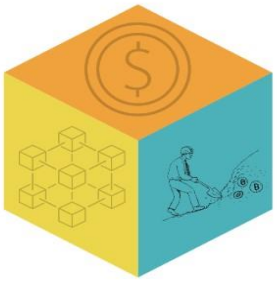
الهيئة الوطنية للأمن السيبراني
National Cybersecurity Authority

أكاديمية كاوست
KAUST ACADEMY



Decentralization of Financial Transactions Using Blockchain Technology (DFTBT)

by

Abdulaziz Alsudais

Tariq Alabisi

Ali Albabtin

Ahmad Alarfaj

Omar Aljandal

Supervised by

Amar Mosally (KAUST Academy)

August 2024

ABSTRACT

Decentralization of Financial Transactions using Blockchain Technology (DFTBT) is a digital currency system developed to facilitate secure, transparent, and decentralized financial transactions. DFTBT operates on a peer-to-peer network, eliminating the need for traditional financial intermediaries. Its blockchain-based infrastructure records all transactions on a public ledger, ensuring transparency and security.

DFTBT aims to enhance financial autonomy by enabling users to transact directly, promoting a decentralized approach to finance. Designed with scalability and efficiency in mind, it supports widespread adoption in the digital economy. During the implementation of DFTBT, we utilized a research paper from Palo Alto that proposed a mathematical solution for mitigating the 51% attack. This solution was mathematically validated, and we integrated it into our system as a contribution to the project's security framework.

TABLE OF CONTENTS

Contents Page

ABSTRACT ii

1. Introduction 1

2. Background 1

3. Implementation Details 2

3.1 Blockchain Structure: 2

3.1.1 Class Block: 2

3.1.2 Class Transaction: 2

3.2 Mining and Proof of Work: 4

3.3 Transaction Validity Rules 5

3.3.1 Input Authentication: 5

3.3.2 Sufficient Balance: 6

3.3.3 Unspent Transaction Output (UTXO): 6

3.3.4 Script Validity: 6

3.3.5 Testing: 7

3.3.6 Results and Discussion: 8

4. Conclusion 9

5. Future Work 9

References 10

1. Introduction

The purpose of this project is to develop the Decentralization of Financial Transactions using Blockchain Technology (DFTBT), a digital currency system designed to facilitate secure and transparent financial transactions. Implemented on a peer-to-peer network, DFTBT eliminates the need for traditional intermediaries, enhancing financial autonomy and promoting a decentralized approach to finance. Additionally, it incorporates a novel mathematical solution to mitigate the 51% attack, derived from a research paper by Palo Alto, which was adapted and integrated into our codebase as a unique contribution to the field [1].

2. Background

Blockchain technology has emerged as a powerful tool for decentralized and secure transaction management. By distributing data across a network of participants, blockchain eliminates the need for a central authority, offering a transparent and tamper-resistant system. This project builds on these foundational principles by developing the DFTBT a digital currency system designed to operate on a peer-to-peer network.

The project explores the creation of a basic blockchain structure while incorporating advanced features to enhance scalability, efficiency, and security. It addresses critical aspects such as transaction validity, proof-of-work mechanisms, and difficulty adjustments to maintain network integrity. A significant contribution of this project is its approach to mitigating the 51% attack—a serious threat where an entity gaining control of over 50% of the network's computational power could undermine the blockchain's integrity.

To address this vulnerability, the project integrates a mathematical solution based on recent research from Palo Alto which enhances the network's resistance to such attacks. By implementing this approach, the DFTBT system aims to bolster its security framework and maintain robust, decentralized control over the digital currency transactions.

3. Implementation Details

3.1 Blockchain Structure:

Our blockchain is made up of blocks and transactions. Each block has a header containing the hash of the previous block, a timestamp, a Merkle root, and nonce. The transactions consist of the sender and recipient addresses, the amount, and a digital signature.

