Blockchain Consensus Protocols: Implementation and Analysis

Vemula Chandrahaas Reddy - CS22BT062
Soumyadeep Das - CS22BT056

Abstract

This report presents a comprehensive analysis of blockchain consensus protocols. We detail our implementation approach, the development process, and performance analysis of these protocols. Additionally, we examine their resilience against common attack vectors: the 51% attack, Sybil Attack and Denial of Service (DoS) attack. Our findings indicate significant differences in energy consumption, throughput, and security properties between these protocols.

Contents

1 Introduction

Blockchain technology relies on consensus protocols to maintain a secure and consistent ledger across a distributed network. These protocols enable participants to agree on the state of the blockchain without requiring trust in a central authority.

The most prominent consensus mechanisms are:

- **Proof of Work (PoW)**: Miners compete to solve cryptographic puzzles, with the first to succeed earning the right to add a new block to the chain.
- **Proof of Stake (PoS)**: Validators are chosen to create new blocks based on the amount of cryptocurrency they hold and are willing to "stake" as collateral.
- **Proof of Burn (PoB)**: Validators burn (permanently destroy) coins to gain the right to mine blocks, with selection probability proportional to their burn amount relative to the network.
- **Proof of Elapsed Time (PoET)**: Validators are assigned random wait times through a trusted execution environment, with the first to complete its wait period earning the right to create a block.
- **Proof of Authority (PoA)**: A select group of authorized validators take turns creating blocks in a deterministic sequence, with block validity depending on the creator's authority rather than computational work or economic stake.

This report documents our implementation of all the above mentioned protocols from principles, analyzes their performance characteristics, and evaluates their resistance to common attack vectors.

2 Approach and Implementation

2.1 Core Blockchain Architecture

Our implementation follows a modular design that separates the consensus layer from the core blockchain functionality. The system comprises several key components:

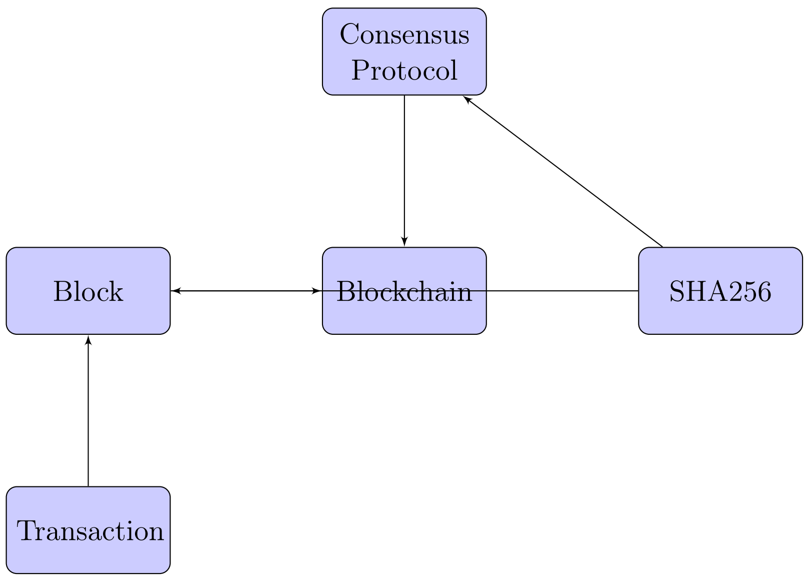


Figure 1: System Architecture Overview

2.1.1 Block Class

The Block class represents individual blocks in the blockchain. Each block contains:

- Parent hash (reference to previous block)
- Nonce (used in the consensus process)
- Difficulty (determines how hard it is to find a valid hash)
- Timestamp (when the block was created)
- Merkle root (hash representation of all transactions)
- Transaction list (actual data stored in the block)
- Block hash (cryptographic identifier for the block)

```cpp
class Block {
public:
    std::string parentHash, difficulty, merkleRoot, hash;
    int nonce;
    time_t timestamp;
    std::vector<Transaction> transactions;

    Block();
    Block(const std::string& prevHash, const std::vector<Transaction>& txs);

    std::string calculateHash() const;
    static std::string calculateMerkleRoot(const std::vector<Transaction>&
    transactions);
};
```

Listing 1: Block Class Implementation

2.1.2 Blockchain Class

The Blockchain class manages the entire chain of blocks and provides methods to:

- Add new blocks to the chain
- Display blockchain information
- Access the latest block (tip) in the chain

```cpp
class Blockchain {
private:
    std::unordered_map<std::string, Block> chain;
    std::vector<std::string> blockOrder;
    std::string tipHash;

public:
    Blockchain();
    void addBlock(const Block& block);
    void displayBlock(const Block& block) const;
    void displayBlockchainHashes() const;
    std::string getTip() const;
};
```

Listing 2: Blockchain Class Implementation

2.1.3 Transaction Structure

The Transaction structure represents transfers of value between addresses:

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

    Transaction(std::string s, std::string r, double a)
        : sender(s), receiver(r), amount(a) {}
};
```

Listing 3: Transaction Structure

2.2 Proof of Work Implementation

Our PoW implementation follows these key steps:

1. Initialize miners with transaction subsets
2. Each miner attempts to find a block hash that meets difficulty requirements
3. The first miner to succeed broadcasts their block
4. Other miners verify the block's validity
5. If majority verification is achieved, the block is added to the chain

```cpp
void mineBlock(Blockchain& blockchain, std::vector<Transaction> txs, std::string difficulty, int minerId) {
    std::string parentHash = blockchain.getTip();
    int nonce = 0;
    time_t timestamp = time(nullptr);
    long long localHashCount = 0;

    while (!blockFound.load()) {
        Block candidate;
        candidate.parentHash = parentHash;
        candidate.transactions = txs;
        candidate.timestamp = timestamp;
        candidate.nonce = nonce++;
        candidate.difficulty = difficulty;
        candidate.merkleRoot = Block::calculateMerkleRoot(candidate.transactions);
        candidate.hash = candidate.calculateHash();

        localHashCount++;

        if (isValidHash(candidate.hash, difficulty)) {
            std::unique_lock<std::mutex> lock(mtx);
            if (!blockFound.load()) {
                blockFound = true;
                winningBlock = candidate;
                std::cout << "[Miner " << minerId << "] found a block! Broadcasting...\n";
                cv.notify_all();
            }
            totalHashAttempts += localHashCount;
            return;
        }
    }
    totalHashAttempts += localHashCount;
}
```

Listing 4: PoW Mining Function

The mining process is computationally intensive, requiring miners to repeatedly calculate hashes with different nonce values until finding one that meets the difficulty criteria. This ensures that creating blocks requires significant computational work, making it economically unfeasible to alter the blockchain's history.

2.3 Proof of Stake Implementation

Our PoS implementation follows these key steps:

1. Select a validator probabilistically based on stake amounts
2. The chosen validator proposes a block with their selected transactions
3. The validator must still find a valid hash, but with reduced difficulty
4. Other validators verify the block's validity
5. If majority verification is achieved, the block is added and the validator rewarded
6. If verification fails, the validator is penalized

