Creating A Fake Cryptocurrency Unit

Sai Venkatesh Pabba
spabba@stcloudstate.edu

Creating a Fake Cryptocurrency Unit

by

Sai Venkatesh Pabba

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Information Assurance

December, 2018

Starred Paper Committee
Guster Dennis, Chairperson
Lynn Collen
Balasubramanian Kasi

Abstract

In the recent years, cryptocurrencies gained lots of popularity. Many new cryptocurrencies are introduced day by day. Though new cryptocurrencies are being introduced, they are based on the same Blockchain technology. Cryptocurrencies are virtual currencies and differ from traditional money in a way which made them very popular among the users. Bitcoin which was the first cryptocurrency introduced by Satoshi Nakamoto in late 2008 as a Peer-to-Peer Electronic cash system. The most important feature of this system was that it was de-centralized meaning that there is no centralized authority controlling the payment network. Instead, every single entity of the network realizes all the tasks of the centralized server. Cryptocurrencies rely on miners who verify the transactions and add the block to the blockchain. Miners depend on high computation power to solve a mathematical problem following a mining algorithm which also rewards them with some cryptocurrency.

This paper provides a comprehensive overview of the technology behind cryptocurrencies and explores the security and privacy issues that are involved with cryptocurrencies and introduces a mechanism to create fake cryptocurrency units.

Keywords: cryptocurrency, Bitcoin, miner, blockchain

Table of Contents

Page

List of Tables

Table Page

List of Figures

Figure Page

Chapter I: Introduction

Introduction

Since the creation of Bitcoin in 2009, many cryptocurrencies have been introduced. These cryptocurrencies have attracted many investors. Bitcoin is the most popular cryptocurrency saw a peak conversion rate of $19,783.21 on December 17, 2017 (Higgins, 2017). This surge in the price of bitcoin was due to growing popularity among the users and miners. Due to this increased popularity of cryptocurrencies, many financial institutions began accepting cryptocurrencies as a form of payment. The first bitcoin ATM was installed in a coffee shop in Canada which allows users to convert bitcoins to Canadian dollars and vice-versa (Who is accepting bitcoin). Overstock.com had started accepting bitcoin for payments since 2014, and since then many other websites like WordPress, Reddit, Namecheap, etc. started accepting bitcoins as payment (Who is accepting bitcoin).

A cryptocurrency is a peer-to-peer digital exchange system in which cryptography is used to generate and distribute currency units (Farell, 2015). The main essence of cryptocurrencies is in the concept of Blockchain technology. Blockchain technology involves processing a transaction without no centralized authority. In usual monetary systems, there will be a central authority (usually a bank) who overlooks the entire system. This authority is responsible for maintaining verify, validate, and process the transactions, log them, and lets the user know about the status of the transaction. On the other hand, decentralized monetary systems have no such centralized system to handle and monitor the transactions (Nakamoto, 2008). The main concept behind this decentralized currency is the anonymity of the transaction. These kinds of systems are realized through Blockchain technology.

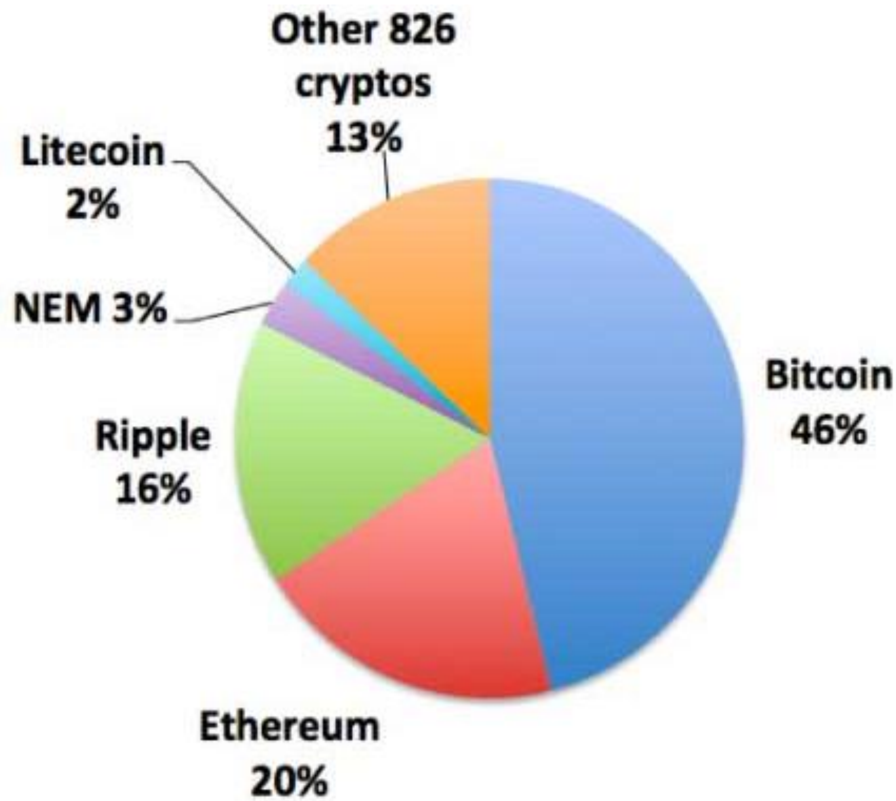


Figure 1: Shares of cryptocurrencies (source: https://www.ofwealth.com)

The above figure shows the shares of different cryptocurrencies in the market. Bitcoin takes the highest share of the market with 46% of the whole share which is followed by Ethereum with a 20% share followed by Ripple with a 16% share. NEM holds a share of 3%, Litecoin holds 2% of the share, and the rest hold about 13% altogether.

Problem Statement

Cryptocurrency units usually have a value that is higher than most traditional currency units. For example, a single bitcoin costs around 8000 US dollars. Given the conversion rates, fake cryptocurrency units can cause a huge amount of loss to the economy.

This research will help explore the attacks that involve creating an illegitimate cryptocurrency unit by analyzing the algorithms involved in the cryptocurrency, find the flaws in them to implement and prove that a fake cryptocurrency unit can be created.

Nature and Significance of the Problem

There is a rapid increase in the market cap of cryptocurrencies in recent years. Figure 2 shows the market cap of cryptocurrencies from April 2017 to April 2018.

Figure 2: Market cap of cryptocurrencies. (source: https://coinmarketcap.com/charts/)

The figure above shows the market cap of cryptocurrencies. The market is almost about 750 billion US Dollars. Considering the amount of money involved in these cryptocurrencies, adding a fake cryptocurrency unit to the system creates a huge loss to legitimate users of the cryptocurrency. If users come to know that a certain cryptocurrency unit can be duplicated, they might no longer want to hold their crypto coins and might decide to sell them. Lack of users will lead to a huge downfall in the prices of these cryptocurrency coins thereby pushing legit users and investors into a huge loss.

The Objective of the Research

The primary objective of this research is to observe various cryptocurrencies to find out the vulnerabilities in them and determine how they would react to an attack trying to duplicate cryptocurrency units. Finally, come up with a mechanism to create a duplicate crypto coin using the details found from the research.

Research Questions

How secure are the cryptocurrency networks?

What are the different algorithms that are being used in cryptocurrency systems?

What kind of security mechanisms are in place to avoid attacks on cryptocurrency networks?

Are there any known vulnerabilities in the mechanisms that are being used to implement the cryptocurrency system?

How does the network respond to attacks on any of its nodes?

If an attack is made on a node, how does the network handle the situation to solve the problem?

What happens if a node is acting maliciously?

Limitations of the Study

Cryptocurrency networks are usually very huge and involve some thousands of nodes forming the backbones of the network. These nodes usually have high computational capabilities. So, it is not feasible for an individual to overpower the computing power of these nodes put together. Also attacking these networks would be illegal and might incur a loss to some user to the cryptocurrency network.

Definition of Terms

A cryptocurrency is a peer-to-peer digital exchange system in which cryptography is used to generate and distribute currency units (Farell, 2015). A duplicate cryptocurrency unit is something that gets added to blockchain against the cryptocurrency protocol and gets into circulation in the network. It also talks about the final objective.

Summary

This chapter discusses the basics of cryptocurrency starting with what a cryptocurrency is and how important they are by introducing their market cap. It talks about the research work that the paper concentrates on and the hurdles that should be overcome for the research to succeed. The next chapter talks about the previous work that is involved in securing cryptocurrency networks and the working of the cryptocurrency networks in greater detail.

Chapter II: Background and Review of Literature

Introduction

In this chapter, we will be discussing the working of various cryptocurrency networks, mechanisms involved in them and their working. This chapter also talks about the previous literature related to the research area and provides an opinion about those works.

Background Related to the Problem

The blockchain is a database which stores the transactions that were performed using a cryptocurrency. A blockchain creates a sort of digital ledger which constitutes blocks. These blocks contain the data related to the transaction and are linked to each other to form a chain-like structure hence the name blockchain.

Each node in blockchain is identified by a hash value that is computed using a hashing algorithm. Every block in the chain contains the hash value of its previous block thereby linking the blocks together. A single block can have multiple child blocks but can have only one parent block (Singh & Singh, 2017). Each node on the network will maintain a copy of this chain. In case of Bitcoin, on an average, a block gets added to the chain for every 10 minutes (Kaushal, Bagga, & Sobti, 2017). As of 04/05/2018 19:00 the length of blockchain is 163,278MB.

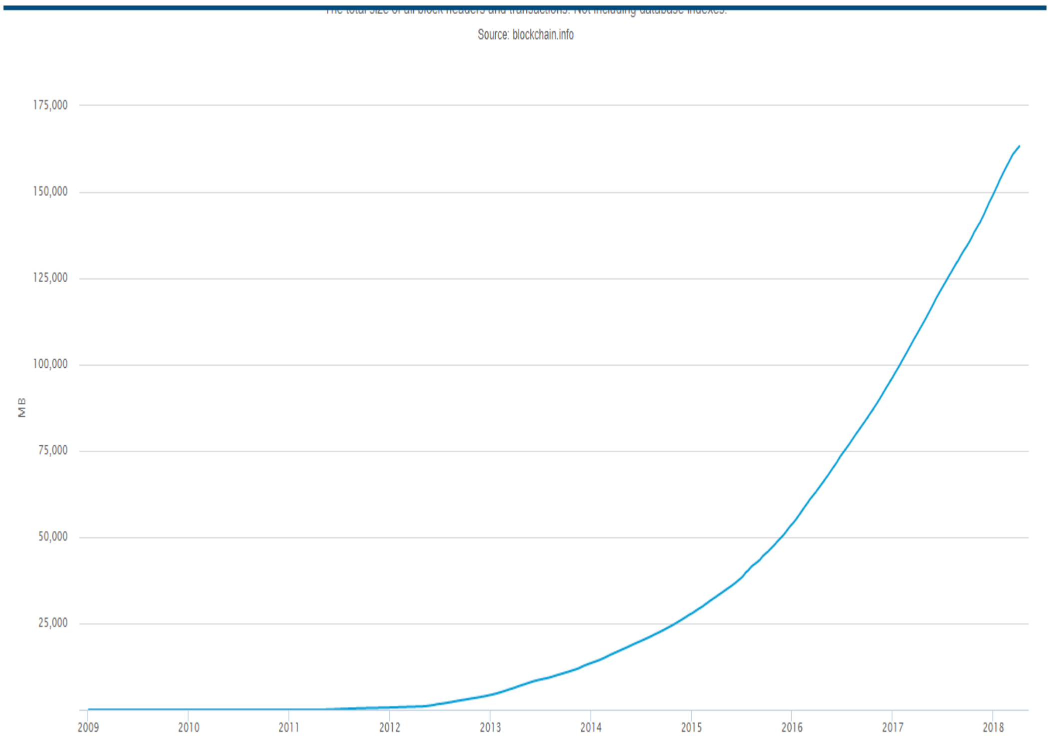


Figure 3: Size of blockchain (Y-axis) vs. time (X-axis) (Source: blockchain.info)

The figure above shows the plot of the size of blockchain in megabytes to time in years starting from 2009. As it can be understood from the figure, the size of blockchain took a sudden surge from mid-2012, and since then it kept growing until it reached the size of approximately 175,000 megabytes. This is a huge amount of data considering that only a minimal amount of data is put in each block along with a header. Hash of the previous block and the hash of the entire data to be stored in the block along with the difficulty level and the create coin transaction for the miner who added the block to the chain.

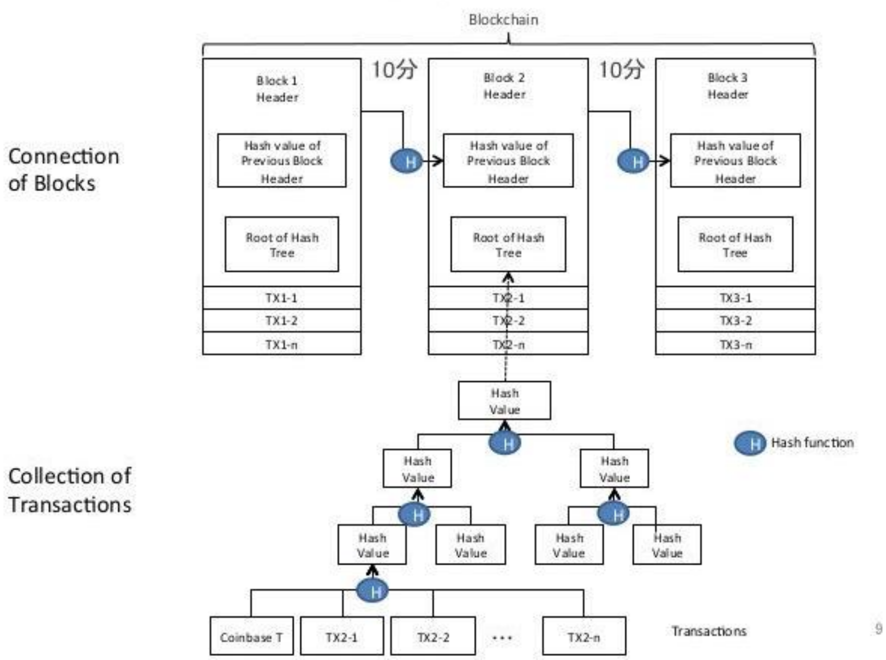


Figure 4: Structure of Blockchain (Source: https://www.pinterest.com)

The above figure shows the structure of part of a blockchain. The very first block in the blockchain is called the Genesis block. Every block in the chain can have a variable number of transactions. Usually, all the transactions that are generated in a 10-minute period are put into a block. This number usually ranges from 1000 to 3000. A unique hash can identify every block in the chain.

