direct method calls between objects in a single process (e.g., `node_B.sync(node_A)`). It lacks a true network stack.

- There is no implementation of peer discovery, TCP/IP socket handling, or gossip protocols.
- Network latency and packet loss, which are critical in real-world consensus convergence, are not simulated.

Improvement: Implementing a dedicated P2P networking module using Python's `asyncio` or libraries like `libp2p` is necessary to simulate real-world propagation delays.

7.3 Mining Algorithm Efficiency

Weakness: The Proof-of-Work loop is implemented in pure Python.

```
while not hash.startswith(target): nonce += 1
```

Python's Global Interpreter Lock (GIL) and interpreted nature make this extremely inefficient compared to C++ or hardware implementations (ASICs). This makes the current prototype unsuitable for high-difficulty environments.

7.4 Lack of Merkle Proof Verification

Weakness: Although the `Block` class calculates a Merkle Root, the system lacks the functionality for "Simplified Payment Verification" (SPV). Currently, there is no method for a light client to verify transaction inclusion without running a full node.

8 Conclusion

This report has presented a comprehensive design and implementation analysis of a Python-based cryptocurrency prototype. By synthesizing the **Proof-of-Work** consensus mechanism of Bitcoin with the **Account-based Model** of Ethereum, we constructed a hybrid architecture that successfully demonstrates the core principles of decentralized ledgers.

8.1 Key Achievements

The implementation achieved several technical milestones:

- **Modern Cryptography:** The successful integration of **Ed25519** signatures demonstrated that blockchain systems can achieve higher performance and smaller key sizes compared to legacy ECDSA implementations.
- **Economic Security:** The introduction of **Adaptive Precision** provided a novel approach to spam mitigation, dynamically linking the currency's atomic unit to the network's security (difficulty). Furthermore, the **Fee Market** simulation confirmed that rational miners correctly prioritize transactions based on economic incentives.
- **Consensus Resilience:** The system proved resilient against critical failures. The **Fork Resolution** simulation validated the "Longest Chain Rule," ensuring that the network eventually converges to a single version of the truth even in the presence of partitions.

8.2 Future Directions

While this prototype serves as a robust educational tool, evolving it into a production-ready protocol requires addressing specific limitations identified in Section 7. Future development should focus on implementing a **Persistent Storage Layer** (e.g., LevelDB) to prevent data loss and a full **P2P Network Stack** to replace the current in-memory simulation. Additionally, optimizing the mining algorithm from Python to C++ (or adopting ASIC-friendly algorithms) would be essential for realistic security guarantees.

In conclusion, this project validates that a lightweight, Python-based implementation can effectively model complex distributed consensus behaviors, providing a valuable foundation for further research into blockchain scalability and security.

References

[1] Chaum, D. (1983). Blind signatures for untraceable payments. *Advances in Cryptology*, 199-203.

[2] Szabo, N. (2005). Bit Gold. *Unenumerated (Blog)*. Retrieved from `https://unenumerated.blogspot.com/`

[3] Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. *Bitcoin.org*. Retrieved from `https://bitcoin.org/bitcoin.pdf`

[4] Buterin, V. (2014). Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. *Ethereum Whitepaper*.

[5] Johnson, D., Menezes, A., & Vanstone, S. (2001). The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1), 36-63.

[6] Certicom Research. (2010). SEC 2: Recommended Elliptic Curve Domain Parameters. *Standards for Efficient Cryptography*.