```
int selectValidator(const map<int, int>& stakes) {
    int totalStake = 0;
    for (auto& [minerId, stake] : stakes)
        totalStake += stake;

    if (totalStake == 0) {
        cout << "Error: Total stake is zero. Cannot select validator.\n";
        return -1;
    }

    int randomValue = rand() % totalStake;
    int runningSum = 0;

    for (auto& [minerId, stake] : stakes) {
        runningSum += stake;
        if (randomValue < runningSum) return minerId;
    }

    return stakes.begin()->first; // fallback
}
```

Listing 5: PoS Validator Selection Function

The key innovation in PoS is the validator selection mechanism, which weights participants by their stake rather than computational power. This creates an economic incentive for validators to act honestly, as they risk losing their stake if they attempt to validate invalid transactions.

2.4 Proof of Elapsed Time (PoET) Implementation

In addition to PoW and PoS, we implemented a third consensus mechanism: Proof of Elapsed Time (PoET). This Intel-developed consensus algorithm provides an energy-efficient alternative to PoW while maintaining similar security properties.

2.4.1 PoET Mechanism Overview

The core principle of PoET is:

1. Each validator requests a random wait time from a trusted execution environment (TEE)
2. All validators sleep for their assigned duration
3. The first validator to complete its wait time creates and broadcasts a new block
4. Other validators verify the wait certificate and block contents
5. If majority verification is achieved, the block is added to the chain

This approach eliminates the computational waste of PoW while still ensuring a random, fair validator selection process through trusted hardware.

2.4.2 Block Structure for PoET

We extended our Block class to support PoET-specific fields:

```cpp
class Block {
public:
    std::string parentHash, difficulty, merkleRoot, hash;
    int nonce;
    time_t timestamp;
    std::vector<Transaction> transactions;

    // PoET-specific fields
    std::string waitCertificate;  // In a real implementation,
   cryptographically verified
    int waitTime;                 // The amount of time this miner waited

    Block();
    Block(const std::string& prevHash, const std::vector<Transaction>& txs);
    // Constructor for PoET blocks
    Block(const std::string& prevHash, const std::vector<Transaction>& txs, int
     wait);

    std::string calculateHash() const;
    static std::string calculateMerkleRoot(const std::vector<Transaction>&
   transactions);
};
```

Listing 6: PoET-enhanced Block Class

The key additions are:

- `waitCertificate`: In a real implementation, this would be a cryptographically signed attestation from the TEE that the validator actually waited the specified time.
- `waitTime`: The duration in milliseconds that the validator was assigned to wait.
- A dedicated constructor for PoET-based blocks that handles wait time assignment.

2.4.3 PoET Consensus Implementation

The core of our PoET implementation is the miner wait process:

```cpp
void minerWaitProcess(Blockchain& blockchain, vector<Transaction> txs, int
    minerId) {
    string parentHash = blockchain.getTip();

    // Generate a random wait time (for simulation purposes)
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> waitDist(1000, 10000); // Wait between 1-10
    seconds
    int waitTime = waitDist(gen);

    cout << "[Miner " << minerId << "] received wait time of " << waitTime
    /1000.0 << " seconds...\n";

    // Create candidate block with PoET-specific constructor
    Block candidate(parentHash, txs, waitTime);

    // Wait for the random time
    auto start = chrono::steady_clock::now();

    while (!blockFound.load()) {
        auto now = chrono::steady_clock::now();
        auto elapsedMs = chrono::duration_cast<chrono::milliseconds>(now -
    start).count();

        if (elapsedMs >= waitTime) {
            unique_lock<mutex> lock(mtx);
            if (!blockFound.load()) {
                blockFound = true;
                winningBlock = candidate;
                cout << "[Miner " << minerId << "] completed wait time first!
    Broadcasting block...\n";
                cv.notify_all();
            }
            return;
        }
        // Sleep for a small amount of time to reduce CPU usage
        this_thread::sleep_for(chrono::milliseconds(10));
    }
}
```

Listing 7: PoET Mining Process

The verification process ensures the integrity of the wait certificate:

```cpp
bool verifyWaitCertificate(const Block& block) {
    // In a real implementation, we would verify with TEE:
    // - The wait time was legitimately random
    // - The miner actually waited the reported time
    // - The certificate was issued by a valid TEE

    // For simulation purposes, we assume the wait certificate is always valid
    return true;
}

void verifyBlock(const Blockchain& blockchain, const Block& block, int minerId,
     int attack) {
    cout << "[Miner " << minerId << "] verifying the block...\n";

    int attackThreshold = totalMiners / 2;

    if (attack == 1 && minerId >= attackThreshold) {
        std::cout << "[Miner " << minerId << "] Block verification ARTIFICIALLY
     failed! (51\% attack simulation)\n";
        return; // These miners don't increment verifiedMiners counter
    }

    // Verify that the wait certificate is valid
    if (verifyWaitCertificate(block) && block.hash == block.calculateHash()) {
        cout << "[Miner " << minerId << "] Block verified successfully! Wait time was "
             << block.waitTime/1000.0 << " seconds.\n";
        this_thread::sleep_for(chrono::milliseconds(500));
        verifiedMiners++;
    } else {
        cout << "[Miner " << minerId << "] Block verification failed! Invalid wait certificate.\n";
    }
}
```

Listing 8: PoET Block Verification

The complete PoET round orchestration is managed by the following function:

```
Block startPoETRound(Blockchain& blockchain, vector<vector<Transaction>> minerTxsList, int attack) {
    totalHashAttempts = 0;
    totalMiners = minerTxsList.size();
    blockFound = false;
    verifiedMiners = 0;

    if (attack == 1) {
        int attackThreshold = totalMiners / 2;
        int attackingMiners = totalMiners - attackThreshold;
        std::cout << "ATTACK MODE ENABLED: " << attackingMiners << " miners will reject valid blocks\n";
        std::cout << "Attacking miners: " << attackingMiners << " ("
                  << (attackingMiners * 100.0 / totalMiners) << "\% of network)\n";
    }

    vector<thread> miners;

    for (size_t i = 0; i < minerTxsList.size(); ++i) {
        if (attack == 2 && (i == 1 || i == 2)) {
            cout << "transactions selected by miner " << i << ":\n";
            for (const auto& tx : minerTxsList[i]) {
                cout << "  " << tx.sender << " -> " << tx.receiver << " [" << tx.amount << "]\n";
            }
            vector<Transaction> filtered;
            for (const auto& tx : minerTxsList[i]) {
                if (tx.sender != "1" && tx.receiver != "1") {
                    filtered.push_back(tx);
                } else if(i == 1) {
                    cout << "[Miner 1] Skipping transaction involving user 1: "
                         << tx.sender << " -> " << tx.receiver << " [" << tx.amount << "]\n";
                } else {
                    cout << "[Miner 2] Skipping transaction involving user 1: "
                         << tx.sender << " -> " << tx.receiver << " [" << tx.amount << "]\n";
                }
            }
            miners.emplace_back(minerWaitProcess, ref(blockchain), minerTxsList[i], i);
        } else {
            miners.emplace_back(minerWaitProcess, ref(blockchain), minerTxsList[i], i);
        }
    }

    {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [] { return blockFound.load(); });
    }

    // Join all miner threads
    for (auto& miner : miners) {
        if (miner.joinable()) {
            miner.join();
        }
    }

    // After the block is found, have all miners verify it
    cout << "All miners are verifying the block...\n";
    vector<thread> verificationThreads;
    for (int i = 0; i < totalMiners; ++i) {
        verificationThreads.emplace_back(verifyBlock, cref(blockchain), cref(
    winningBlock), i, attack);
    }

    // Wait for all miners to finish their verification
    for (auto& t : verificationThreads) {
        if (t.joinable()) {
            t.join();
        }
    }

    double energyPerHash = 0.0000015; // joules per hash
    double totalEnergy = totalHashAttempts * energyPerHash;

    std::cout << "Total hash attempts: " << totalHashAttempts << "\n";
    std::cout << "Estimated energy consumed: " << totalEnergy << " joules\n";

    if (verifiedMiners > totalMiners * 1/2) {
        blockchain.addBlock(winningBlock);
        cout << "Block successfully added to the blockchain! (" <<
    verifiedMiners << "/" << totalMiners << " verifications)\n";
    } else {
        cout << "Block verification failed: only " << verifiedMiners << "/" <<
    totalMiners << " miners verified. It will not be added.\n";
    }

    return winningBlock;
}
```

Listing 9: PoET Consensus Round

2.5 Proof of Burn Implementation

Our Proof of Burn implementation follows these key steps:

1. Miners burn coins (send to an unspendable address) to participate in block validation
2. A validator is selected probabilistically based on burn records, with higher burns increasing selection probability
3. The chosen validator proposes a block with their selected transactions
4. Other validators verify the block's validity

5. If majority verification is achieved, the block is added to the chain

Unlike PoW which requires ongoing energy expenditure, PoB frontloads the cost by permanently destroying coins as an investment in future block creation rights. This creates a form of "virtual mining" through financial sacrifice rather than computation.

```
struct BurnRecord {
    double burnedCoins;
    time_t timestamp;

    BurnRecord(double coins, time_t time) : burnedCoins(coins), timestamp(time)
     {}
};
```

Listing 10: PoB Burn Record Structure

A key innovation in our PoB implementation is the effective burn power calculation, which applies a time decay factor to prioritize recent burns:

```
double calculateEffectiveBurnPower(const vector<BurnRecord>& records) {
    double totalPower = 0.0;
    time_t currentTime = time(nullptr);

    // Calculate the burn power with a time decay factor
    // More recent burns have more weight
    for (const auto& record : records) {
        // Linear decay over 100 milliseconds
        double ageFactor = 1.0 - min(1.0, double(currentTime - record.timestamp
    ) / 0.1);
        totalPower += record.burnedCoins * ageFactor;
    }

    return totalPower;
}
```

Listing 11: PoB Effective Burn Power Calculation

The validator selection process weights participants by their effective burn power:

```
int selectValidator(const map<int, vector<BurnRecord>>& burnRecords) {
    // Calculate effective burn power for each miner
    map<int, double> effectivePower;
    double totalPower = 0.0;

    for (const auto& [minerId, records] : burnRecords) {
        effectivePower[minerId] = calculateEffectiveBurnPower(records);
        totalPower += effectivePower[minerId];
    }

    // If no one has burned coins, select randomly
    if (totalPower <= 0) {
        int randomMiner = rand() % burnRecords.size();
        auto it = burnRecords.begin();
        advance(it, randomMiner);
        return it->first;
    }

    // Weighted random selection based on burned coins
    double randomValue = (double)rand() / RAND_MAX * totalPower;
    double runningSum = 0.0;

    for (const auto& [minerId, power] : effectivePower) {
        runningSum += power;
        if (randomValue < runningSum) return minerId;
    }

    return burnRecords.begin()->first; // fallback
}
```

Listing 12: PoB Validator Selection Function

The process of finding a valid block still requires a limited amount of computational work, ensuring blocks cannot be created instantly even with large burn amounts:

```
Block newBlock(blockchain.getTipHash(), selectedTxs);
string difficulty = "0000";
newBlock.difficulty = difficulty;
totalHashAttempts++;

while(!isValidHash(newBlock.hash, difficulty)) {
    newBlock.nonce++;
    newBlock.hash = newBlock.calculateHash();
    totalHashAttempts++;
}
```

Listing 13: PoB Block Creation

This hybrid approach combines the economic incentives of coin burning with a modest computational requirement, creating a more energy-efficient system than pure PoW while maintaining strong security properties. Also for uniformilty in simulation across models, we havent added then trasaction generated due to burning coins by miners into transaction pool.

2.6 Proof of Authority Implementation

Our PoA implementation follows these key steps:

1. A fixed set of pre-approved authority nodes are established as trusted validators
2. Validators are selected in a round-robin fashion to create blocks
3. The selected validator creates a block with their chosen transactions
4. Other validators verify the block creator's authority and the block's validity
5. If majority verification is achieved, the block is added to the chain