Every block contains a header that describes the block, i.e., it contains the metadata about the block, the transactions, Merkle root of the transactions. The hash is generated from all these fields, and the block itself would be added to the chain by a miner.

Table 1: Cryptocurrencies and hashing algorithms (source: https://bitcoinguide.online)

COIN	HASH ALGORITHM
Bitcoin (BTC)	SHA256
Ethereum (ETH)	Sash
Litecoin (LTC)	Scrypt
DigitalCash (DASH)	X11
Monero (XMR)	CryptoNight
Nxt (NXT)	PoS
Ethereum Classic (ETC)	Ethash
Dogecoin (DOGE)	Scrypt
Bitshares (BTS)	SHA-512
DigiByte (DGB)	Multiple
BitcoinDark (BTCD)	SHA256
CraigsCoin (CRAIG)	X11
Bitstake (XBS)	X11
PayCoin (XPY)	SHA256

A wallet may be considered as a piece of software or hardware that holds private keys associated with a cryptocurrency user. These wallets are responsible for storing the private keys of a user securely. It is very important that these private keys remain private because anyone who knows the private key can access all the crypto coins of a user and if a user loses access to his private key, he won't be able to access his crypto coins. We will be discussing this in the next sections.

Every user of a cryptocurrency has a public address associated with it. This address will be used by the blockchain to identify the user. When the user decides to perform a transaction, which may be buying some crypto coins for himself or sending crypto coins to some other user, the transaction will contain the public address of the sender, receiver and the amount that must be sent. This entire transaction is then signed by using the user's private key which will be validated by the nodes in the network by using public keys.

So, as explained earlier, if a user loses his private key, he will lose access to his entire cryptocurrency as he no longer has a private key that is associated with the coins. Once the nodes in the network have validated the transaction, it is put into a block which in turn is added to the blockchain. To calculate a user's balance, all the transactions associated with the user's address are to be collected from the blockchain and then put together to get the final balance.

When a user who has a wallet creates a transaction, this transaction will be verified by the nodes in the network. This process of verifying a transaction and adding it to the blockchain is called mining. Mining is done by miners who turn a huge volume of data into a hash of fixed length. Other miners should then accept this hash in the network. If most of the miners accept this hash, only then it will be added to the blockchain (Singh & Singh, 2017). If a person tries to create a fake transaction, this transaction cannot be validated by other miners and will be rejected. All that it would create is a nuisance in the network and is very easily rectified.

When a miner successfully adds a block to the blockchain, the node gets rewarded with some Bitcoins which are generated through a coin creation transaction with the recipient address as the node's address (Kaushal, Bagga, & Sobti, 2017). In turn for verifying the transactions, the miners get to keep the rewarded coins generated during the mining process. Whenever a user wants to perform a transaction, the user will be charged a transaction fee which is proportional to the amount to of data being added to the blockchain, and this transaction fee goes to the miner who adds the block to the blockchain.

All the cryptocurrencies are based on the decentralized public ledger that is append-only. If any false data gets appended to the chain, it can't be removed from the chain as it is very computationally expensive. Therefore, there must be a mechanism to determine if something that is being added to the blockchain is indeed true. Consensus algorithms provide such mechanism (Glazer, 2014). Double spending and Byzantine Generals Problem are the problems faced by currency systems that can be solved using consensus algorithms.

As the name suggests, Double spending is the problem of spending a unit of currency twice. Physical currency does not have the problem of double spending because it is an entity and it must be exchanged to make a purchase but the same doesn’t apply to internet transactions. Cryptocurrencies solve this problem by verifying the transaction by the nodes in the network (Yu, Shiwen, & Li, 2017).

Byzantine Generals problem is the problem that is faced by distributed systems. It occurs when a node in the network is infected and is acting maliciously. So, there should be a mechanism to differentiate data that has been altered or generated by malicious nodes. Cryptocurrencies solve this problem by using consensus algorithms (Mingxiao, Xiaofeng, & Zhe, 2017).

In PoW, the miner must generate a hash using a random nonce and the data in the transaction and the hash from the previous block satisfying a certain level of difficulty. The network automatically adjusts the level of difficulty to compensate for the hash rate. If the time taken to add a block takes less time than the expected time, then the difficulty level is increased and vice versa (Sleiman, Lauf, & Yampolskiy, 2016).

The above figure shows the working of the Proof of Work consensus mechanism. The upper part of the figure shows the working of the PoW systems at a very high level. For a miner to add a block, he should take the hash from the previous block, the data of the new block, Merkle root of the transactions and then the miner must select an incremental nonce value and put it through a hashing algorithm to generate a hash. The miner then compares this hash to the target value. If the hash value is less than the target value then, the block is accepted and gets added to the chain. The target value is calculated from a difficulty value which is set automatically by the network depending on the hash rate.

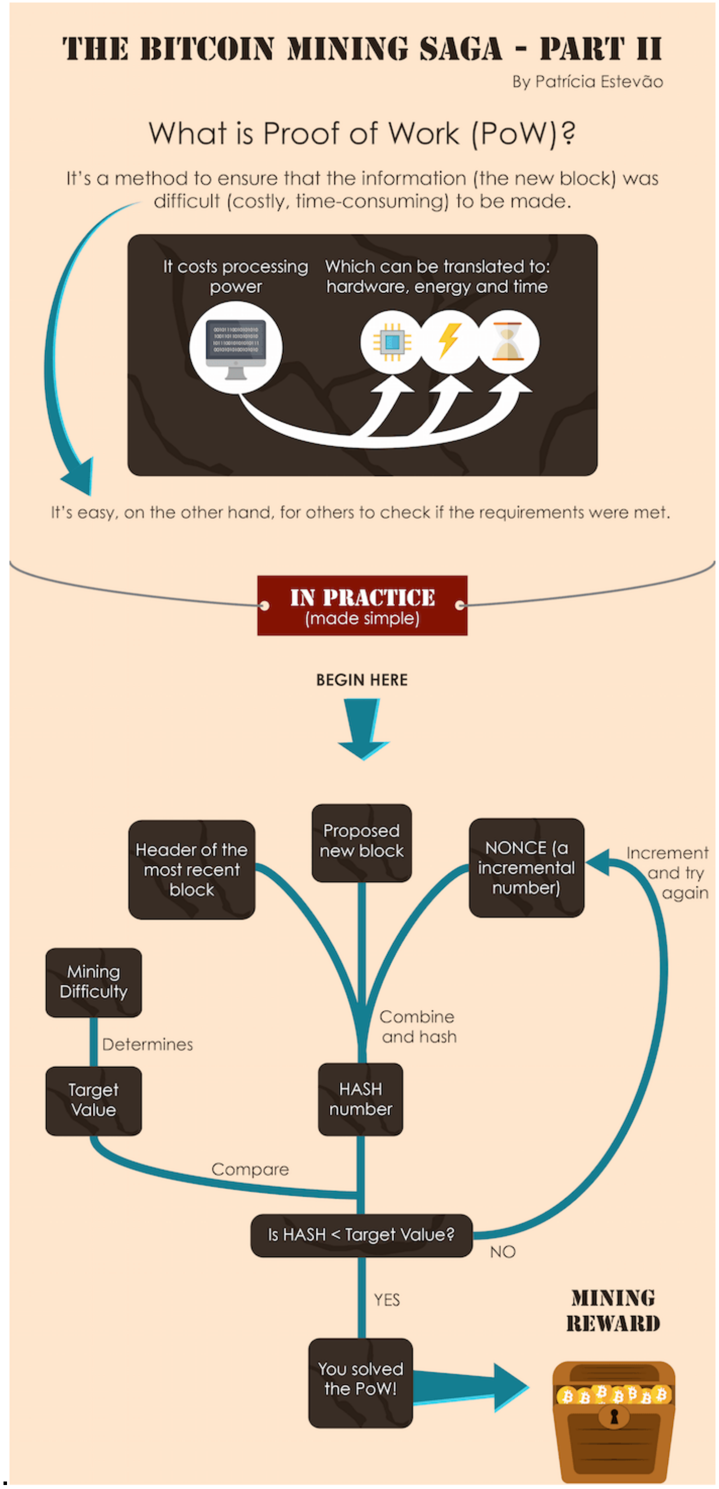


Figure 5: Illustration of PoW systems (source: https://www.bitcoinmining.com/what-is-proof-of-work/)

The PoW consensus creates a sense of competition among the miners for the incentive that is rewarded in addition to the transaction fee. In PoW systems the incentive rewarded to the miner gets halved for every four years, and when the incentive is gone, the only motivation that miners will have is the transaction fee which might lead miners to abandon mining completely (Tosh, Shetty, & Liang, 2018). The PoW increases the wastage of computational power to make the mining difficult and it might lead to a 51% problem and might eventually convert forging into a centralized task.

The hashing algorithm that we will be using for this research is SHA-256 which is the algorithm used by bitcoin. The steps involved in the SHA-256 algorithm are as follows.

Preprocessing.

1. If we consider M as the message to be hashed, and l is the length of M in bits where $l < 2^{64}$, then we create the padded message M' since SHA-256 can only operate on blocks size in multiples of 512, which is message M plus a right padding, such that M' is of length l', a multiple of 512. Specifically, we use a padding P such that M' is multiple of 512.

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

$$\sigma_0^{\{256\}}(x) = ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

2. M’ is split into N blocks of size 512 bits, ranging from M^1 to M^N, and each block is expressed as 16 input blocks of size 32 bits, M_0 to M_{15}.

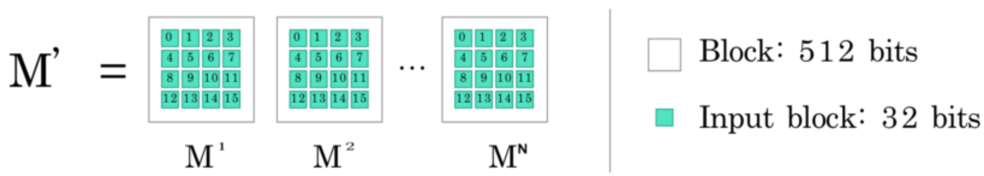


Figure 6: splitting the padded message into blocks

3. To calculate the initial hash H^0, square roots of the first eight prime numbers are calculated. Since prime numbers never have a perfect number, the first 32 bits of their fractional part is taken as H^0 through H_7^0

$$H_0^{(0)} = \texttt{6a09e667}$$
$$H_1^{(0)} = \texttt{bb67ae85}$$
$$H_2^{(0)} = \texttt{3c6ef372}$$
$$H_3^{(0)} = \texttt{a54ff53a}$$
$$H_4^{(0)} = \texttt{510e527f}$$
$$H_5^{(0)} = \texttt{9b05688c}$$
$$H_6^{(0)} = \texttt{1f83d9ab}$$
$$H_7^{(0)} = \texttt{5be0cd19}$$

Figure 7: initial hashes

Hashing.

1. A message schedule W^i is created from four 512-bit message blocks. The first block of W^i is message block M^i, and the next three blocks are variations of M^i.

2. The input blocks are then shuffled. The shuffle function takes a hash $w^i(t)$ and message schedule input block $W^i(t)$ as inputs. The output of shuffle function is a hash $w^i(t+1)$. The diagram below describes the shuffle function.

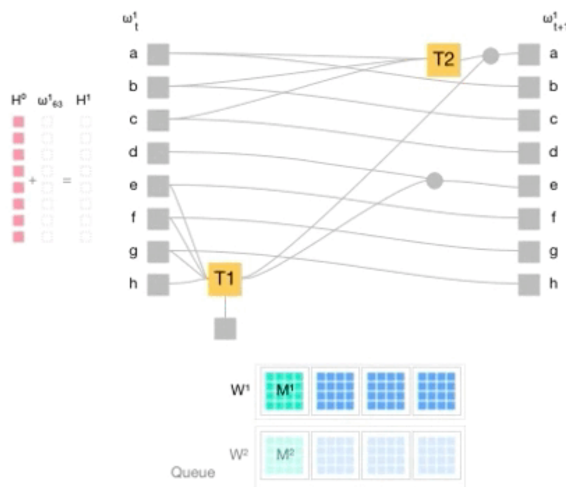


Figure 8: Shuffling the blocks

3. The new hash H^i can be created by using the formula:

$$H^i(j) = H^{i-1}(j) + w^i(63)(j)$$

The same process is to be repeated for each of the input blocks M^i if M^i is the last block then, the hash H^i produced can be considered as the final hash output of the algorithm.

Unlike Proof of work technique does not involve any incentive or intense computation but instead for any user to be able to become a forger (one who mines coins), he must hold a stake in the network by involving some of his coins. In this mechanism, the forger puts his coins at stake and if the forger is involved in any malicious behavior, all his coins, his ability to continue as a forger will be taken away from him.

In PoS, the creator of a new block is chosen deterministically. The forger with the highest stake will the given the highest priority to forge coins and the second priority will be given to the one holding the second highest number of coins and so on. Simply put, the percentage of blocks that can be forged by a forger will be the percentage of coins that he owns in the whole network (Tosh, Shetty, & Liang, 2018).In PoS systems, there is no incentive given to the miners. The number of currency units in the entire network is fixed, and the same coins are circulated throughout the network. The miners get to keep the transaction fee associated with a block upon successful addition of a block to the blockchain. Ethereum is the most popular cryptocurrency which is switching from PoW to PoS They designed a system namely Casper which provides some defenses against fatal crashes (Griffith, 2017).

Structure of bitcoin block.

1. Magic Number: This is a number that represents that the data contained in a bitcoin block. Its value is always “0xD9B4BEF9”.
2. Block Size: It is an Integer, and it represents the block size in bytes.
3. Transaction counter: This represents the number of transactions in the bitcoin block. The number of transactions in the bitcoin block does not have any restrictions. It is totally up to the miner. The minimum limit is one transaction. Though there is no maximum limit, the number of transactions that can be included in the bitcoin block is limited by the size of the total block. The size of a block can never exceed 1MB.
4. Transactions: These are the transactions that are included in this block.

5. Block Header: This is the header that represents the block and used to calculate the hash of the entire block by varying the nonce value.

Structure of a bitcoin transaction.