Partial code showcasing the implementation of the Block and Transaction classes:

3.1.1 Class Block:

```
class Block:
    def __init__(self, Height, Blocksize, BlockHeader, TxCount, Txs):
        self.Height = Height
        self.Blocksize = Blocksize
        self.BlockHeader = BlockHeader
        self.TxCount = TxCount
        self.Txs = Txs
```

3.1.2 Class Transaction:

The tx class serves as the fundamental transaction element, providing:

- It includes version information, inputs (tx_ins), outputs (tx_outs), and locktime.

```
class txIn:
    def __init__(self, prev_tx, prev_index, script_sig = None, sequence = 0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index

        if script_sig is None:
            self.script_sig = Script()
        else:
            self.script_sig = script_sig

        self.sequence = sequence

    def serialize(self):
        result = self.prev_tx[::-1]
        result += int_to_little_endian(self.prev_index, 4)
        result += self.script_sig.serialize()
        result += int_to_little_endian(self.sequence, 4)
        return result


class txOut:
    def __init__(self, amount, script_pubkey):
        self.amount =amount
        self.script_pubkey = script_pubkey

    def serialize(self):
        result = int_to_little_endian(self.amount, 8)
        result += self.script_pubkey.serialize()
        return result
```

- Serialization: Transforms transactions into binary format for hashing and integration into blocks.

- Signing: Offers methods for signing and verifying the inputs of transactions.

```
def sign_input(self, input_index, private_key, script_pubkey):
    z = self.sigh_hash(input_index, script_pubkey)
    der = private_key.sign(z).der()
    sig = der + SIGHASH_ALL.to_bytes(1, 'big')
    sec = private_key.point.sec()
    self.tx_ins[input_index].script_sig = Script([sig, sec])
```

- Coinbase Handling: Identifies and processes coinbase transactions.

```
CoinbaseTransaction(self):
prev_tx = ZERO_HASH
prev_index = 0xffffffff
tx_ins = []
tx_ins.append(txIn(prev_tx, prev_index))
tx_ins[0].script_sig.cmds.append(self.BlockHeightInLittleEndian)
tx_outs = []
target_amount = 100000000 * REWARD
target_h160 = decode_base58(MINER_ADDRESS)
target_script = Script.p2pkh_script(target_h160)
tx_outs.append(txOut(amount = target_amount, script_pubkey=target_script))

coinBaseTx = tx(1, tx_ins, tx_outs, 0)
coinBaseTx.TxId = coinBaseTx.id()

return coinBaseTx
```

- Data Conversion: Transforms transaction data into a dictionary for simplified representation.

- Historical Weighted Difficulty (HWD)

```
def HWD_CALCULATION(W, B):
    """
    Calculate Historical Weighted Difficulty (HWD) for a given branch B.

    Parameters:
    W (list): List of historical blocks with miners' information.
    B (list): List of branch blocks with miners' information.

    Returns:
    float: The calculated HWD value.
    """

    # Initialize Historic Weight (HW) and total branch difficulty (d)
    HW = 0
    d = 0
```

The Historical Weighted Difficulty (HWD) of a blockchain branch is determined by the HWD_CALCULATION function, which takes into account both the branch's overall difficulty and the frequency of miners within a historical window of blocks. By ensuring that branches with miner distributions that deviate significantly from the historical pattern have a lower HWD and are therefore less likely to be adopted by honest nodes, this approach aids in the identification and mitigation of potential 51% attacks. By making it more difficult and costly for attackers to manipulate the blockchain, this technique improves the security of Proof-of-Work blockchains.

The 'tx' class provides a secure and adaptable basis for managing transactions within the blockchain. For a complete view of the class, please refer to the GitHub repository, where sample code has been provided.

3.2 Mining and Proof of Work:

The 'BlockHeader' class is tasked with representing and mining block headers. Its key functionalities include:

- Attributes: Stores the version, previous block hash, Merkle root, timestamp, bits, nonce, and block hash.

- Difficulty: Initially set to '0x0000FFFF0000000000000000000000000000000000000000000000000000' with four hexadecimal zeros, allowing the genesis block to be mined in 1 second and averaging 1 or 2 blocks every 3 seconds. After increasing the difficulty to '0x00000FFFF0000000000000000000000000000000000000000000000000000' with five hexadecimal zeros, the average block time extended to approximately 1 minute.