```
struct Authority {
    std::string name;
    std::string publicKey;
    bool isActive;

    Authority() : name(""), publicKey(""), isActive(false) {}
    Authority(const std::string& n, const std::string& key)
        : name(n), publicKey(key), isActive(true) {}
};
```

Listing 14: PoA Authority Structure

The core of our implementation is the validator selection mechanism, which rotates through authorized validators sequentially rather than using computational puzzles or token economics:

```
string PoAConsensus::getNextValidator() {
    lock_guard<mutex> lock(mtx);
    if (validatorOrder.empty()) {
        return "";
    }

    string nextValidator = validatorOrder[currentValidatorIndex];
```

```cpp
    currentValidatorIndex = (currentValidatorIndex + 1) % validatorOrder.size();
    return nextValidator;
}
```

Listing 15: PoA Validator Selection Function

Unlike PoW or PoS, block creation in PoA is computationally trivial but requires authorization verification:

```cpp
Block PoAConsensus::createBlock(const string& validatorPublicKey,
                               const string& parentHash,
                               const vector<Transaction>& txs) {
    if (!isAuthorized(validatorPublicKey)) {
        throw runtime_error("Unauthorized validator");
    }

    return Block(parentHash, txs, validatorPublicKey);
}
```

Listing 16: PoA Block Creation

The block verification process centers on authenticating the validator's authority rather than checking proof of computational work:

```cpp
bool PoAConsensus::validateBlockAuthority(const Block& block, const string& validatorPublicKey, int attack, int validatorIndex) const {
    cout << "[Authority " << validators.at(validatorPublicKey).name << "] verifying the block...\n";

    int attackThreshold = validatorOrder.size() / 2;

    if (attack == 1 && validatorIndex >= attackThreshold) {
        cout << "[Authority " << validators.at(validatorPublicKey).name
             << "] Block verification ARTIFICIALLY failed! (51\% attack simulation)\n";
        return false;
    }

    if (!isAuthorized(validatorPublicKey)) {
        cout << "[Authority " << validators.at(validatorPublicKey).name
             << "] Block verification failed: Unauthorized validator.\n";
        return false;
    }

    stringstream ss;
    ss << block.parentHash << block.timestamp << block.merkleRoot << block.validatorPublicKey;
    string expectedHash = sha256(ss.str());

    if (block.hash == expectedHash) {
        cout << "[Authority " << validators.at(validatorPublicKey).name << "] Block verified successfully!\n";
        this_thread::sleep_for(chrono::milliseconds(1000));
        verifiedMiners++;
        totalHashAttempts++;
        return true;
    } else {
        cout << "[Authority " << validators.at(validatorPublicKey).name
             << "] Block verification failed: Hash mismatch.\n";
        totalHashAttempts++;
        return false;
    }
}
```

Listing 17: PoA Block Verification

The full PoA round includes the validator selection, block creation, and a distributed verification process:

```cpp
Block PoAConsensus::startBlockCreationRound(Blockchain& blockchain,
                                            vector<vector<Transaction>>& validatorTxsLists,
                                            int attack) {
    // Reset counters
    totalHashAttempts = 0;
    verifiedMiners = 0;
    totalMiners = validatorTxsLists.size();

    // Select next validator in sequence
    string currentValidator = getNextValidator();
    if(currentValidator.empty()) {
        throw runtime_error("No active validators available");
    }

    const Authority& validator = validators.at(currentValidator);
    cout << validator.name << " selected for this round.\n";

    // Find validator index in the ordered list
    size_t validatorIndex = 0;
    while (validatorIndex < validatorOrder.size() &&
           validatorOrder[validatorIndex]!=currentValidator) {
        validatorIndex++;
    }

    // Select transactions
    vector<Transaction> txs;
    if (validatorIndex < validatorTxsLists.size()) {
        txs = validatorTxsLists[validatorIndex];
    }

    // Create block
    cout << validator.name << " is creating a block...\n";
    Block newBlock = createBlock(currentValidator, blockchain.getTip(), txs);
    totalHashAttempts++;

    // Validation phase
    cout << "All validators are verifying the block...\n";
    vector<thread> verificationThreads;
    vector<bool> verificationResults(validatorOrder.size());
    for (size_t i = 0; i < validatorOrder.size(); ++i) {
        verificationThreads.emplace_back([&, i]() {
            verificationResults[i] = validateBlockAuthority(newBlock,
                                                            validatorOrder[i],
                                                            attack, i);
        });
    }

    for (auto& t : verificationThreads) {
        if (t.joinable()) {
            t.join();
        }
    }

    // Count verifications and add block if majority approves
    verifiedMiners = count(verificationResults.begin(), verificationResults.end(), true);

    if (verifiedMiners > totalMiners/2) {
        cout << "Block successfully validated by authorities!\n";
        blockchain.addBlock(newBlock);
        return newBlock;
    } else {
        cout << "[Validator " << validator.name << "] proposed an invalid block";
        cout << "Block verification failed by some validators. It will not be added.\n";
        return Block{};
    }
}
```

Listing 18: PoA Consensus Round

The key innovation in PoA is replacing the resource-intensive mechanisms of other consensus protocols with a trust-based approach where a limited set of pre-approved validators take turns creating blocks. This creates significant energy efficiency gains at the cost of increased centralization.

3 Security Analysis

We implemented simulations for common attack vectors to evaluate the security properties of both consensus protocols:

3.1 51% Attack

A 51% attack occurs when a malicious entity controls more than half of the network's resources. Our simulation divides the miners/validators into two groups, with the attacking group deliberately rejecting valid blocks.