1. Version: This indicates the version of the Bitcoin protocol being used.
2. In-Counter: This represents the number of inputs to this transaction.
3. List of Inputs: These are the list of inputs that are being used in the transaction.
4. Out-Counter: It is an Integer and represents the number of outputs of this transaction. The outputs represent the value of an address.
5. List of Outputs: This represents the list of outputs of this transaction.
6. Lock time: Lock time is an integer and can be of two types. If its value is greater than 500 million, it represents the time in epoch milliseconds after which the transaction can be added to a bitcoin block. If this value is less than 500 million, it represents the number of blocks that should be added after the block containing this transaction is added to the blockchain.

Structure of transaction inputs.

1. Previous Transaction Hash: This represents the previous transaction hash whose one of the outputs are being used in this transaction.
2. Transaction Out-Index: This represents the index of the output the previous transaction that should be used in this transaction. Indices start at zero.
3. Input Script Length: This represents the size of the input script.
4. Input Script: This is the script that proves that the output owner is legitimate.

5. Sequence Number: This number is used for replacement and is not currently not being used as a replacement is disabled.

Structure of transaction outputs.

1. Value: This represents the amount that is being transferred for the current output in satoshis. A Satoshi is the smallest divisible unit of a bitcoin. Each Satoshi is worth 10^{-8} bitcoins.
2. Output Script Length: It represents the length of the output script.
3. Output Script: This is the script that that is to be satisfied by anyone who wants to use this output as a part of their input for a transaction.

Structure of a block header.

1. Version: This represents the version of bitcoin that is used for the block.
2. Hash Merkle Root: This represents the Merkle root for the transactions.
3. Hash Previous Block: This represents the hash of the previous block to which this block is to be linked in the blockchain. This value is the hash of the last block in the blockchain. The hash is the field that makes the chain by linking one block to the one before it.
4. Time: The time in epoch milliseconds when this block is generated.
5. Target: The target value at the time of creation of this block. The hash of this block should be lesser than this target.
6. Nonce: It is an Integer and is the value that is altered by miners so that hash of the block is less than the target. Miners typically start with a nonce value of 0 and keep incrementing it for each iteration until they obtain a hash value that is less than the target.

The input and output scripts used by bitcoin are written using a set of instructions specified by the Script instruction set. Below are few of the instructions used by bitcoin scripts.

Table 2: Script Operations (Source: https://en.bitcoin.it/wiki/Script)

Word	**Description**
OP_0, OP_FALSE	An empty array of bytes is pushed onto the stack. (This is not a no-op: an item is added to the stack.)
OP_PUSHDATA	The next byte contains the number of bytes to be pushed onto the stack.
OP_1NEGATE	The number -1 is pushed onto the stack.
OP_1, OP_TRUE	The number 1 is pushed onto the stack.
OP_VERIFY	Marks transaction as invalid if top stack value is not true. The top stack value is removed.
OP_DUP	Duplicates the top item of the stack.
OP_DEPTH	Puts the number of items of items in the stack onto the top of the stack.
OP_IFDUP	Duplicate the top of the stack if it is not zero.
OP_SWAP	The top two items of the stack are swapped.
OP_DROP	Removes the top item from the stack.
OP_SIZE	Adds the size of the item at the top of the stack to the stack. This operation does not pop the item on the stack.

OP_EQUAL	Results in 1 if both the items on the top of the stack are equal else results in 0. Two items are popped from the stack by this operation.
OP_EQUALVERIFY	Pops two items from the top of the stack and compares them. If they are equal, next operation is performed else exits with a failure.
OP_RIPEMD160	The item on the top of the stack is hashed using RipeMD160 hashing algorithm and is put back onto the top of the stack.
OP_SHA256	The item on the top of the stack is hashed using the SHA256 hashing algorithm, and the result is pushed back to the stack.
OP_CHECKSIG	Checks a digital signature to a public key. The item on top is used for public key, and the item below it is considered a digital signature. It results in 1 if the digital signature is valid else results in 0.
OP_CHECKSIGVERIFY	Checks a digital signature to a public key. The item on top is used for public key, and the item below it is considered a digital signature. If the result is true, the transaction is considered valid else it is rejected.

To calculate the hash of a block, the miner starts by calculating the SHA-256 hash of each of the transactions. The order of fields for transaction hashing is version number, input counter, list of inputs, output counter, list of outputs, lock time. The order of fields for inputs is a previous transaction hash, output index, input script length, a sequence number. The order of fields for outputs is value, output script length followed by output script. All these fields are put together in the specified order, and a SHA-256 hash of the resulting string is calculated. When the miner is done calculating these hashes, the miner calculates the hash Merkle root for all the transactions.

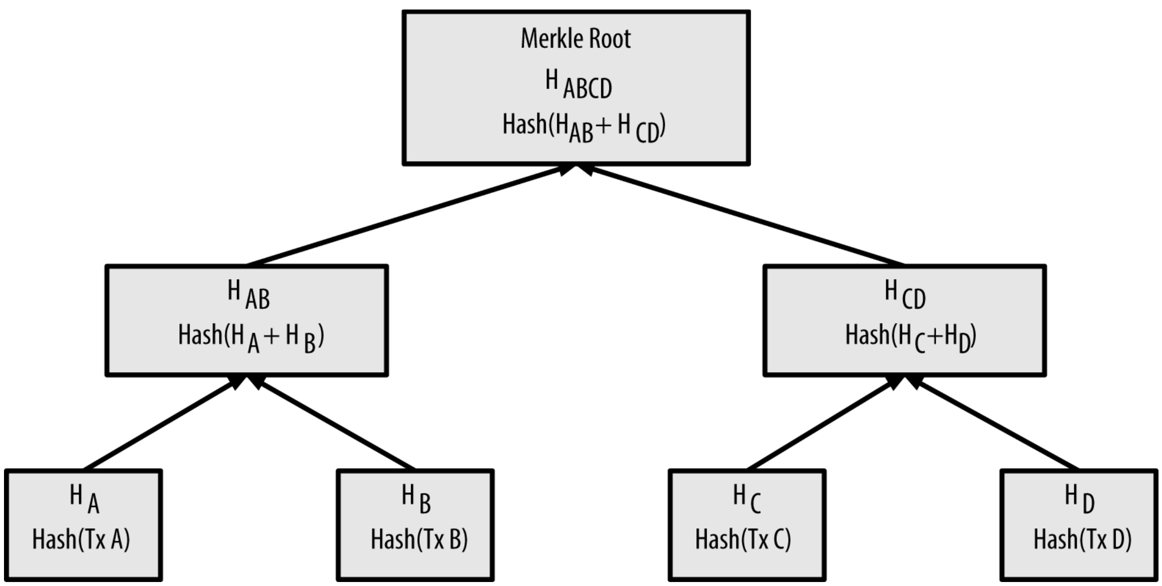


Figure 9 Calculating Hash Merkle Root (Source: https://i.stack.imgur.com/ExJSC.png)

After obtaining the hash Merkle root for the transactions, the block header can be hashed to get the final block hash. To calculate the block hash, the order of fields is version, hash of the previous block, hash merkle root for the transaction set, time, target, nonce. All these fields are put together and then they are hashed twice using SHA-256 algorithm to get the final hash. It is important to remember that the byte ordering followed by bitcoin systems is big endian. So, the byte ordering for the previous hash, version, target is to be reversed before calculating the hash.

For a block to be accepted to put in the block chain, the hash value of the block should be less than the target value. To achieve this, miners alter the nonce and recompute the hash every time making mining a computationally intensive task. Miners usually start off at nonce 0 and keep incrementing it after every in-successful hash and then calculate the hash after every incrementation until they reach the nonce when included in the block makes the hash value lesser the target value.

Literature Related to the Problem

In Mukhopadhyay, U., Skjellum, A., & Hambolu, O.'s A Brief Survey of Cryptocurrency Systems, the authors have explained the basics of cryptocurrencies and the related terms. The paper starts off by explaining the origins of cryptocurrencies all the way from Chaum creating the first anonymous electronic money system to the latest cryptocurrencies that are based on blockchain technology. The paper talks about the basic structure of bitcoin block, the Genesis block and how the links are formed between the blocks at a very high level (Mukhopadhyay, Skjellum, & Hambolu, 2017).

The paper also talks about the different consensus mechanisms that can be used in cryptocurrencies also talks about few of the vulnerabilities of each of these mechanisms. The author talks about the working of 51% attack mechanism (Mukhopadhyay, Skjellum, & Hambolu, 2017) and then discusses the different consensus algorithms used by different cryptocurrencies and the hash algorithms involved in them.

In their paper, Zheng, Z., Xie, S., & Dai, H talked about cryptocurrencies and blockchain technology on the whole. They introduced the structure of blockchain and explained the link formation in the blockchain. The paper also talks about different consensus techniques, advantages, and disadvantages associated with each of them. Then it discusses different issues with cryptocurrencies (Zheng, Xie, & Dai, 2017).

In Sleiman, M. D., Lauf, A. P., & Yampolskiy, R's Bitcoin Message: Data Insertion on a Proof-of-Work Cryptocurrency System, they talked about a concept of embedding messages in a cryptocurrency network. This concept was introduced in Jonathan Warren's paper Bit message: A Peer-to-Peer Message Authentication and Delivery System (Warren, 2012). The authors chose to implement this concept by embedding the encoded message inside the bitcoin's blockchain by encoding the message as the transaction amount and creating a transaction in the blockchain. This approach works since it is a legit transaction transferring a certain number of bitcoins from one address to another (Sleiman, Lauf, & Yampolskiy, 2016). But every time they want to send a message, the message would be accompanied by a transaction fee that goes to the miners for verifying the transaction and is a high cost for fulfilling the task of sending a simple message and moreover there is no confidentiality involved as the data would be posted to a publicly maintained blockchain and the encoding can be broken pretty easily.

In Ahram, T., Sargolzaei, A., & Sargolzaei,S's paper on Blockchain Technology Innovations, they talked about the concept of bitcoin and its applications. The paper introduces the concept of Blockchain powered health chain, a system where the details of the patients are to be stored in blockchain which is maintained by a private organization. The paper talks about HIPAA act which addresses privacy of a patient's data and blockchain can achieve it (Ahram, Sargolzaei, & Sargolzaei, 2017).

In Nayak, K., Kumar, S., & Miller, A's paper Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack, they discuss mining in a mathematical aspect. They talked about selfish mining by using a mathematical approach and compare it to honest mining and how selfish mining affects honest miners. They introduce the concept of stubborn mining which is an extension of selfish mining. In selfish mining, a miner mines and keeps his blocks private when he is in the lead and cooperates with the network if he is not in the lead. In Stubborn mining, the miner keeps working on his private chain even when he is not in the lead (Nayak, Kumar, & Miller, 2016).

Literature Related to the Methodology

In Michael Bedford Taylor's paper The Evolution of Bitcoin Hardware, the author introduces the working of the bitcoin system. He then introduces what mining is and how it works and how blocks are generated. The paper mainly focusses on the hardware involved in the mining and introduces the audience to the concept of hash rate and why it is important regarding mining and how it has a direct impact on the price of bitcoin. The paper talks about how ASIC miners have changed the concept of mining turning it into a race of power-hungry machines mining bitcoins (Michael Bedford Taylor, 2017).

In Lewis Tseng's paper on Bitcoin's consistency property, the author speaks about the consistency property of bitcoin. The paper quotes the bitcoin's eventual consistency property as "Without any new transactions, any participants eventually maintain the same Blockchain, i.e., each participant has the same chain at its local storage.” (Tseng, 2017). The paper provides a simple example transaction where bitcoin protocol does not follow the eventual consistency property. The situation occurs when the blockchain is empty and two miners a & b try to add transaction T_a, T_b to the chain at the same time and the transactions get buffered at the chain and as each of these miners have added only one block they will only have their block as the chain thereby violating the eventual consistency (Tseng, 2017).

Bag, S., Ruj, S., & Sakurai, K’s paper on Bitcoin withholding attack talks about the withholding attack on bitcoin mining pools. The authors start off by explaining what the bitcoin withholding attack is, and then they introduce a scheme which they used to implement the withholding attack. Then they talk about the system model for sponsored withholding attack. They provide mathematical proof for implementing the attack they introduce their lemmas theorems, and corollaries to prove the attack is feasible. They end the discussion by saying how a like-minded mining pool can be profited by the Bitcoin Withholding attack (Bag, Ruj, & Sakurai, 2016).

Summary

This chapter provides a detailed overview of the technology involved in cryptocurrencies. It introduces the concept of the blockchain, how blockchain works, how transactions are performed by the users and the involvement of miners in the transaction to add it to the blockchain. It also explains how miners are benefitted from mining operations. It introduces consensus mechanisms for mining and their working in a great level of detail. The chapter ends with the past works done in the field of cryptocurrencies.

Chapter III: Methodology

Introduction

This chapter describes the coin duplication attacks which will be implemented in this study. This chapter describes how the simulation mechanism will be implemented to create an attack environment and to implement the attack. The design and implementation details of the attack will be discussed in detail. The chapter also discusses the algorithms that will be used to implement the system.

Design of the Study

The research design is all about planning the overall strategy which can effectively address different issues that come up during the implementation. The main issue with implementing the duplication mechanism is the sheer number of nodes and miners across the globe. The bitcoin mining network is gigantic regarding computing power. All the miners involved in the network mine 24/7 to validate the chain. The miners check each transaction that is added to the chain and ignore any fake transactions.

To overcome this problem, the attack will be made on a simulation network which resembles a cryptocurrency network. Designing the simulation network involves the following steps

1. Learn and understand all the implementation details of the cryptocurrency system.
2. Try to analyze each algorithm and consensus mechanisms implemented in the network.
3. Understand the different components of the network and their roles.

4. Create and setup virtual machines where the nodes will live.
5. Install and configure the simulation on the nodes that were just created.
6. Setup the cryptocurrency system on the virtual machines and configure the network.
7. Make a test run inserting a few blocks to make sure that the implementation works.

The Proof of Work mechanism will be used as consensus mechanism for the simulation. Bitcoin is one of the most popular cryptocurrencies implementing this technique. In proof of work implementation, the miner gets new coins with the addition of a new block to the blockchain in addition to transaction fees. In PoW, the miner must generate a hash using a random nonce and the data in the transaction and the hash from the previous block satisfying a certain level of difficulty. The hashes are tough to generate but can be verified very easily by just computing the hash using the nonce and comparing to the hash produced by the miner who added the block. For the simulation, the difficulty level will be level so that the computational power of an ordinary machine will be able to mine the cryptocurrency coins.

When the simulation is done, it is time to design the main attack. The design details of the attack are as follows.