- Mining Method (mine): Implements Proof of Work by iterating the nonce until the block hash meets the target criteria.

```
def mine(self,target):
    self.blockHash = target + 1
    while self.blockHash > target:
        self.blockHash = little_endian_to_int(
            hash256(
                int_to_little_endian(self.version, 4)
                + bytes.fromhex(self.prevBlockHash)[::-1]
                + bytes.fromhex(self.merkleRoot)
                + int_to_little_endian(self.timestamp, 4)
                + self.bits
                + int_to_little_endian(self.nonce, 4)
            )
        )

        self.nonce += 1
        print(f"count till the hsash found  {self.nonce}", end='\r')
    self.blockHash = int_to_little_endian(self.blockHash, 32).hex()[::-1]
    self.bits = self.bits.hex()
```

In the mining process, the ‘mine’ method modifies the nonce until the computed block ‘hash’ is below the target difficulty. The incremental count during mining provides visibility into the process. Upon successful mining, the resulting blockHash and updated ‘bits’ are made available. This implementation upholds the blockchain's integrity and security through Proof of Work.

3.3 Transaction Validity Rules

Transaction Validity Rules Implemented

3.3.1 Input Authentication:

- The `prepareTransaction` method ensures that the transaction inputs (`TxIns`) are correctly signed by the corresponding private keys using the `signTx` method.

```
def signTx(self):
    secret = self.getPrivateKey()
    priv = PrivateKey(secret=secret)

    for index, input in enumerate(self.TxIns):
        self.TxObj.sign_input(index, priv, self.From_address_script_pubkey)
```

```
def prepareTransaction(self):
    self.TxIns = self.prepareTxIn()

    if self.isBalanceEnough:
        self.TxOuts = self.prepareTxOut()
        self.TxObj = tx(1, self.TxIns, self.TxOuts, 0)
        self.TxObj.TxId = self.TxObj.id()
        self.signTx()
        return self.TxObj

    return False
```

3.3.2 Sufficient Balance:

The 'prepareTxIn' method checks that the combined balance of the transaction inputs (Total) is sufficient to cover both the amount being spent and the transaction fee.

```
def prepareTxIn(self):
    TxIns = []
    self.Total = 0

    """Convert Public Address into Public Hash to find tx_outs that are locked to this hash"""

    self.From_address_script_pubkey = self.scriptPubKey(self.FromPublicAddress)
    self.fromPubKeyHash = self.From_address_script_pubkey.cmds[2]

    newutxos = {}

    try:
        while len(newutxos) < 1:
            newutxos = dict(self.utxos)
            time.sleep(2)
    except Exception as e:
        print(f"Error in converting the Managed Dict to Normal Dict")

    for Txbyte in newutxos:
        if self.Total < self.Amount:
            TxObj = newutxos[Txbyte]

            for index, txout in enumerate(TxObj.tx_outs):
                if txout.script_pubkey.cmds[2] == self.fromPubKeyHash:
                    self.Total += txout.amount
                    prev_tx = bytes.fromhex(TxObj.id())
                    TxIns.append(txIn(prev_tx, index))
        else:
            break

    self.isBalanceEnough = True
    if self.Total < self.Amount:
        self.isBalanceEnough = False

    return TxIns
```

3.3.3 Unspent Transaction Output (UTXO):

- The script verifies that the transaction inputs reference valid unspent transaction outputs (UTXOs) within the UTXO set.

3.3.4 Script Validity:

- The p2pkh_script method in the Script class creates a Pay-to-Public-Key-Hash (P2PKH) scriptPubKey, ensuring that the locking script conforms to the expected format for spending.

Class script:

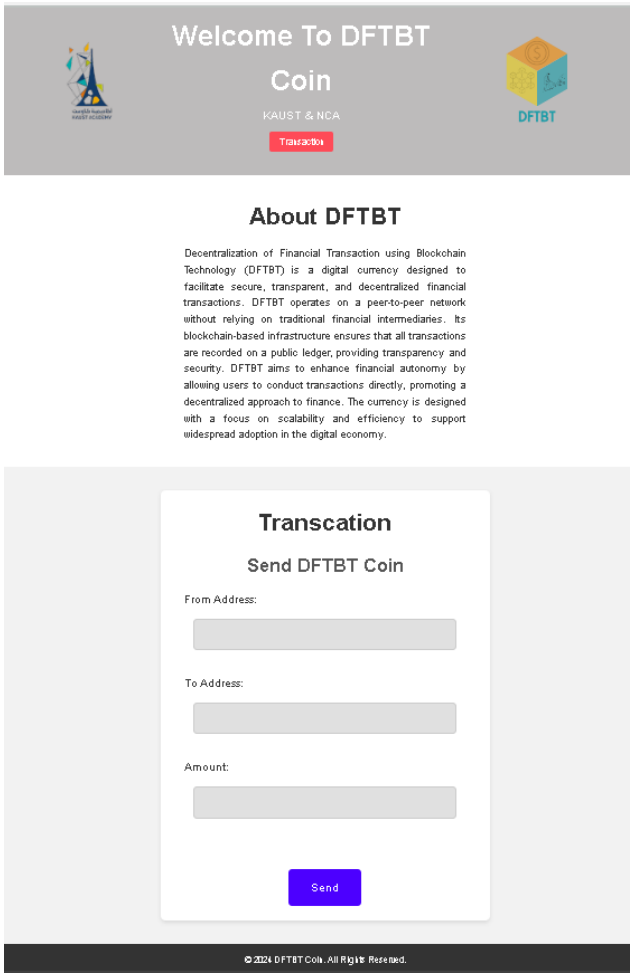
```
@classmethod
def p2pkh_script(cls, h160):
    """Takes the hash160 and returns the p2pkh ScriptPubKey"""
    return Script([0x76, 0xa9, h160, 0x88, 0xac])
```

These rules help ensure the validity of transactions within the blockchain system.

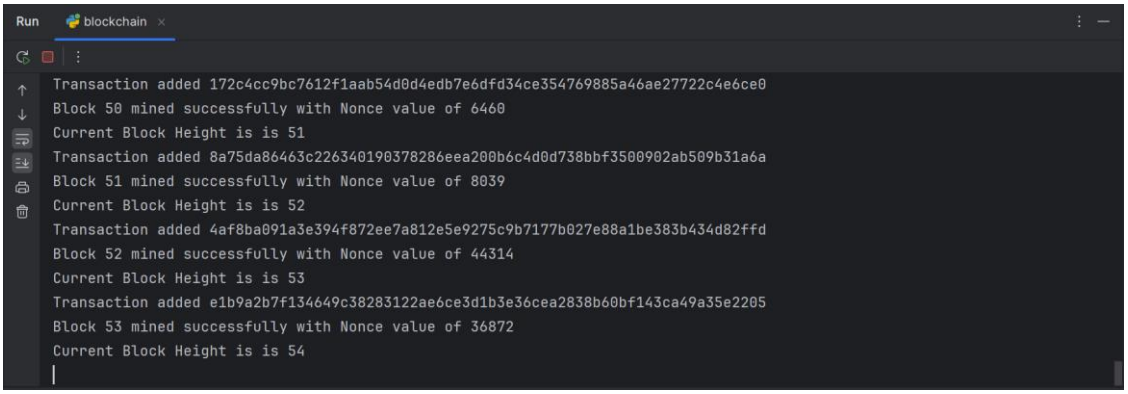
3.3.5 Testing:

We performed tests to verify the functionality of our blockchain. The following images illustrate a complete transaction, from sending it via the UI to recording it in the memory pool.