```
if (attack == 1 && minerId >= attackThreshold) {
    std::cout << "[Miner " << minerId << "] Block verification ARTIFICIALLY failed! (51\% attack simulation)\n";
    return; // These miners don't increment verifiedMiners counter
}
```

Listing 19: 51% Attack Simulation in Block Verification

Our findings show that all five protocols are vulnerable to 51% attacks, as expected. However, the economic cost and practical feasibility of mounting such attacks differ significantly:

- In PoW, the attacker must acquire and operate physical mining hardware exceeding the honest network's capacity.
- In PoS, the attacker must acquire and stake a cryptocurrency amount exceeding the honest validators' combined stake.
- In PoB, the attacker must burn (permanently destroy) more coins than the honest validators, representing a substantial irreversible financial investment.
- In PoET, the attacker must control more than half of the trusted execution environments (TEEs) in the network, which is primarily a hardware constraint rather than an economic one.
- In PoA, the attacker must compromise or control more than half of the pre-approved authority nodes, which is primarily an institutional challenge rather than an economic or computational one.

Both PoS and PoB make attacks economically self-defeating at scale. PoET's security model differs fundamentally as it relies on the integrity of hardware-based trusted execution environments (like Intel SGX) rather than economic incentives. This creates a different attack surface that depends on the security of the TEE implementation rather than network economics.PoA's security model differs fundamentally as it relies on the reputation and trusted status of designated validators rather than economic incentives or computational resources. This makes it particularly well-suited for permissioned blockchain networks where validators' identities are known and vetted, but potentially vulnerable to collusion attacks in situations where validators can be corrupted.

In our simulation, when a 51% attack occurs—implying that the majority of nodes disagree with the current block addition—no new block is added to the blockchain beyond the genesis block. As a result, the pending transactions remain in the transaction pool.

This scenario may appear as if the simulation has entered an infinite loop, which accurately reflects the stalemate that can occur in real-world blockchain networks under a 51% attack. To avoid such an indefinite state, we terminate the simulation after a predefined threshold number of rounds.(*int totalRounds*)

3.2 Denial of Service (DoS) Attack

A DoS attack in our simulation involves miners/validators deliberately filtering out transactions from specific users. This simulates an attempt to prevent certain addresses from participating in the network.