1. There will be a node that has bitcoin daemon running on it. This will be a virtual machine running on the same machine as the miner.
2. Every node and miner in the bitcoin network communicate with each other by exchanging some bitcoin specific messages on port 8333.

3. Designing a mechanism where communication is achieved by exchanging text messages can be daunting experience. There by the attack would be using a third-party library (insight-api) that is installed on top of bitcoind and translates these messages into http calls available at certain endpoints.
4. The miner will be implemented to communicate to the node via http using the insight-api.
5. The miner will make api calls to get the list of pending transactions from the node. Since it is up to the miner to include any number of blocks, in this case we are using 20 transactions. Then the miner adds a coinbase transaction to the list of transactions which is the block reward for mining the block according to the bitcoin protocol.
6. The miner then adds another transaction to the list of transaction which is fake and would not be accepted by any other miners connected to the node. In this case, there is only one miner connected to the node there by there would be no one to complain about the fake transaction being added to the block.
7. The miner then computes the hash merkle root of these transactions together and then starts off by computing the hash of the block using a nonce value 0 and then keeps incrementing it until a valid hash value which is lesser than the target is obtained by altering the nonce values.
8. When such a nonce is discovered, the miner stops and then reports the block to the node. But there is a catch here. The amount added by these

transactions are valid only after the block reaches certain depth (In main-net it is 6 blocks and in reg-test it is 4 blocks).

9. The miner should add three more blocks to the chain before the bitcoins added to the previous block are available for spending. Therefore, the same process is repeated but without adding a fake transaction for three more times. Therefore, the miner adds four blocks in total to the blockchain.
10. The node can be setup on a virtual machine running on a host because it is just a storage node and doesn’t need much computation power. The miner on the other hand must run on the host machine where the computational power is more than that of a virtual machine.

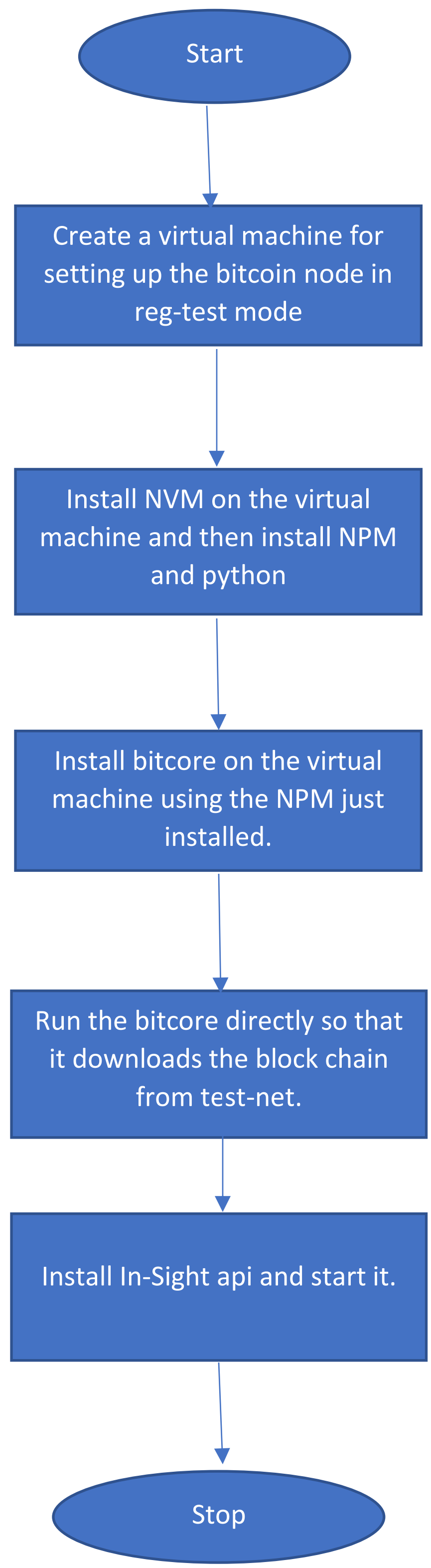


Figure 10 Flowchart for implementing the simulation network

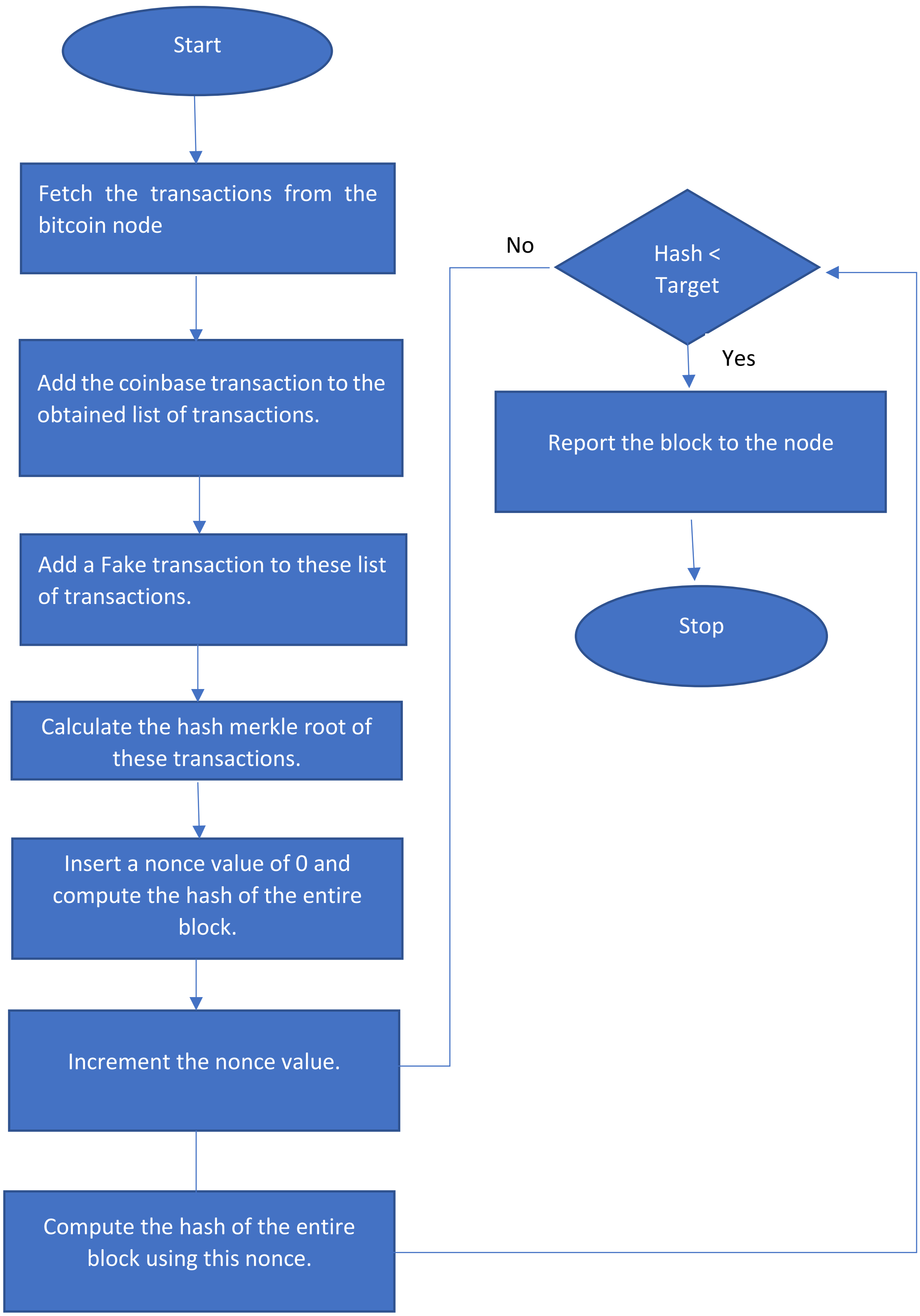


Figure 11: Flowchart for implementing miners

Data Collection

The data needed for this research is algorithms and mechanisms that are used to implement the cryptocurrencies. The data related to these implementations are freely available on the internet. Numerous articles describe the mechanisms of cryptocurrencies in detail. Also, the concept of a decentralized public ledger means that all the transactions ever made are available to everyone who wants to analyze these transactions. The details of the transaction amount, wallet address, the block hash, the address of the miner who mined the block, the number of transactions included in a block are all available online.

Every year there will be conferences held on the security of cryptocurrencies and new additions to the cryptocurrency protocols. These conferences discuss security of the systems and analyze solutions to solve those vulnerabilities. IEEE is a standard which conducts such surveys every year, and the details of conferences can be obtained from IEEE website.

Tools and Technology

To implement the attack, we need a target to attack. The real bitcoin network is huge and is not easier to attack. There are miners guarding the network who are always validating and verifying the transactions added to the network. There is a new block being added to the blockchain for every 10 minutes and any fake transactions added to the chain are just ignored by the miners and the transaction would never get added to the blockchain. The bitcoin network can be attacked if the mining power of the attacker is greater than that of all the miners combined. The miners have capacity to add a new block every 10 minutes. If a legit miner adds a block to the chain before the attacker, the node would propagate that block reported by the legit miner and the block reported by the attacker would not be accepted because it is no longer valid as a similar block is already added to the chain which has the hash of the same previous block and the blockchain is immune to forking.

Bitcoin is not a specific piece of software, it is a specific set of protocols which need to be implemented to perform transactions over the network. There are several client applications which implemented the protocol. Anyone can join the network using their own client application. There are several client applications that are freely available (Bcoin, decred, btcd, Bitcore etc.). Bitcore is the most popular one that supports decent features. Bitcore is an open source client written in java script. Bitcoin has 3 different modes that it can run in. If Bitcore is run no additional flags, it runs in Mainnet mode.

The Mainnet is at least 160GB currently and to setup a node that uses mainnet, all these blocks should be downloaded before any operations can be performed on the chain. There is also a testnet which is like mainnet but only difference being that coins from mainnet are not valid in the testnet and vice versa. The testnet is currently 16GB which is far less than that of the mainnet. The testnet is mostly used by developers who develop applications targeting bitcoin payments and for some other purposes. In the test net too, there are miners who validate the transactions and it would be tough to add fake transactions to the chain. Though attacks can be performed on the Testnet, it is complicated and has been reset twice before. To overcome these problems, the attack would be performed on a simulated network where there are no competitive miners and the attacker can freely mine blocks and report them to the bitcoin node. Bitcoin also offers one more called Simnet mode. In Simnet mode, the node is setup locally with no connections to any other node from mainnet or testnet. A Bitcore node setup in simnet mode is the perfect target for the attack as there would be no other miners involved to verify and validate the transactions added to the network.

Setup bitcore in simnet mode. Bitcore is a bitcoin client application written using Nodejs. To install and to run Bitcore, we need to setup Node Version Manager (NVM) and then install Node Package Manager (NPM) using NVM. When we have NPM installed, we will also need python3 installed. We will also need to execute 'apt-get install libzmq3-dev build-essential.

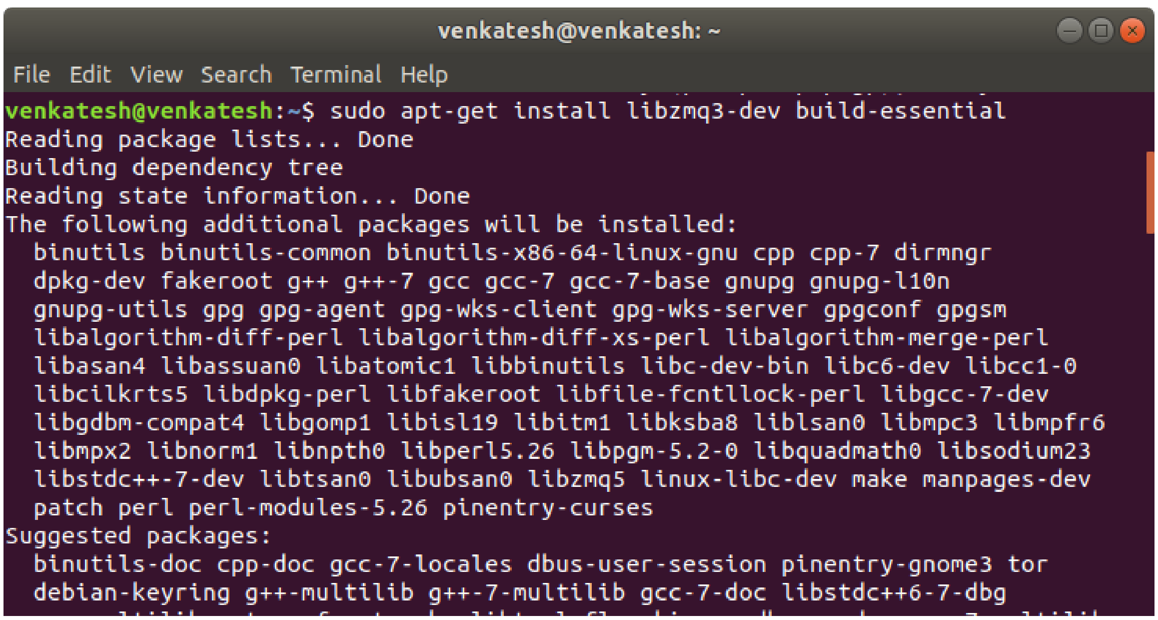


Figure 12: Installing Bitcore dependencies

With all the criteria met, we can install Bitcore by using npm by using the command 'npm install -g bitcore'. This command installs Bitcore globally across the system. After the installation of Bitcore, we need to setup a node using Bitcore which can be done easily using the command 'bitcore create <node name> --simnet'. This will make Bitcore download the block chain create a node with the given node.