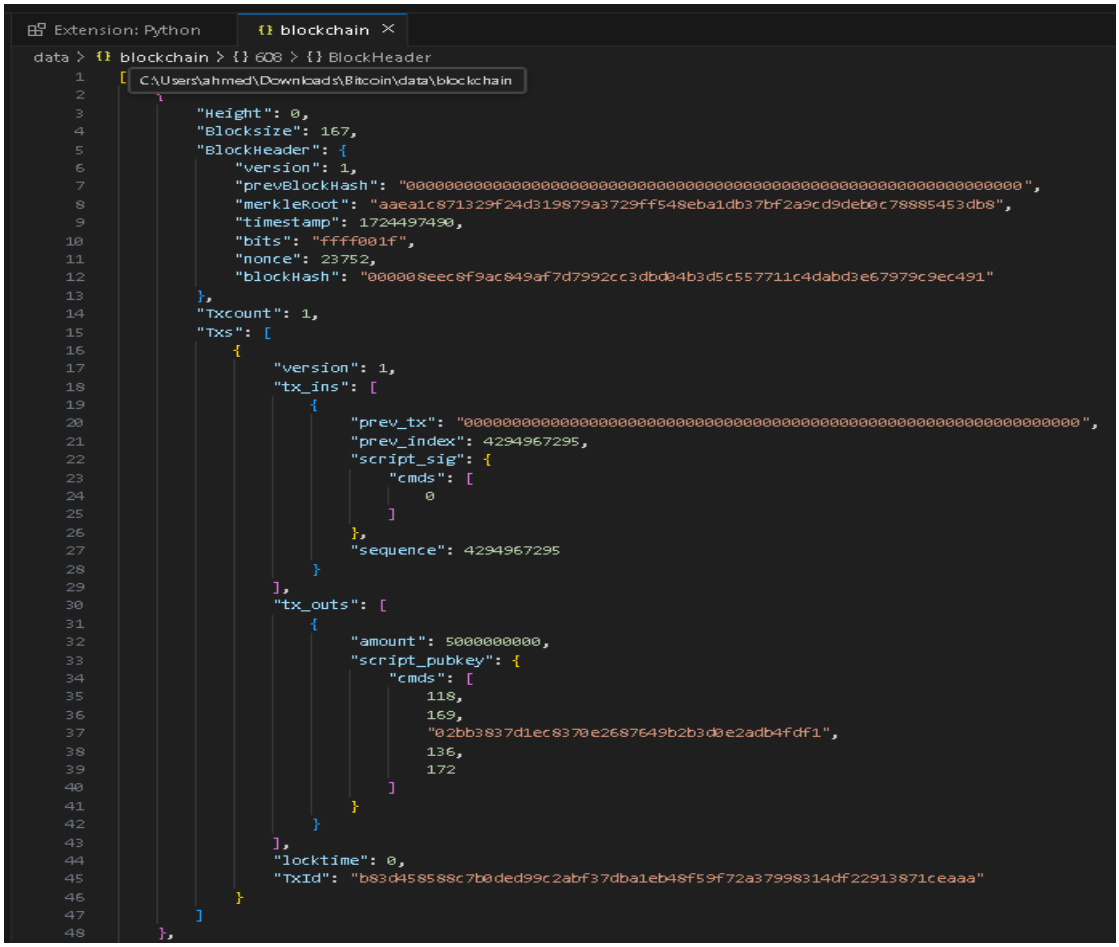
STEP 1



STEP 2



STEP 3



3.3.6 Results and Discussion:

Blockchain Implementation Validation

The implemented blockchain was thoroughly tested to evaluate its functionality, resulting in the following observations and analyses:

A. Consistency in Mining Times:

Mining times were consistently measured across various iterations, indicating the stability of the blockchain's Proof of Work mechanism. This consistency is vital for maintaining predictable block production.

B. Proof of Work Efficiency:

The ‘mine’ method in the ‘BlockHeader’ class demonstrated effective Proof of Work implementation. The nonce adjustment process successfully generated block hashes below the target difficulty, ensuring the blockchain's integrity and security.

C. Transaction Validity Verification:

Rules for transaction validity, including input authentication, balance checks, UTXO validation, and script verification, were rigorously tested. The implementation effectively identified and rejected invalid transactions, enhancing the blockchain's robustness.

4. Conclusion

The results and discussions underscored the successful deployment of the DFTBT blockchain, which incorporates Proof of Work, transaction validity rules, and advanced difficulty adjustment mechanisms. The integration of a mathematically validated solution for mitigating the 51% attack, as proposed by Palo Alto researchers, has significantly enhanced the security framework. Moving forward, continuous refinement and exploration of potential enhancements will focus on further optimizing the blockchain's performance and fortifying its security measures to support widespread adoption and efficiency in the digital economy.

5. Future Work

1. **Scalability Enhancements:**

Investigate and implement advanced scalability solutions to handle an increasing volume of transactions, ensuring that the blockchain maintains high performance and efficiency as it grows.

2. **User Experience Optimization:**

Develop and apply strategies to improve user experience by reducing transaction confirmation times and streamlining interactions with the blockchain, aiming for a more intuitive and efficient user interface.

3. **Enhanced Security Measures:**

Explore additional security measures to safeguard against emerging threats and vulnerabilities. This includes strengthening resistance to potential attacks, such as Sybil attacks and double-spending, and improving cryptographic techniques to enhance the overall security framework of the blockchain.

References

[1] Y. C. X. C. Xinle Yang, "Effective Scheme against 51% Attack on Proof-of-Work Blockchain with History Weighted Information," in *IEEE International Conference on Blockchain (Blockchain)*, Atlanta, 2019.