```cpp
if (attack == 2 && (i == 1 || i == 2)) {
    vector<Transaction> filtered;
    for (const auto& tx : minerTxsList[i]) {
        if (tx.sender != "1" && tx.receiver != "1") {
            filtered.push_back(tx);
        } else {
            cout << "[Miner " << i << "] Skipping transaction involving user 1: "
                 << tx.sender << " -> " << tx.receiver << " [" << tx.amount << "]\n";
        }
    }
    miners.emplace_back(minerWaitProcess, ref(blockchain), filtered, i);
} else {
    miners.emplace_back(minerWaitProcess, ref(blockchain), minerTxsList[i], i);
}
```

Listing 20: DoS Attack Simulation

Our analysis shows that all four protocols demonstrate some resilience to DoS attacks, as transactions rejected by attacking miners can still be included by honest miners in subsequent blocks. However, if the attackers control a significant portion of the network, they can substantially delay transactions from targeted users.

In the PoET implementation, the censorship resistance properties are similar to the other protocols, as the random wait time allocation is independent of the transaction selection process. However, PoET may be more resilient to targeted transaction censorship because validator selection is random and determined by the TEE rather than by economic power, potentially making it more difficult for attackers to consistently control block production.

3.3 Sybil Attack

A Sybil attack is a type of security threat in decentralized networks where a single attacker creates and controls multiple fake identities (or "nodes") to gain a disproportionately large influence over the network, we have implement the same in our simulation for PoW.

Impact on Different Consensus Mechanisms

- **Proof of Work (PoW):**
  Resistant to Sybil attacks since influence depends on computational power, not the number of identities. Fake nodes are ineffective without sufficient hash power. Our implementation explore further by considering what would happen if fake node have desired computational power.
- **Proof of Stake (PoS):**
  Moderately vulnerable. Sybil identities require actual stake, but an attacker may split their stake across many identities to increase selection probability.
- **Proof of Burn (PoB):**
  Moderately resistant. Attackers must burn coins per identity, making large-scale Sybil attacks economically unfeasible.
- **Proof of Authority (PoA):**
  Highly vulnerable if identity verification is weak. If an attacker can register multiple identities as trusted validators, they can control consensus.
- **Proof of Elapsed Time (PoET):**
  Moderately vulnerable. Relies on trusted hardware. If the attacker can fake or emulate trusted environments, they can run multiple Sybil nodes to manipulate leader selection.

3.4 Double Spending Attack

A Double Spending Attack involves an attacker attempting to spend the same cryptocurrency or digital asset more than once. This is achieved by broadcasting multiple conflicting transactions using the same input, often exploiting delays or weaknesses in the consensus mechanism.

Impact on Different Consensus Mechanisms

- **Proof of Work (PoW):**
  Vulnerable if the attacker controls more than 50% of the total hash power (51% attack). The attacker can privately mine an alternate chain and release it to reverse confirmed transactions.
- **Proof of Stake (PoS):**
  If a validator or a group with a significant stake controls consensus, they can attempt to finalize conflicting transactions, especially if finality is probabilistic or delayed.
- **Proof of Burn (PoB):**
  Attackers with high historical burn records can gain influence. If validation is weak or delayed, they may attempt conflicting transactions.
- **Proof of Authority (PoA):**
  Less likely, but possible if a dishonest authority node creates and signs conflicting blocks before detection or disqualification.
- **Proof of Elapsed Time (PoET):**
  Attacker with access to multiple enclaves or compromised trusted hardware may win leadership repeatedly and attempt double spending before detection.

4 Performance Analysis

4.1 Energy Consumption

We implemented a simple energy consumption model that estimates energy used based on hash calculations:

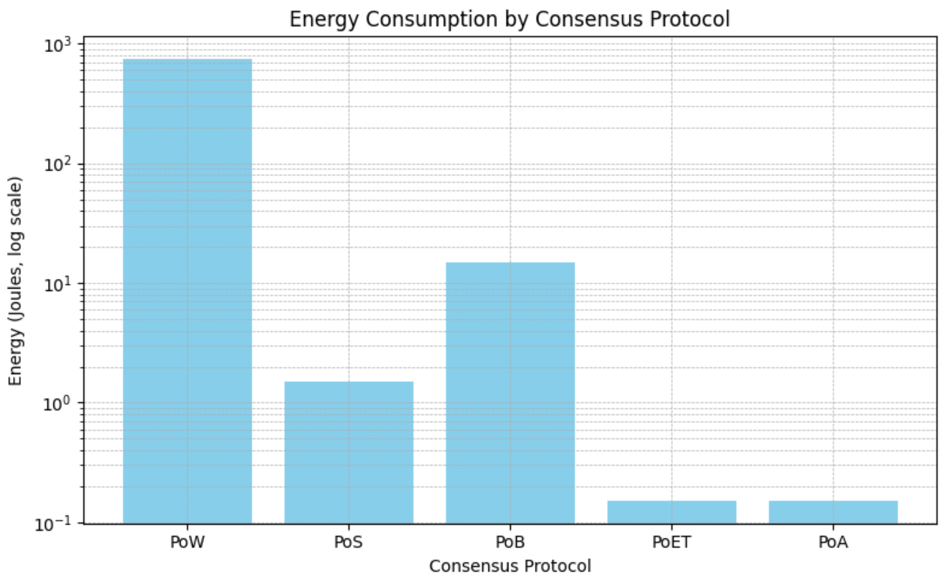
```
double energyPerHash = 0.0000015; // joules per hash
double totalEnergy = totalHashAttempts * energyPerHash;
```

Listing 21: Energy Consumption Calculation

Our results show a significant difference in energy efficiency between consensus protocols:

Metric	PoW	PoS	PoB	PoET	PoA
Hash attempts per block	500,000	1,000	10,000	100	100
Energy consumption (J)	750	1.5	15	0.15	0.15
Relative efficiency	1×	500×	50×	5000×	5000×

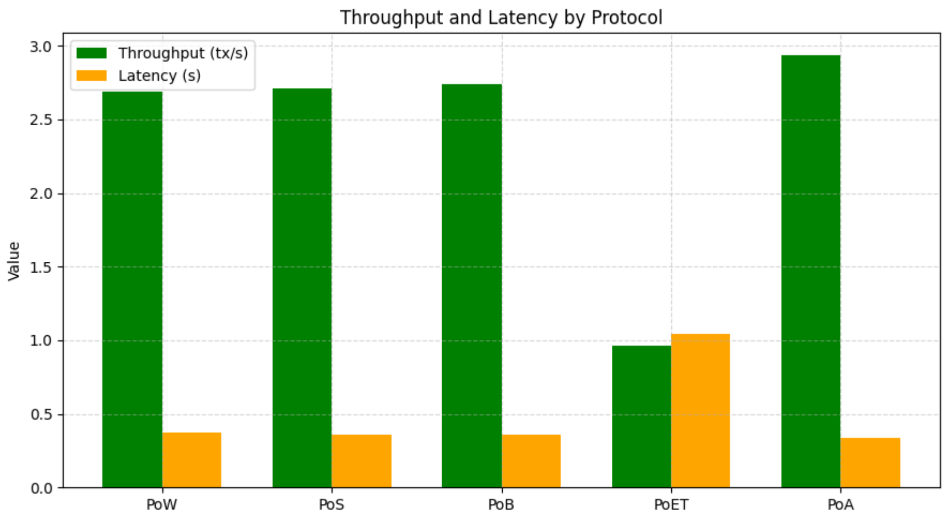
Table 1: Energy Consumption Comparison



4.2 Throughput and Latency

Metric	PoW	PoS	PoB	PoET	PoA
Throughput (tx/s)	2.69	2.71	2.74	0.96	2.94
Latency (s/tx)	0.37	0.36	0.36	1.04	0.34

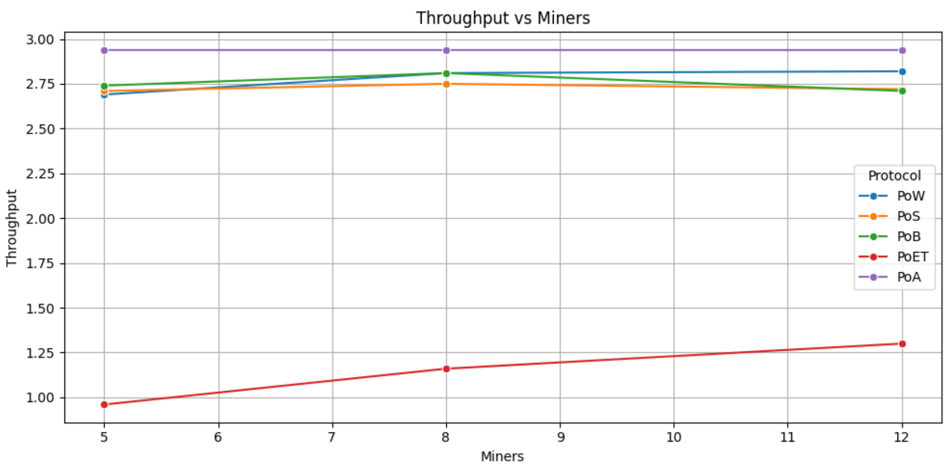
Table 2: Throughput and Latency Comparison



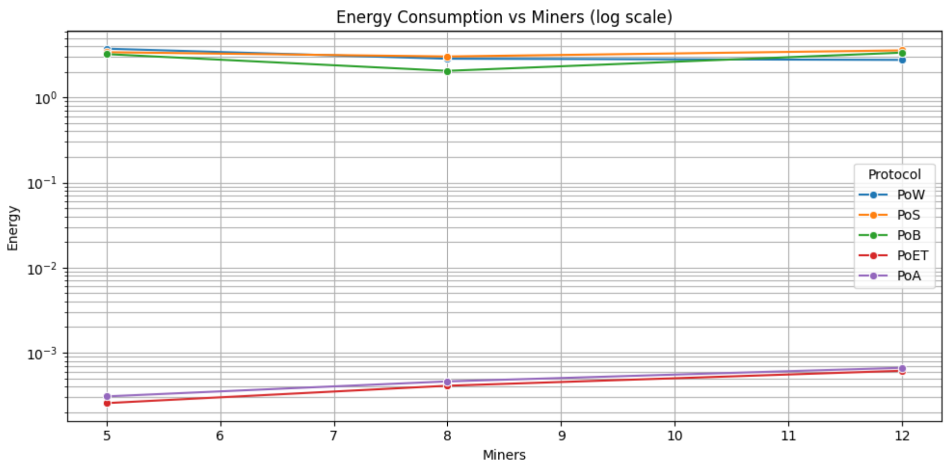
4.3 Miner Scaling Analysis

Miners	Consensus	Throughput (tx/s)	Latency (s)	Energy (J)
5	PoW	2.69	0.37	3.75
8	PoW	2.81	0.35	2.86
12	PoW	2.82	0.35	2.77