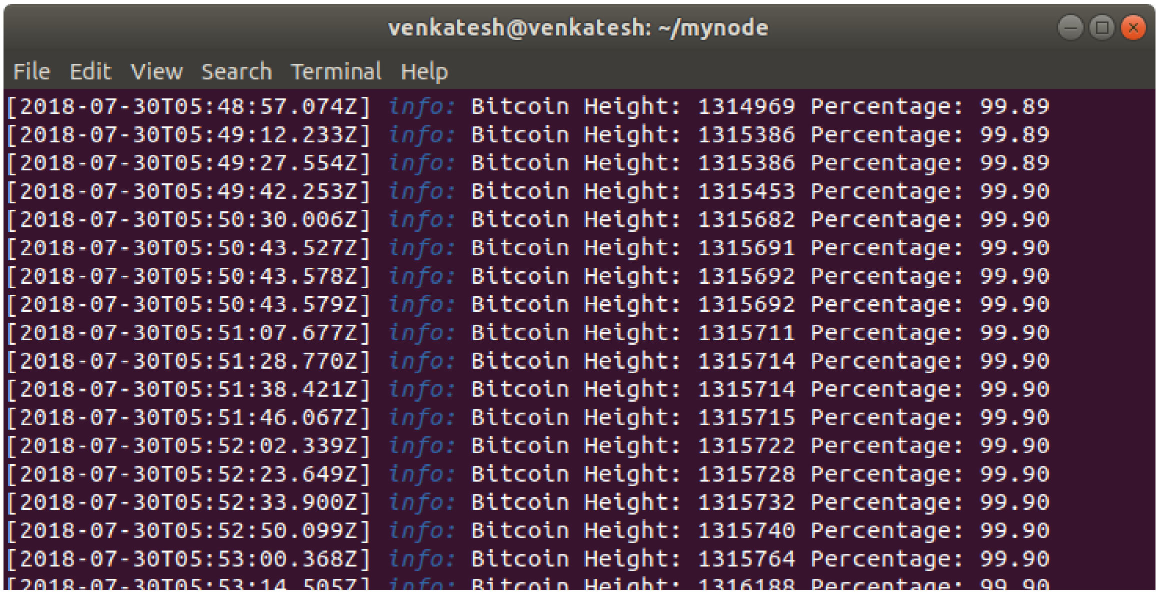


Figure 13: Bitcore Downloading blockchain

The Bitcoin protocol specification allows communication over a specific port (depends on the network type). The communication is done using some messages specific to the protocol. It would be a little complicated to make communication using these messages as we need to code all these messages and communicate over a socket. Instead communicating over http would be easy. Insight API is a software package that can be installed on top of Bitcore client. This provides a convenient http API which offers different endpoints to get and post transaction and block data from the node. Insight API can be installed using the command ‘bitcore install insight-api’ and then when Bitcore is started, along with it starts the In-sight api.

To implement the attacking miner java language will be used. The reason for choosing java over other programming language is due to the huge set of libraries that it provides and making http calls using the libraries and converting the response from the node into objects and vice versa is comparatively easy and non-clumsy. Gradle is used for dependency management and as a build tool. Spring framework is used for managing beans and dependency injection.

```
public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(Miner.class);
    MinerUtil fraudMiner = ctx.getBean(MinerUtil.class);
    fraudMiner.startMiners();
  }
```

This is the main class that would be run by the gradle daemon. This class creates the two necessary beans one of type OkHttpClient which will be used all over by the application to make http calls. The other bean is of type ObjectMapper which is used to convert json strings to objects and vice versa. This class also has the main method which gets the bean of type MinerUtil which contains the code responsible for starting the miner thread and then joins it to the main thread so that the application does not terminate while this thread is still running. We will look at the MinerUtil class in a moment. For now, it is class which has the miner implementation and the bean is automatically created and managed by the spring container service. Bitcoin protocol involves data exchange in hexadecimal format and Big-Endian byte ordering. The application needs to convert this data back to data that can be worked with.

The application has a utility class named HexUtil that helps with these conversions. All the methods of this class are public static methods. The first of them is reverseByteOrdering. This method accepts a String argument which is proper hexadecimal format. This method is used by the application to reverse the byte ordering of a hexadecimal data and vice versa. This method works by decoding the hexadecimal string to get a byte array which is reversed and then gets pushed to a byte buffer and is then extracted from byte buffer, encoded and then converted back to string and returned. The method reverseOrderAndParseLong is used by the application to reverse byte ordering of the input string and then converting the obtained value to a Long which gets returned.

The TxIn class represents a transaction input to a transaction. The class has the fields 'previousTransactionHash' which is of type String and represents the hash of the previous transaction that is to be used as input for this transaction. The field 'txOutIndex' is of type String which indicates the output index in the output set of the previous transaction. This field is made as a String because it is represented in hexadecimal format. The field 'txInScriptLength' represents the length of the input script for the specified input. This is a string that represents the length of the input script in hexadecimal format. The field 'txInScript' represents the input script for the transaction. It is a string containing the script which will be validated by the miner during the transaction verification. The field sequence number is a string type field. In addition to the above fields, this class overrides toString method to return the data in hexadecimal format for hashing.

The TxOut class is model class that represents the output of a transaction. This class has the fields value which is of type String and represents the value of output in satoshis (10^{-8} bitcoin) in hexadecimal format. This field follows Big Endian byte ordering and needs to be reordered before processing. The field txOutScriptLength represents the size of transaction output script in bytes with big endian byte ordering. The field txOutScript represents the output script which will be validated when a transaction wants to use this output as an input. This too is in hexadecimal format with big endian byte ordering. This class overrides the toString method from object class to combine all the above-mentioned fields by reversing their byte ordering and returns the concatenated string.

The Transaction class represents a bitcoin transaction. This class has the fields versionNo which represents the version of bitcoin protocol being used in hexadecimal format with big endian byte ordering. The field inCounter represents the number of inputs to transaction in hexadecimal format with big endian byte ordering. The field listOfInputs represents the inputs to this transaction. The field outCounter represents the number of outputs from this transaction in hexadecimal format with big endian byte ordering. The field listOfOutputs represents the outputs from this transaction. The field lockTime is a string representing locktime and can be of two types. If its value is greater than 500 million, it represents the time in epoch milliseconds after which the transaction can be added to a bitcoin block. If this value is less than 500 million, it represents the number of blocks that should be added after the block containing this transaction is added to the block chain.

```
public Try<String> calculateHash() {
    return Try.of(() -> {
      String data = "";
      data += HexUtil.reverseByteOrdering(this.versionNo);
      data += HexUtil.reverseByteOrdering(this.inCounter);
      for (TxIn in : this.listOfinputs) {
        data += in.toString();
      }
```

```
        data += HexUtil.reverseByteOrdering(this.outCounter);
        for (TxOut in : this.listOfOutputs) {
          data += in.toString();
        }
        data += HexUtil.reverseByteOrdering(this.lockTime);
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(data.getBytes());
        byte[] digest1 = digest.digest(hash);
        return HexUtil.reverseByteOrdering(new String(Hex.encode(digest1)));
      });
    }
```

The class also contains a method calculateHash which is used to calculate the SHA-256 hash of the transaction. This method works by reversing the byte ordering of versionNo, inCounter, outCounter, lockTime and concatenating them together in the order versionNo, inCounter, listOfInputs, outCounter, listOfOutputs, lockTime. To get the list of inputs, the method uses the overridden toString implementation from TxIn class which converts the fields from TxIn in the order previousTransactionHash, txInScriptLength, txInScript, sequenceNumber, and returns the concatenated string. To get the list of outputs, the method uses the overridden toString implementation from TxOut class which converts the fields from TxOut in the order value, txOutScriptLength, txOutScript, and returns the concatenated string. It then applies SHA-256 on the hex string twice and then reverses the byte ordering of the resultant hash and the final hash of the transaction.

The BlockHeader class represents the block header of a bitcoin block. The fields in this class are used to calculate the hash of the block. The version field represents the version of bitcoin protocol used for this block. The field hashMerkleRoot represents the hash merkle root of the transactions included in this block. The field hashPrevBlock represents the hash of the previous block. The field time represents the time of creating this block. The field target represents the target value during the block creation time. The field nonce represents the nonce value used to achieve the target. All the fields represented in this class are in hexadecimal format and follow Big Endian byte ordering. The class also includes a method to calculate the hash of the block. This method starts by reversing the byte order of the fields and concatenating them in the order version, hashPrevBlock, hashMerkleRoot, time, nonce. The resultant string is hashed twice using SHA-256 hashing algorithm which in turn is encoded and byte order reversed to get the final hash. This is the hash that is hash of the block containing this header.

```
public Try<String> calculateHash() {
    return Try.of(() -> {
        String data = "";
        data += HexUtil.reverseByteOrdering(this.version);
        data += HexUtil.reverseByteOrdering(this.hashPrevBlock);
        data += HexUtil.reverseByteOrdering(this.hashMerkleRoot);
        data += HexUtil.reverseByteOrdering(this.time);
        data += HexUtil.reverseByteOrdering(this.nonce);
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(Hex.decode(data));
        byte[] digest1 = digest.digest(hash);
        return HexUtil.reverseByteOrdering(new String(Hex.encode(digest1)));
    });
  }
```

The class Block represents a bitcoin block. The field magicNumber represents the magic number of the bitcoin protocol. This field is always set to 0xD9B4BEF9 which represents that the data included is bitcoin data. The field blockSize represents the size of the block in bytes. The field transactionCounter represents the number of transactions included in the bitcoin block. The field transaction represents the transactions that are included in this block. The field blockHeader represents the header corresponding to the block. The fields magicNumber, blockSize, transactionCounter are all in hexadecimal format and follow Big Endian byte ordering. This class has method to compute the merkle root of the transactions which is to be included in the block header. The merkle root can be considered as the combined hash of all the transactions. To calculate merkle root, hashes of each of the transactions are calculated and then the hash of the first transaction is concatenated with the hash of the second transaction and the SHA-256 of this combined hash is calculated. Then the same thing is done for the next transactions and so on. This gives us a set of hashes once more and the same process is continued on more levels until we end up with just one single hash and this hash is called the merkle root for the set of transactions. It is important to remember that bitcoin network has Big Endian byte ordering and this byte ordering of data must be reversed before hashing. The method handles this by using the utility class’s methods.

```
public Try<String> calculateMerkleRoot() {
    return Try.of(() -> {
      MessageDigest digest = MessageDigest.getInstance("SHA-256");
      List<String> hashes = this.transactions
            .stream()
            .map(transaction -> HexUtil.reverseByteOrdering(transaction.calculateHash().get()))
```

```
            .collect(Collectors.toList());
        while (hashes.size() > 1) {
          for (int i = 1; i < hashes.size(); i += 2) {
            digest.reset();
            String hash1 = HexUtil.reverseByteOrdering(hashes.get(i - 1));
            String hash2 = HexUtil.reverseByteOrdering(hashes.get(i));
            byte[] bytes = Hex.decode(hash1.concat(hash2));
            String combinedHash = new String(Hex.encode(digest.digest(bytes)));
            hashes.remove(i);
            hashes.remove(i - 1);
            hashes.add(i - 1, HexUtil.reverseByteOrdering(combinedHash));
          }
        }
        return hashes.get(0);
      });
    }
```

The Bitcoin protocol specification allows communication over a specific port (depends on the network type). The communication is done using some messages specific to the protocol. It would be a little complicated to make communication using these messages as we need to code all these messages and communicate over a socket. Instead communicating over http would be easy. Insight API is a software package that can be installed on top of Bitcore client. This provides a convenient http API which offers different endpoints to get and post transaction and block data from the node. To communicate with Insight-api, the application OkHttp library. The application has a few client classes that help it communicate with the Bitcore node.

The TransactionClient class helps the application get the transactions from the node. This class uses the two beans of types OkHttpClient and ObjectMapper respectively that are maintained by the spring container. These beans are configured to be autowired into the TransactionClient. The TransactionClient is annotated with @Component which makes it a spring component therefore automatically creating a bean and managing it. This class has two methods namely getPendingTransactions and listTransactions. The method getPendingTransactions is responsible for making a http call to the endpoint http://ubuntuvm:8080/insight-api/transactions/pending to get the list of pending transactions and converts them into Transaction objects using the objectmapper that was injected into the bean. The method listTransactions is responsible for making a http call to endpoint http://ubuntuvm:8080/insight-api/transactions and returning the obtained transactions.

The BlockClient class has methods that help the application to get the blocks and to report newly created blocks to the bitcoin node. The class is annotated with @Component which marks it as component bean to be created and managed by the spring application container. This class uses OkHttpClient to make http calls to endpoints and this bean is configured to be autowired by the spring container. The class has two methods namely getLastBlock and postBlock which help the application to get the last block of the blockchain and to post a new block to the blockchain respectively. The getLastBlock method makes a GET http call to the endpoint http://ubuntuvm:8080/insight-api/blocks/last to get the last block that was successfully posted to the block chain, converts the obtained JSON to Block object and then returns it. The postBlock method makes a POST http call to http://ubuntuvm:8080/insight-api/blocks with a new block data in JSON format to add a new block to the blockchain.

The class MinerUtil is the class that does the actual mining. The field transactionClient is an instance of the TransactionClient class. It is a bean managed by the spring container and is constructor injected into the MinerUtil bean which is also managed by the spring container. The field TargetClient is an instance of type TargetClient and is a bean managed by spring container. The field keys are a list which contains the public and private key pairs. This list is populated by the method generateKeysAndAddresses which is configured to run after the MinerUtil bean creation. The fields are injected using constructor injection mechanism are autowired by the spring container. The class has a method generateKeysAndAddress is a method that generates two pairs of public, private keys and generates bitcoin addresses. The first pair of keys is used to generate the script for coinbase transaction. The second set of keys is used to generate the script for the coinbase transaction. This method is annotated with @PostConstruct so that it gets executed soon after the MinerUtil bean's construction. This method uses a static method adjustTo64 which concatenates the key with 0s before it. The method getBlockData uses the transactionClient bean to get the list of pending transactions, current target, the lastly added block, then uses some of the pending transactions to build a block and then returns the same.

```
@PostConstruct
   private void generateKeysAndAddress() throws InvalidAlgorithmParameterException,
NoSuchAlgorithmException, NoSuchProviderException, UnsupportedEncodingException {
       for (Integer I = 1; I <= 2; I++) {
           KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC");
           ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
           keyGen.initialize(ecSpec);
           KeyPair kp = keyGen.generateKeyPair();
           PublicKey pub = kp.getPublic();
           PrivateKey pvt = kp.getPrivate();
           ECPrivateKey epvt = (ECPrivateKey) pvt;
           String privateKey = adjustTo64(epvt.getS().toString(16)).toUpperCase();
           ECPublicKey epub = (ECPublicKey) pub;
           ECPoint pt = epub.getW();
           String sx = adjustTo64(pt.getAffineX().toString(16)).toUpperCase();
           String sy = adjustTo64(pt.getAffineY().toString(16)).toUpperCase();
           String publicKey = "04" + sx + sy;
           keys.add(ImmutablePair.of(privateKey, publicKey));
           MessageDigest sha = MessageDigest.getInstance("SHA-256");
           byte[] s1 = sha.digest(publicKey.getBytes("UTF-8"));
           Security.addProvider(new BouncyCastleProvider());
           MessageDigest rmd = MessageDigest.getInstance("RipeMD160", "BC");
           byte[] r1 = rmd.digest(s1);
           byte[] r2 = new byte[r1.length + 1];
           r2[0] = 0;
           for (int i = 0; i < r1.length; i++) {
              r2[i + 1] = r1[i];
           }          byte[] s2 = sha.digest(r2);
           byte[] s3 = sha.digest(s2);
           byte[] a1 = new byte[25];
```

```
        for (int i = 0; i < r2.length; i++) {
            a1[i] = r2[i];
        }
        for (int i = 0; i < 5; i++) {
            a1[20 + i] = s3[i];
        }
        log.info("address" + l + "={}", Base58.encode(a1));
    }
```

Bitcoin network imposes a restriction on the block size. A block cannot exceed 1mb in size. This application limits the number of transactions to 20. The target client is used to get the current target from the node It is a string target that gets returned from the client and can be directly used to build the block. The method insertCoinbaseTransaction is used by the application to insert a coin base transaction into the block. This method uses the public key generated by generateKeysAndAddresses method. The application uses pay to public key transaction for the coinbase and fake transaction. The input script for these transactions is OP_PUSH_DATA public key CHECKSIG. To use this output, the private key for this public key must be used to generate a digital signature.