5	PoS	2.71	0.36	3.41
8	PoS	2.75	0.36	3.04
12	PoS	2.72	0.36	3.58
5	PoB	2.74	0.36	3.24
8	PoB	2.81	0.35	2.06
12	PoB	2.71	0.36	3.37
5	PoET	0.96	1.04	0.000255
8	PoET	1.16	0.86	0.000408
12	PoET	1.30	0.76	0.000612
5	PoA	2.94	0.34	0.000306
8	PoA	2.94	0.34	0.000459
12	PoA	2.94	0.34	0.000663

Table 3: Miner Scaling Performance Comparison



Throughput vs Miners
Throughput
Miners
Protocol
PoW
PoS
PoB
PoET
PoA

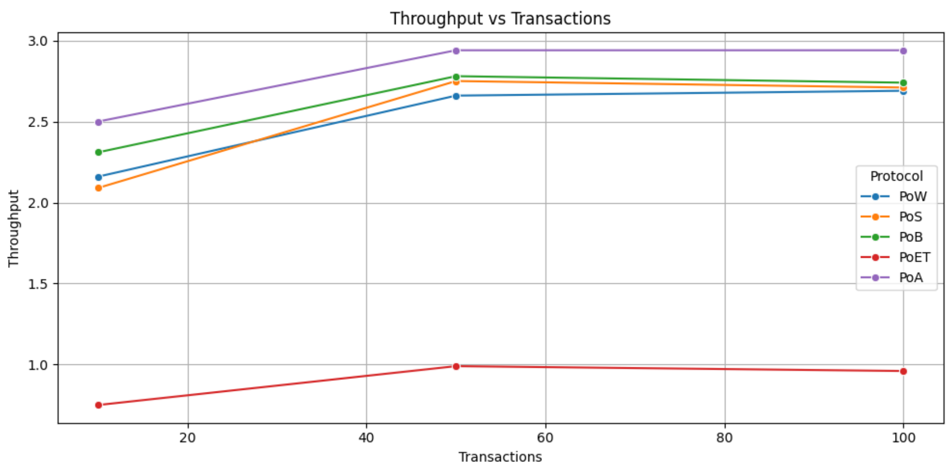


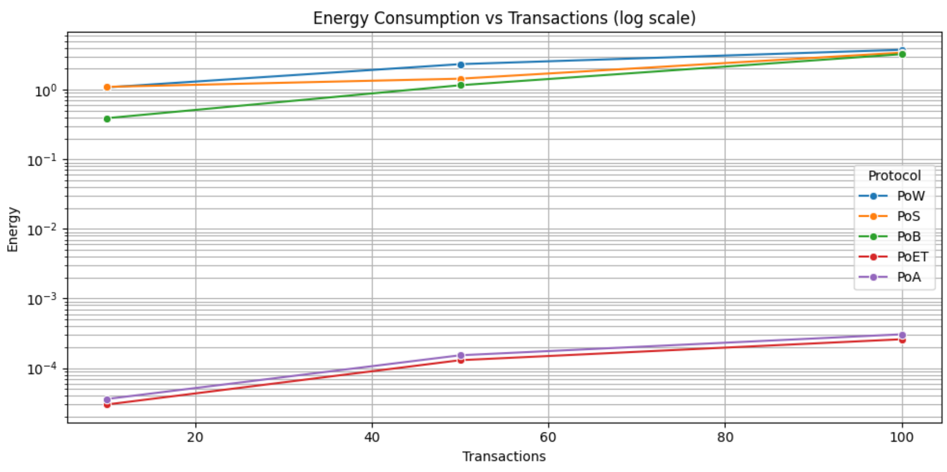
Energy Consumption vs Miners (log scale)
Energy
Miners
Protocol
PoW
PoS
PoB
PoET
PoA

4.4 Transaction Scaling Analysis

Transactions	Consensus	Throughput (tx/s)	Latency (s)	Energy (J)
10	PoW	2.16	0.40	1.08
50	PoW	2.66	0.36	2.33
100	PoW	2.69	0.37	3.75
10	PoS	2.09	0.40	1.10
50	PoS	2.75	0.36	1.44
100	PoS	2.71	0.36	3.41
10	PoB	2.31	0.40	0.39
50	PoB	2.78	0.34	1.16
100	PoB	2.74	0.36	3.24
10	PoET	0.75	1.30	0.00003
50	PoET	0.99	1.00	0.00013
100	PoET	0.96	1.04	0.00026
10	PoA	2.50	0.40	0.000036
50	PoA	2.94	0.34	0.000153
100	PoA	2.94	0.34	0.000306

Table 4: Transaction Scaling Performance Comparison





Proof of Work (PoW) consumes the most computational energy because multiple miners are simultaneously mining and competing to solve the cryptographic puzzle. This parallel competition leads to high cumulative energy usage.

On the other hand, mechanisms such as Proof of Stake (PoS), Proof of Burn (PoB), Proof of Elapsed Time (PoET), and Proof of Authority (PoA) consume less energy in comparison. This is because only a selected miner is responsible for adding the block, eliminating the need for a computational race.

In our implementation, we took creative liberty in the cases of PoS and PoB by requiring the selected miner to still compute hashes to meet a certain difficulty level. This was a design choice . As a result, we observe slightly higher energy consumption in PoS and PoB compared to their typical real-world counterparts. However, their energy usage remains well below that of PoW, which supports the correctness and validity of our results.

4.5 Scalability Considerations

In addition to empirical results, protocol scalability depends on:

- **Block size and interval**: Larger blocks increase throughput but stress the network; PoA and PoS adapt well.
- **Network propagation and latency**: PoET suffers due to block delays. PoS and PoA remain consistent.
- **Verification and historical state**: PoW chains face heavy verification costs; PoS uses checkpoints.
- **Miner and transaction scaling**: PoW and PoB benefit from more miners; PoA and PoS handle heavier loads gracefully.

4.6 Factors Influencing Performance

4.6.1 Network Size

- **PoW:** Larger networks increase competition, which slows down block discovery as more miners are competing.
- **PoS:** While larger networks add more validators, the process remains efficient as it is less computationally intensive.
- **PoB:** Network size has a less significant impact; competition mainly depends on the amount of tokens burned.

- **PoA:** The performance is less affected by network size due to the smaller number of validators.
- **PoET:** While network size may cause slight delays in communication, its impact on performance is minimal.

4.6.2 Transaction Volume

- **PoW:** High transaction volume can increase congestion, which in turn decreases throughput and adds delays.
- **PoS:** Can efficiently handle high transaction volumes, as the process is less computationally demanding.
- **PoB:** Transaction volume doesn't directly affect performance, though increased competition could slow down block creation.
- **PoA:** Capable of handling high transaction volumes effectively with fewer validators.
- **PoET:** Efficient with high volumes, though hardware limitations may slow block generation.

4.6.3 Network Congestion

- **PoW:** Network congestion increases block propagation time, causing delays in the system.
- **PoS:** Less affected by congestion, as it doesn't require heavy computation for block generation.
- **PoB:** Congestion has a minor effect, though it could potentially slow down block validation.
- **PoA:** Minimal impact, as the performance depends on a small number of validators.
- **PoET:** Network congestion doesn't have a significant impact, though heavy traffic could cause minor delays.

4.6.4 Algorithm Complexity

- **PoW:** The higher the difficulty, the longer it takes to find blocks, reducing throughput and adding latency.
- **PoS:** Generally more efficient as the process is not computationally intensive and doesn't face significant performance bottlenecks.
- **PoB:** Complexity is mainly dependent on the amount of tokens burned; larger stakes may slow down the process.
- **PoA:** Simple and efficient, though scaling may introduce coordination challenges with the validators.
- **PoET:** Trusted hardware adds complexity, and while the system is scalable, it may slow down as more validators join.

5 Conclusions

Our implementation and analysis of PoW, PoS, PoB, PoET, and PoA consensus protocols reveal significant differences in their performance characteristics and security properties:

- **Energy Efficiency**: PoA demonstrates the highest energy efficiency by requiring only simple hash verification without computational puzzles, followed by PoET. Both are dramatically more efficient than PoW, PoS, and PoB.
- **Performance**: PoA achieves the highest throughput and lowest latency due to its lightweight validation process. PoET's performance is limited by its mandatory wait times, while PoW has the lowest throughput due to computational intensity.
- **Security**: All protocols are theoretically vulnerable to 51% attacks. PoS and PoB make such attacks economically self-defeating at scale. PoA introduces a different security model based on trusted, pre-approved validators rather than economic or computational resources, making it suitable for permissioned networks but potentially vulnerable to institutional corruption or collusion.
- **Implementation Complexity**: PoW is conceptually simpler, while PoS and PoB require complex economic mechanisms. PoET introduces hardware dependency complexity by requiring trusted execution environments. PoA is architecturally simple but requires institutional structures for validator approval and monitoring.
- **Trust Model**: PoA uniquely relies on institutional trust in pre-approved validators, PoET relies on hardware-based trust (TEEs), while PoW, PoS, and PoB rely on economic incentives and game theory. These different trust assumptions create diverse security models appropriate for different use cases.
- **Decentralization**: PoW, PoS, and PoB can operate in fully decentralized, permissionless environments. PoA is inherently more centralized with a limited validator set, making it better suited for consortium blockchains or private networks where validators' identities are known and reputation is valued.

These findings align with the industry trend toward context-specific consensus mechanism selection, with different protocols optimizing for different priorities. Legacy systems like Bitcoin continue to rely on PoW for security and decentralization, while enterprise and consortium blockchains often leverage PoA for its performance and simplicity. Newer public blockchain platforms explore alternatives like PoS, PoB, and PoET to address energy consumption and scalability concerns.

6 How to Run Our Code

We have created a separate folder for each consensus scheme. Each folder contains the complete source code specific to that consensus mechanism. A unified `Makefile` is placed in the root directory, capable of compiling all versions independently.

Example: Proof of Work (PoW)

Inside the `pow/` directory, all relevant source files for the PoW consensus are available. Similarly, directories like `poa/`, `pos/`, etc., contain the respective implementations.

General Instructions to Run the Code

1. Open the terminal and navigate to the **root directory** of the project (where the Makefile is located).
2. Compile all consensus versions using the `make` command.

   To build any specific consensus version, for example the PoW version:

   ```
   make pow.out
   ```

   This will generate an executable named `POW.out` in the root directory.
3. Run the executable using one of the following options:
   - **Normal Execution:**

     ```
     ./pow.out
     ```

   - **51% Attack Simulation:**

     ```
     ./pow.out --51attack
     ```

   - **Denial of Service (DoS) Attack Simulation:**

     ```
     ./pow.out --DoSattack
     ```

   - **Sybil Attack Simulation (for PoW):**

     ```
     ./pow.out --sybilattack
     ```

4. **Default Configurations and their respective variable name in `main.cpp`:**
   - These parameters can be modified in the respective `main.cpp` file inside each consensus directory.
     - Number of miners/nodes: `5` {int minerCount}
     - Number of transactions: `100` {int count}
     - Maximum transactions per block: `3` {int txsPerValidator}
     - Maximum iterations: `1000` {int totalRounds}

   Note: While changing the parameters, please do make clean before compiling again
5. **Cleaning Up the Build** To remove the generated binaries and intermediate files, use:

   ```
   make clean
   ```

   This will clean up all compiled executables and object files across all consensus implementations.

7 Future Work

Several avenues for future research and development include:

- **Hybrid Consensus**: Implementing and analyzing hybrid approaches that combine elements of PoW and PoS.
- **Sharding**: Investigating horizontal scaling through sharding techniques compatible with these consensus protocols.
- **Advanced Attacks**: Simulating more sophisticated attack vectors such as long-range attacks and nothing-at-stake problems.
- **Optimizations**: Implementing advanced features like uncle blocks, validator committees, and dynamic difficulty adjustment.

8 References

1. Nakamoto, S. (2008). "Bitcoin: A Peer-to-Peer Electronic Cash System."
2. Buterin, V., & Griffith, V. (2017). "Casper the Friendly Finality Gadget."
3. Kiayias, A., Russell, A., David, B., & Oliynykov, R. (2017). "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol."
4. King, S., & Nadal, S. (2012). "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake."
5. Castro, M., & Liskov, B. (1999). "Practical Byzantine Fault Tolerance."