The method insertFakeTransaction is used by the application to insert a fake transaction into the block. This method is same as insertCoinbaseTransaction except that it uses the second public key to generate the script for the transaction. The method startMining is the method which changes the nonce value in steps of 1 and calculates the hash of the block. If this hash is more than the target value, the nonce is incremented by 1 and hash is recalculated until a nonce which yields a hash less than target is reached. This method starts off by getting the blockdata from the node and then adds the coinbase transaction and fake transactions to the block and starts calculating the hash starting with a nonce of 0 and incrementing in steps of 1. After every iteration, the application compares the hash to that of the target. If the hash is less than the target, it would be accepted by the node, so it would break out of the loop and post the block to the node.

```
public void startMiners() {
    new Thread(() -> {
      for (int i = 1; i <= 4; i++) {
        Block block = getBlockData();
        insertCoinbaseTransaction(block);
        if (i == 1)
          insertFakeTransaction(block);
        String target = block.getBlockHeader().getTarget();
        for (Long nonce = 0L; nonce < Long.MAX_VALUE; nonce++) {
          block.getBlockHeader().setNonce(HexUtil.formatLongAndReverseOrdering(nonce));
          String hash = block.getBlockHeader().calculateHash().get();
          if (target.compareTo(hash) > 0) {
            log.info("nonce={} hash={} target={}", nonce, hash, target);
            break;
          }}
        if (blockClient.postBlock(block).get())
          log.info("posted block={}", block);
      }
    }).start();
  }
```

Summary

This chapter provides details about the implementation of research. It discusses the design and approach that are taken to implement the attack. It discusses the algorithms and consensus mechanisms that will be used to realize the objectives of the research by overcoming the challenges. It also provides a brief walkthrough of the code to give a high-level view of the implementation

Chapter IV: Analysis of Results

Introduction

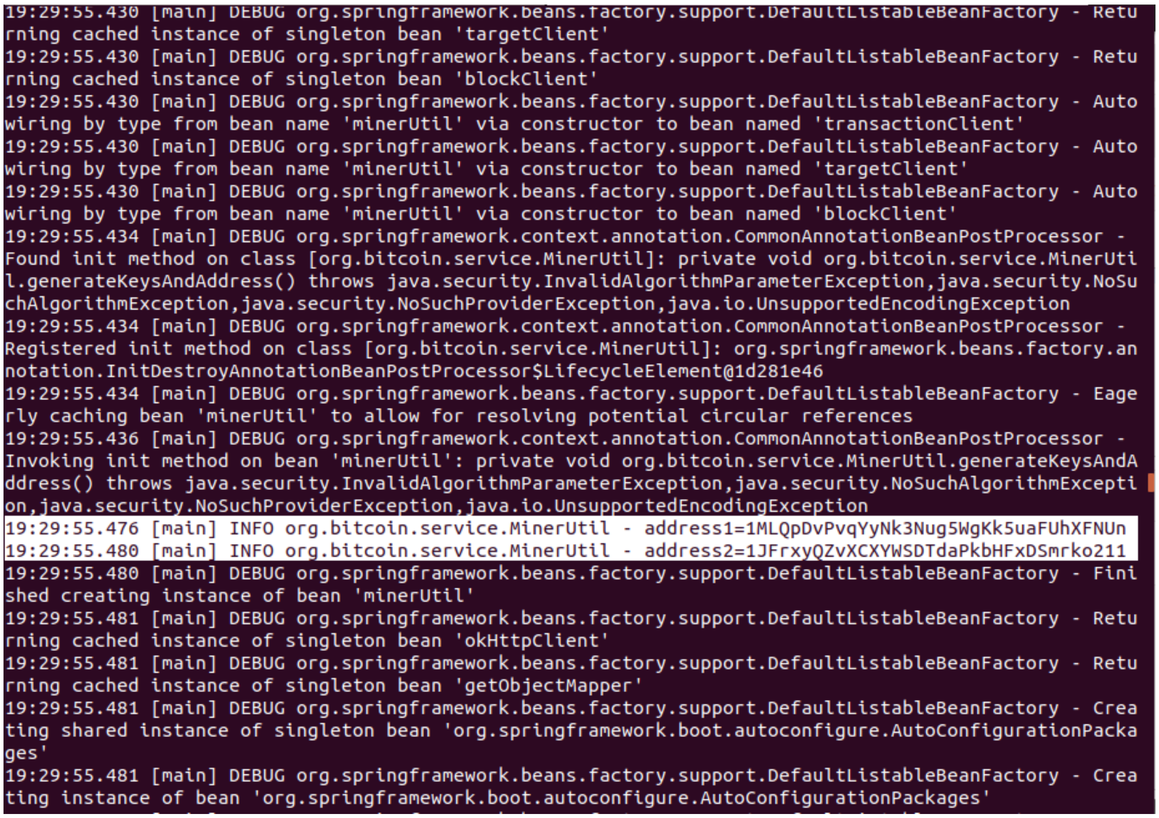
This chapter talks focusses on analyzing the results of the attack implementation. In the previous chapter, we discussed about the implementation design and provided a walkthrough of it. This chapter provides details of the attack and shows screenshots of the attack in various stages and provides a detailed information of the output produced by the attacking miner. This chapter will also provide details related to the execution of the attack by the miner.

Data Presentation

The bitcoin node is setup on a virtual machine using Bitcore bitcoin client. The host is named as ubuntuvm on the host machine. The miner will be run on the host machine since it has more computational power when compared to the virtual machine. The miner can be started by navigating to the directory and using the command java -jar Miner.jar. This automatically starts the Mining operation and will continue running until 4 blocks are added to the blockchain. The node is setup to reduce the difficulty of mining. Despite this lower difficulty, the mining operation takes huge amount of time (approximately 2-3 hours for a single block). Starting the application automatically initializes the spring application container from the main method and then gets the MinerUtil bean from the spring application context and then starts the mining thread.

Figure 14: Starting the mining application

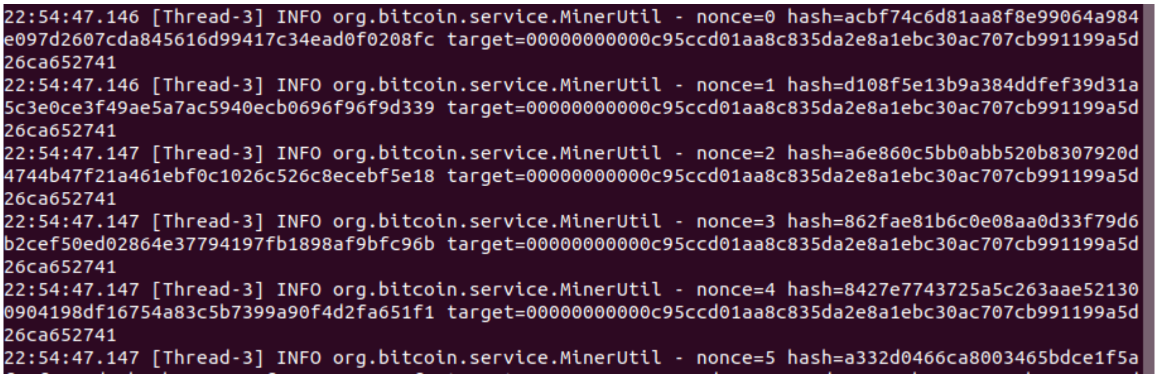
The application uses a separate thread to mine bitcoins since it is not considered as a good practice to run computationally intensive tasks on the main thread. The application spawns a new thread from the main thread and joins it with the main thread. This thread as soon as it is started, makes api call to the Bitcore node to get the list of transactions, creates a block using these transactions and then starts off the mining process with a nonce 1 and then keeps incrementing it till it reaches a hash value that is less than the target value. When the spring application context is initialized, MinerUtil's generateKeysAndAddresses method gets executed which generates the keys set and the address to be used to generate the output script for the coinbase and fake transactions respectively. The application logs these addresses to verify that the addresses are assigned some coins by using the transactions.



```
19:29:55.430 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Retu
rning cached instance of singleton bean 'targetClient'
19:29:55.430 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Retu
rning cached instance of singleton bean 'blockClient'
19:29:55.430 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Auto
wiring by type from bean name 'minerUtil' via constructor to bean named 'transactionClient'
19:29:55.430 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Auto
wiring by type from bean name 'minerUtil' via constructor to bean named 'targetClient'
19:29:55.430 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Auto
wiring by type from bean name 'minerUtil' via constructor to bean named 'blockClient'
19:29:55.434 [main] DEBUG org.springframework.context.annotation.CommonAnnotationBeanPostProcessor -
Found init method on class [org.bitcoin.service.MinerUtil]: private void org.bitcoin.service.MinerUti
l.generateKeysAndAddress() throws java.security.InvalidAlgorithmParameterException,java.security.NoSu
chAlgorithmException,java.security.NoSuchProviderException,java.io.UnsupportedEncodingException
19:29:55.434 [main] DEBUG org.springframework.context.annotation.CommonAnnotationBeanPostProcessor -
Registered init method on class [org.bitcoin.service.MinerUtil]: org.springframework.beans.factory.an
notation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@1d281e46
19:29:55.434 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Eage
rly caching bean 'minerUtil' to allow for resolving potential circular references
19:29:55.436 [main] DEBUG org.springframework.context.annotation.CommonAnnotationBeanPostProcessor -
Invoking init method on bean 'minerUtil': private void org.bitcoin.service.MinerUtil.generateKeysAndA
ddress() throws java.security.InvalidAlgorithmParameterException,java.security.NoSuchAlgorithmExcepti
on,java.security.NoSuchProviderException,java.io.UnsupportedEncodingException
19:29:55.476 [main] INFO org.bitcoin.service.MinerUtil - address1=1MLQpDvPvqYyNk3Nug5WgKk5uaFUhXFNUn
19:29:55.480 [main] INFO org.bitcoin.service.MinerUtil - address2=1JFrxyQZvXCXYWSDTdaPkbHFxDSmrko211
19:29:55.480 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Fini
shed creating instance of bean 'minerUtil'
19:29:55.481 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Retu
rning cached instance of singleton bean 'okHttpClient'
19:29:55.481 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Retu
rning cached instance of singleton bean 'getObjectMapper'
19:29:55.481 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Crea
ting shared instance of singleton bean 'org.springframework.boot.autoconfigure.AutoConfigurationPacka
ges'
19:29:55.481 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Crea
ting instance of bean 'org.springframework.boot.autoconfigure.AutoConfigurationPackages'
```

Figure 15: Addresses generated by the application

The application does not log the keys used since the Bitcore node can be queried by using the public addresses and do not need the keys for that. As explained previously, the mining process starts off with a nonce value of 0 and keeps incrementing for each of the rounds until a hash with lesser value is obtained.



```
22:54:47.146 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=0 hash=acbf74c6d81aa8f8e99064a984
e097d2607cda845616d99417c34ead0f0208fc target=00000000000c95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d
26ca652741
22:54:47.146 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=1 hash=d108f5e13b9a384ddfef39d31a
5c3e0ce3f49ae5a7ac5940ecb0696f96f9d339 target=00000000000c95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d
26ca652741
22:54:47.147 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=2 hash=a6e860c5bb0abb520b8307920d
4744b47f21a461ebf0c1026c526c8ecebf5e18 target=00000000000c95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d
26ca652741
22:54:47.147 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=3 hash=862fae81b6c0e08aa0d33f79d6
b2cef50ed02864e37794197fb1898af9bfc96b target=00000000000c95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d
26ca652741
22:54:47.147 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=4 hash=8427e7743725a5c263aae52130
0904198df16754a83c5b7399a90f4d2fa651f1 target=00000000000c95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d
26ca652741
22:54:47.147 [Thread-3] INFO org.bitcoin.service.MinerUtil - nonce=5 hash=a332d0466ca8003465bdce1f5a
```

Figure 16 Application mining bitcoins

Data Analysis

The application log is huge because it generates a log statement for each of the nonce values and to generate a single block it takes at least 1 million iterations. So, it logs at least 1 million lines for each block and four such blocks need to successfully add a block containing fake transaction which means that in the best case it logs out 4 million lines which is a lot and the terminal buffer doesn't have the ability to hold all these log statements. During the development of this application, alternative options like writing the logs to a different file was tried out but even that approach failed causing the host system to run out of disk space. So, application was modified to printout the log statements only when a block was successfully generated and reported back to the node.

Figure 17: Adding the blocks to the blockchain.

The screenshot above shows the output of the application after successful mining of four bitcoin blocks. The target value obtained from the Bitcore node is 0000000ad1dc95ccd01aa8c835da2e8a1ebc30ac707cb991199a5d26ca652741. The nonce value used for the blocks are 166213763, 860207930 and 1082383936 respectively. According to the bitcoin protocol, any funds added by a block are only available only after the block containing the transaction reaches certain depth. In the main chain, this depth should be at least 6 meaning that funds added by a transaction are only available when the block containing this transaction is at depth 6. In the main chain, a new block gets added to the chain every 10 minutes. So, after a new transaction is added to the chain, it takes at least 1 hour for the funds to be available for spending. In Reg test mode, this depth is 4 blocks. So, a fake transaction is added to the chain using the first block and then, three more blocks are added to the block by the application.

Results

Approximately running for about 8 hours, the application was able to add four blocks to the blockchain. This was possible since there were no other miners in the network who would verify the block reported by the fraud miner. When all the four blocks were successfully reported to the node, these funds are readily available for spending which can be verified by querying the node for balance of the address that was logged by the application. The api also provides with an endpoint which enables anyone to query the UTXOs (Unspent transaction outputs) of the node. The bitcoin node keeps track of unspent transactions using a database that is separate from the blockchain itself.

There is no partial spending of a transaction output in bitcoin network. In a situation where only, a part of a transaction output is to be spent, the transaction has an extra output which pays to the same address which contains the change from different output. This way, there is a standardized way to keep track of un-spent outputs. The node can be queried for the un-spent outputs using the endpoint provided by the insight-api.

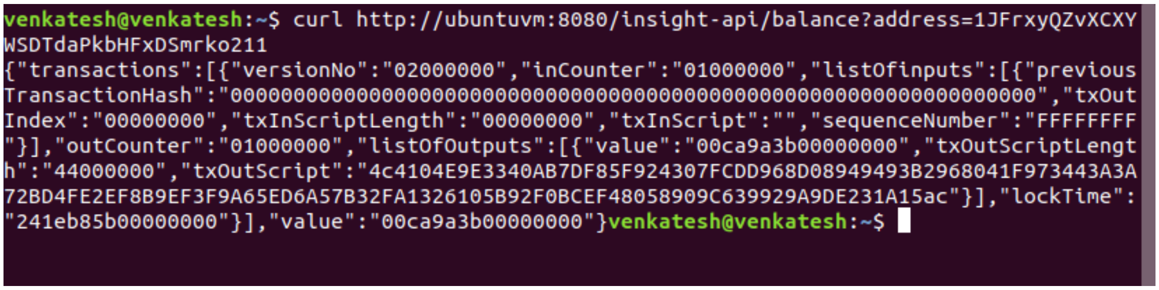


Figure 18: Fake coins added to the chain

Summary

This chapter talks about the results of the application. In the previous chapter, we talked about details of the implementation and some part of the code. In this section, we looked at details of running the application and how to perform the mining operation. This chapter also talks about the troubles that were faced during the implementation of the application and how they were overcome during the development of the application.

Chapter V: Conclusions and Future Work

Introduction

This chapter talks about the overall summary of the paper. In the previous chapter, we looked at the results of the application used to create the fake cryptocurrency. It focused mainly about the results of the mining operation and how to obtain the value of the fake coins that were added by using the application. This chapter mainly focusses on the concluding the entire research and putting forth the difficulty of adding fake coins to the main bitcoin network. This chapter also puts forth the future work that can be done to improve the bitcoin protocol to avoid the attacks that are proposed by this research work.

Discussion

This paper talks about cryptocurrencies, the technology they depend on, the mechanism which they to use of work, the different consensus mechanisms that are involved in each of those cryptocurrencies. This paper also dives into the implementation details of Proof of work and Proof of stake consensus mechanisms, how they work and the effectiveness of the algorithms. The paper introduces the basic concepts of Blockchains and describes how blockchain is linked together. It talks about the SHA-256 algorithm and the different steps involved in hashing. It talks about bitcoin in detail starting at transactions, different types of transactions, the fields involved in a transaction, how a transaction is included in a bitcoin block, it talks about hash merkle root of a bitcoin block, and how it can be generated. Finally, it introduces a mechanism which can be used to insert a fake transaction into the bitcoin blockchain there by producing fake bitcoins.

Conclusions

Miners are responsible for maintaining the integrity of the bitcoin network and the blockchain itself. Miners validate transactions, verify new blocks added to the bitcoin network thereby defending the blockchain against attackers. But in a situation where a miner commits fraud may not be much serious trouble but if the miner possesses more computational power than the rest of the miners put together, the miner might be able to cause much damage to the chain as demonstrated by this research. Though it is very unlikely to happen even by using super computers, this is still a vulnerability waiting to be exploited.

Future Work

Blockchain and cryptocurrency technologies are brand new and there is lots of research that can be done to improve these networks against attacks. Without miners, these networks would cease to exist. The incentive in the form of coinbase transactions is the main cause driving the miners towards mining but there is a limit to the number of coins that can be mined by the miners. When this incentive is gone, miners may not be interested anymore in mining to validate the transactions. Without miners, the cryptocurrency network is open to attackers and the entire network might need a reset (Happened twice in case of bitcoin test net). The future work related to this paper would be to introduce safety mechanisms so that these kinds of problems can be overcome.

References

Ahram T., Sargolzaei A., & Sargolzaei S. (2017, August). Blockchain technology innovations. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7998367/

Bag S, Ruj S, & Sakurai K. (2016, November). Bitcoin block withholding attack. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7728010/

Farell R. (2015, May). An analysis of the cryptocurrency industry. Retrieved from repository.upenn.edu: https://repository.upenn.edu/cgi/viewcontent.cgi?article=1133&context=wharton_research_scholars

Glazer P. (2014, March). An overview of cryptocurrency consensus algorithms. Retrieved from hackernoon.com: https://hackernoon.com/an-overview-of-cryptocurrency-consensus-algorithms-9d744289378f

Griffith V. (2017, October). Casper the friendly finality gadget. Retrieved from arxiv.org: https://arxiv.org/abs/1710.09437

Higgins S. (2017, December). Bitcoin's historic 2017 price run revisited. Retrieved from coindesk.com: https://www.coindesk.com/900-20000-bitcoins-historic-2017-price-run-revisited/

Kaushal P, K Bagga D. A, & Sobti D. R. (2017, July). Evolution of bitcoin and security risk in bitcoin. Retrieved from ieeexplore.ieee.org: http://ieeexplore.ieee.org/document/8003959/?reload=true

Michael Bedford Taylor (2017, September). The evolution of bitcoin hardware. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/8048662/

Mingxiao D., Xiaofeng M, & Zhe Z. (2017, December 1). A review on consensus algorithm of blockchain. Retrieved from ieeexplore.ieee.org: http://ieeexplore.ieee.org/document/8123011/

Mukhopadhyay U, Skjellum A, & Hambolu O. (2017, April). A brief survey of cryptocurrency systems. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7906988/

Nakamoto S. (2008). Bitcoin: a peer-to-peer electronic cash system. Retrieved from bitcoin.org: https://bitcoin.org/bitcoin.pdf

Nayak K, Kumar S, & Miller A. (2016, May). Stubborn mining: generalizing selfish mining and combining with an eclipse attack. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7467362/

Singh S., & Singh N. (2017, May). Blockchain: future of financial and cyber security. Retrieved from ieeexplore.ieee.org: http://ieeexplore.ieee.org/document/7918009/

Sleiman M. D., Lauf A. P., & Yampolskiy R. (2016, February). Bitcoin message: data insertion on a proof-of-work cryptocurrency system. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7398436/

Tosh D. K., Shetty S., & Liang X. (2018, January). Consensus protocols for blockchain-based data provenance: challenges and opportunities . Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/8249088/versions

Tseng L. (2017, May). Bitcoin's consistency property. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/7920619

Warren J. (2012, November). Bitmessage: a peer-to-peer message authentication and delivery System. Retrieved from bitmessage.org: https://bitmessage.org/bitmessage.pdf

Who is accepting bitcoin. (n.d.). Retrieved from coinreport.net: https://coinreport.net/coin-101/accepting-bitcoin/

Yu X., Shiwen M. T., & Li Y. (2017, October 19). Fair deposits against double-spending for bitcoin transactions. Retrieved from ieeexplore.ieee.org: http://ieeexplore.ieee.org/document/8073796/

Zheng Z., Xie S., & Dai H. (2017, September). An overview of blockchain technology: architecture, consensus, and future trends. Retrieved from ieeexplore.ieee.org: https://ieeexplore.ieee.org/document/8029379

Appendix

Application.java

```
package org.bitcoin;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.google.common.collect.Lists;

import io.vavr.control.Try;

import java.util.List;

import lombok.extern.slf4j.Slf4j;

import okhttp3.OkHttpClient;

import org.bitcoin.client.TransactionClient;

import org.bitcoin.model.Transaction;

import org.bitcoin.service.MinerUtil;

import org.bouncycastle.util.encoders.Hex;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;
```

```
import org.springframework.context.annotation.Configuration;

@Configuration

@EnableAutoConfiguration

@ComponentScan(basePackages = {"org.bitcoin"})

@Slf4j

public class Miner {

    public static void main(String[] args) {

        ApplicationContext ctx

                = new AnnotationConfigApplicationContext(Miner.class);

        MinerUtil fraudMiner = ctx.getBean(MinerUtil.class);

        fraudMiner.startMiners();

    }

    @Bean

    public OkHttpClient okHttpClient() {

        return new OkHttpClient();

    }

    @Bean

    public ObjectMapper getObjectMapper() {
```

```
        return new ObjectMapper();
    }
}
```

MinerUtil.java

```
package org.bitcoin.service;
import com.google.common.collect.Lists;
import java.io.UnsupportedEncodingException;
import java.nio.*;
import java.security.*;
import java.time.Instant;
import java.util.*;
import java.util.stream.*;
import javax.annotation.PostConstruct;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.tuple.*;
import org.bitcoin.client.*;
import org.bitcoin.model.*;
import org.bitcoin.util.HexUtil;
```

```
import org.bitcoinj.core.Base58;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

import org.bouncycastle.util.encoders.Hex;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Slf4j

@Component

public class MinerUtil {

    private Boolean miningStarted;

    private Thread currentMiningThread;

    private final TransactionClient transactionClient;

    private final TargetClient targetClient;

    private final BlockClient blockClient;

    private List<Pair<String, String>> keys;

    @Autowired

    public MinerUtil(final TransactionClient transactionClient, final TargetClient
targetClient, final BlockClient blockClient) {

        this.transactionClient = transactionClient;
```

```
        this.targetClient = targetClient;
        this.blockClient = blockClient;
        keys = Lists.newArrayList();
    }
    @PostConstruct
    private void generateKeysAndAddress() throws InvalidAlgorithmParameterException,
NoSuchAlgorithmException, NoSuchProviderException,
UnsupportedEncodingException {
        for (Integer I = 1; I <= 2; I++) {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC");
            ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
            keyGen.initialize(ecSpec);
            KeyPair kp = keyGen.generateKeyPair();
            PublicKey pub = kp.getPublic();
            PrivateKey pvt = kp.getPrivate();
            ECPrivateKey epvt = (ECPrivateKey) pvt;
            String privateKey = adjustTo64(epvt.getS().toString(16)).toUpperCase();
            ECPublicKey epub = (ECPublicKey) pub;
```

```
ECPoint pt = epub.getW();

String sx = adjustTo64(pt.getAffineX().toString(16)).toUpperCase();

String sy = adjustTo64(pt.getAffineY().toString(16)).toUpperCase();

String publicKey = "04" + sx + sy;

keys.add(ImmutablePair.of(privateKey, publicKey));

MessageDigest sha = MessageDigest.getInstance("SHA-256");

byte[] s1 = sha.digest(publicKey.getBytes("UTF-8"));

Security.addProvider(new BouncyCastleProvider());

MessageDigest rmd = MessageDigest.getInstance("RipeMD160", "BC");

byte[] r1 = rmd.digest(s1);

byte[] r2 = new byte[r1.length + 1];

r2[0] = 0;

for (int i = 0; i < r1.length; i++) {

    r2[i + 1] = r1[i];

}

byte[] s2 = sha.digest(r2);

byte[] s3 = sha.digest(s2);

byte[] a1 = new byte[25];
```

```
        for (int i = 0; i < r2.length; i++) {
            a1[i] = r2[i];
        }
        for (int i = 0; i < 5; i++) {
            a1[20 + i] = s3[i];
        }
        log.info("address" + l + "={}", Base58.encode(a1));
    }
}
static private String adjustTo64(String s) {
    switch (s.length()) {
        case 62:
            return "00" + s;
        case 63:
            return "0" + s;
        case 64:
            return s;
        default:
```

```
            throw new IllegalArgumentException("not a valid key: " + s);
        }
    }
    private Block getBlockData() {
        List<Transaction> transactions = transactionClient.getPendingTransactions()
                .onFailure(exception -> log.error("failed to get transactions from node",
exception))
                .getOrElse(Lists.newArrayList())
                .stream().limit(20).collect(Collectors.toList());
        String targetValue = targetClient.getCurrentTarget()
                .onFailure(exception -> log.error("failed to get current target from node",
exception));
        Block lastBlock = blockClient.getLastBlock()
                .onFailure(exception -> log.error("failed to get current target from node",
exception))
                .getOrElse(Block.builder().build());
        Block block = Block.builder()
                .magicNumber("D9B4BEF9")
```

```
            .blockHeader(BlockHeader.builder()
.hashPrevBlock(lastBlock.getBlockHeader().calculateHash().getOrElse(""))
                  .target(targetValue)
.time(HexUtil.formatLongAndReverseOrdering(Instant.now().getEpochSecond()))
                  .version("02000000")
                  .build())
            .build();
      block.setTransactions(transactions);
      return block;
   }
   private void insertCoinbaseTransaction(Block block) {
      Pair<String, String> keyPair = keys.get(0);
      String privateKey = keyPair.getLeft();
      String publicKey = keyPair.getRight();
      String outScript = "4c" + String.format("%02x", (publicKey.length() / 2)) + publicKey
+ "ac";
      String outputScriptLength =
HexUtil.formatIntegerAndReverseOrdering(outScript.length() / 2);
```

```
        String lockTime =
HexUtil.formatLongAndReverseOrdering(Instant.now().getEpochSecond());
        block.getTransactions().add(Transaction.builder()
                .inCounter("01000000")
                .listOfinputs(Collections.singletonList(TxIn.builder()
.previousTransactionHash("000000000000000000000000000000000000000000000000
0000000000000000")
                        .txOutIndex("00000000")
                        .txInScript("")
                        .txInScriptLength("00000000")
                        .sequenceNumber("FFFFFFFF")
                        .build()))
                .outCounter("01000000")
                .listOfOutputs(
                        Collections.singletonList(
                                TxOut.builder()
                                        .txOutScript(outScript)
.value(HexUtil.formatLongAndReverseOrdering(5000000000L))
```

```
                    .txOutScriptLength(outputScriptLength)
                    .build()))
            .versionNo("02000000")
            .lockTime(lockTime)
            .build());
block.setTransactionCounter(HexUtil.formatIntegerAndReverseOrdering(block.getTrans
actions().size()));
  }
  private void insertFakeTransaction(Block block) {
    Pair<String, String> keyPair = keys.get(1);
    String privateKey = keyPair.getLeft();
    String publicKey = keyPair.getRight();
    String outScript = "4c" + String.format("%02x", (publicKey.length() / 2)) + publicKey
+ "ac";
    String outputScriptLength =
HexUtil.formatIntegerAndReverseOrdering(outScript.length() / 2);
    String lockTime =
HexUtil.formatLongAndReverseOrdering(Instant.now().getEpochSecond());
    block.getTransactions().add(Transaction.builder()
```

```
        .inCounter("01000000")

        .listOfinputs(Collections.singletonList(TxIn.builder()

.previousTransactionHash("0000000000000000000000000000000000000000000000
0000000000000000")

                .txOutIndex("00000000")

                .txInScript("")

                .txInScriptLength("00000000")

                .sequenceNumber("FFFFFFFF")

                .build()))

        .outCounter("01000000")

        .listOfOutputs(

                Collections.singletonList(

                        TxOut.builder()

                                .txOutScript(outScript)


.value(HexUtil.formatLongAndReverseOrdering(1000000000L))

                                .txOutScriptLength(outputScriptLength)

                                .build()))
```

```
                .versionNo("02000000")

                .lockTime(lockTime)

                .build());


block.setTransactionCounter(HexUtil.formatIntegerAndReverseOrdering(block.getTrans
actions().size()));

    }

    public void startMiners() {

        new Thread(() -> {

            for (int i = 1; i <= 4; i++) {

                Block block = getBlockData();

                insertCoinbaseTransaction(block);

                if (i == 1) {

                    insertFakeTransaction(block);

                }

                String target = block.getBlockHeader().getTarget();

                for (Long nonce = 0L; nonce < Long.MAX_VALUE; nonce++) {
```

```
block.getBlockHeader().setNonce(HexUtil.formatLongAndReverseOrdering(nonce));
                    String hash = block.getBlockHeader().calculateHash().get();
                    if (target.compareTo(hash) > 0) {
                        log.info("nonce={} hash={} target={}", nonce, hash, target);
                        break;
                    }
                }
                if (blockClient.postBlock(block).get()) {
                    log.info("posted block={}", block);
                }
            }
        }).start();
    }
}
```

HexUtil.java

```
package org.bitcoin.util;
import java.nio.ByteBuffer;
```

```
import java.nio.ByteOrder;

import org.bouncycastle.util.encoders.Hex;

import org.spongycastle.util.Arrays;

public class HexUtil {

    public static String reverseByteOrdering(String value) {

        ByteBuffer buffer = ByteBuffer.allocate(value.length() / 2);

        buffer.order(ByteOrder.LITTLE_ENDIAN);

        buffer.put(Arrays.reverse(Hex.decode(value)));

        return new String(Hex.encode(buffer.array()));

    }

    public static long reverseOrderingAndParseLong(String value) {

        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);

        buffer.order(ByteOrder.LITTLE_ENDIAN);

        buffer.put(Arrays.reverse(Hex.decode(value)));

        return Long.parseLong(new String(Hex.encode(buffer.array())).substring(0, 8), 16);

    }

    public static String formatIntegerAndReverseOrdering(Integer value) {

        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
```

```
        buffer.order(ByteOrder.LITTLE_ENDIAN);

        buffer.putInt(value);

        return new String(Hex.encode(buffer.array())).substring(0, 8);

    }

    public static String formatLongAndReverseOrdering(Long value) {

        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);

        buffer.order(ByteOrder.LITTLE_ENDIAN);

        buffer.putLong(value);

        return new String(Hex.encode(buffer.array()));

    }

}
```

Block.java

```
package org.bitcoin.model;

import io.vavr.control.Try;

import java.security.MessageDigest;

import java.util.List;

import java.util.stream.Collectors;

import lombok.AllArgsConstructor;
```

```
import lombok.Builder;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;
import lombok.extern.slf4j.Slf4j;
import org.bitcoin.util.HexUtil;
import org.bouncycastle.util.encoders.Hex;
@ToString
@EqualsAndHashCode
@Builder
@Slf4j
@NoArgsConstructor
@AllArgsConstructor
public class Block {
    @Getter
    @Setter
```

```
    private String magicNumber;

    @Getter

    @Setter

    private String blockSize;

    @Getter

    @Setter

    private String transactionCounter;

    @Getter

    private List<Transaction> transactions;

    @Getter

    @Setter

    private BlockHeader blockHeader;

    public void setTransactions(List<Transaction> transactions) {

        this.transactions = transactions;

        this.transactionCounter =
HexUtil.formatIntegerAndReverseOrdering(transactions.size());

        if (this.blockHeader == null) {

            this.blockHeader = BlockHeader.builder().build();
```

```
        }
        this.blockHeader.setHashMerkleRoot(calculateMerkleRoot().getOrElse(""));
    }
    public Try<String> calculateMerkleRoot() {
        return Try.of(() -> {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            List<String> hashes = this.transactions
                    .stream()
                    .map(transaction ->
HexUtil.reverseByteOrdering(transaction.calculateHash().get()))
                    .collect(Collectors.toList());
            while (hashes.size() > 1) {
                for (int i = 1; i < hashes.size(); i += 2) {
                    digest.reset();
                    String hash1 = HexUtil.reverseByteOrdering(hashes.get(i - 1));
                    String hash2 = HexUtil.reverseByteOrdering(hashes.get(i));
                    byte[] bytes = Hex.decode(hash1.concat(hash2));
                    String combinedHash = new String(Hex.encode(digest.digest(bytes)));
```

```
                hashes.remove(i);
                hashes.remove(i - 1);
                hashes.add(i - 1, HexUtil.reverseByteOrdering(combinedHash));
            }
        }
        return hashes.get(0);
    });
  }
}
```

Transaction.java

```
package org.bitcoin.model;
import com.google.common.collect.Lists;
import io.vavr.control.Try;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.security.MessageDigest;
import java.util.Collections;
import java.util.List;
```

```
import lombok.AllArgsConstructor;

import lombok.Builder;

import lombok.EqualsAndHashCode;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

import lombok.ToString;

import lombok.extern.slf4j.Slf4j;

import org.bitcoin.util.HexUtil;

import org.bouncycastle.util.encoders.Hex;

@ToString

@EqualsAndHashCode

@Getter

@Setter

@Builder

@AllArgsConstructor

@NoArgsConstructor

@Slf4j
```

```
public class Transaction {

    private String versionNo;

    private String inCounter;

    private List<TxIn> listOfinputs;

    private String outCounter;

    private List<TxOut> listOfOutputs;

    private String lockTime;

    public Try<String> calculateHash() {

        return Try.of(() -> {

            String data = "";

            data += HexUtil.reverseByteOrdering(this.versionNo);

            data += HexUtil.reverseByteOrdering(this.inCounter);

            for (TxIn in : this.listOfinputs) {

                data += in.toString();

            }

            data += HexUtil.reverseByteOrdering(this.outCounter);

            for (TxOut in : this.listOfOutputs) {

                data += in.toString();
```

```
        }
        data += HexUtil.reverseByteOrdering(this.lockTime);
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(data.getBytes());
        byte[] digest1 = digest.digest(hash);
        return HexUtil.reverseByteOrdering(new String(Hex.encode(digest1)));
      });
    }
}
```

TxIn.java

```
package org.bitcoin.model;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
```

```
import lombok.Setter;

import lombok.extern.slf4j.Slf4j;

import org.bitcoin.util.HexUtil;

import org.bouncycastle.util.encoders.Hex;

import org.spongycastle.util.Arrays;

@EqualsAndHashCode

@Getter

@Setter

@Builder

@Slf4j

@AllArgsConstructor

@NoArgsConstructor

public class TxIn {

    private String previousTransactionHash;

    private String txOutIndex;

    private String txInScriptLength;

    private String txInScript;

    private String sequenceNumber;
```

```
@Override
public String toString() {
    try {
        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
        buffer.order(ByteOrder.LITTLE_ENDIAN);
        StringBuilder data = new StringBuilder();
        data.append(previousTransactionHash);
        if (this.txOutIndex != null) {
            data.append(HexUtil.reverseByteOrdering(this.txOutIndex));
        }
        if (this.txInScriptLength != null) {
            data.append(HexUtil.reverseByteOrdering(this.txInScriptLength));
        }
        if (this.txInScript != null) {
            data.append(HexUtil.reverseByteOrdering(this.txInScript));
        }
        if (this.sequenceNumber != null) {
            data.append(HexUtil.reverseByteOrdering(this.sequenceNumber));
```

```
            }
            return data.toString();
        } catch (Exception e) {
            e.printStackTrace();
            throw e;
        }
    }
}
```

TxOut.java

```
package org.bitcoin.model;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
```

```
import lombok.extern.slf4j.Slf4j;

import org.bitcoin.util.HexUtil;

import org.bouncycastle.util.encoders.Hex;

import org.spongycastle.util.Arrays;

@EqualsAndHashCode

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

@Slf4j

public class TxOut {

    private String value;

    private String txOutScriptLength;

    private String txOutScript;

    @Override

    public String toString() {

        String data = "";
```

```
        data += HexUtil.reverseByteOrdering(this.value);
        data += HexUtil.reverseByteOrdering(this.txOutScriptLength);
        data += HexUtil.reverseByteOrdering(this.txOutScript);
        return data;
    }
}
```

BlockHeader.java

```
package org.bitcoin.model;
import com.google.common.collect.Lists;
import io.vavr.control.Try;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.security.MessageDigest;
import java.time.Instant;
import java.util.Collections;
import java.util.List;
import lombok.AllArgsConstructor;
import lombok.Builder;
```

```
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;
import lombok.extern.slf4j.Slf4j;
import org.bitcoin.util.HexUtil;
import org.bouncycastle.util.encoders.Hex;
@ToString
@EqualsAndHashCode
@Getter
@Setter
@Builder
@Slf4j
@NoArgsConstructor
@AllArgsConstructor
public class BlockHeader {
    private String version;
```

```
private String hashMerkleRoot;

private String hashPrevBlock;

private String time;

private String target;

private String nonce;

public Try<String> calculateHash() {

    return Try.of(() -> {

        String data = "";

        data += HexUtil.reverseByteOrdering(this.version);

        data += HexUtil.reverseByteOrdering(this.hashPrevBlock);

        data += HexUtil.reverseByteOrdering(this.hashMerkleRoot);

        data += HexUtil.reverseByteOrdering(this.time);

        data += HexUtil.reverseByteOrdering(this.nonce);

        MessageDigest digest = MessageDigest.getInstance("SHA-256");

        byte[] hash = digest.digest(Hex.decode(data));

        byte[] digest1 = digest.digest(hash);

        return HexUtil.reverseByteOrdering(new String(Hex.encode(digest1)));

    });
```

```
    }
}
```

BlockClient.java

```
package org.bitcoin.client;

import com.fasterxml.jackson.core.type.TypeReference;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;

import io.vavr.control.Try;

import lombok.extern.slf4j.Slf4j;

import okhttp3.MediaType;

import okhttp3.OkHttpClient;

import okhttp3.Request;

import okhttp3.RequestBody;

import okhttp3.Response;

import org.bitcoin.model.Block;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component
```

```
@Slf4j

public class BlockClient {

    private final OkHttpClient httpClient;

    private final ObjectMapper objectMapper;

    @Autowired

    public BlockClient(OkHttpClient httpClient, ObjectMapper objectMapper) {

        this.httpClient = httpClient;

        this.objectMapper = objectMapper;

    }

    public Try<Block> getLastBlock() {

        return Try.of(() -> {

            Request request = new Request.Builder()

                    .url("http://ubuntuvm:8080/insight-api/blocks/last")

                    .get()

                    .build();

            Response response = httpClient.newCall(request).execute();

            objectMapper.registerModule(new JavaTimeModule());

            String body = response.body().string();
```

}


```
            response.close();
            return objectMapper.readValue(body, new TypeReference<Block>() {
            });
        });
    }
    public Try<Boolean> postBlock(final Block block) {
        return Try.of(() -> {
            RequestBody body = RequestBody.create(MediaType.parse("application/json;
charset=utf-8"), objectMapper.writeValueAsString(block));
            Request request = new Request.Builder()
                    .url("http://ubuntuvm:8080/insight-api/blocks")
                    .post(body)
                    .build();
            Response response = httpClient.newCall(request).execute();
            response.close();
            return true;
        });
    }
```

```
}

TargetClient.java

package org.bitcoin.client;

import io.vavr.control.Try;

import okhttp3.OkHttpClient;

import okhttp3.Request;

import okhttp3.Response;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class TargetClient {

    private final OkHttpClient httpClient;

    @Autowired

    public TargetClient(OkHttpClient httpClient) {

        this.httpClient = httpClient;

    }

    public Try<String> getCurrentTarget() {

        return Try.of(() -> {
```

```
            Request request = new Request.Builder()
                    .url("http://ubuntuvm:8080/insight-api/target")
                    .get()
                    .build();
            Response response = httpClient.newCall(request).execute();
            String body = response.body().string();
            response.close();
            return body;
        });
    }
}
```

TransactionClient.java

```
package org.bitcoin.client;
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.vavr.control.Try;
import java.util.List;
import okhttp3.OkHttpClient;
```

```
import okhttp3.Request;

import okhttp3.Response;

import org.bitcoin.model.Transaction;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class TransactionClient {

    private final OkHttpClient httpClient;

    private final ObjectMapper objectMapper;

    @Autowired

    public TransactionClient(final OkHttpClient httpClient, final ObjectMapper
objectMapper) {

        this.httpClient = httpClient;

        this.objectMapper = objectMapper;

    }

    public Try<List<Transaction>> getPendingTransactions() {

        return Try.of(() -> {

            Request request = new Request.Builder()
```

```
                .url("http://ubuntuvm:8080/insight-api/transactions/pending")
                .get()
                .build();
            Response response = httpClient.newCall(request).execute();
            String body = response.body().string();
            response.close();
            return objectMapper.readValue(body, new TypeReference<List<Transaction>>()
{
            });
        });
    }
    public Try<List<Transaction>> listTransactions() {
        return Try.of(() -> {
            Request request = new Request.Builder()
                .url("http://ubuntuvm:8080/insight-api/transactions")
                .get()
                .build();
            Response response = httpClient.newCall(request).execute();
```

```
            String body = response.body().string();

            response.close();

            return objectMapper.readValue(body, new TypeReference<List<Transaction>>()
{

            });

        });

    }}
```