NEA Documentation

# Analysis

**Project Introduction**

My project is a blockchain data structure and network system which solves many issues. When you use a currency like British Pounds or US Dollars, you are trusting a third party. The supply and value of these currencies are heavily affected by the respective government’s own control and monetary policy decisions. These currencies are regulated and issued by the respective governments. This can be an issue as this means a central authority has power to influence money supply, interest rates, and any other aspects of the currency’s operation. This is called centralisation, when a currency is managed by a central authority like a government and its bank associates. These central authorities can print more money or withdraw money from circulation, controlling the currency’s inflation rates.

My proposed program solves this problem by getting rid of the idea of a central authority controlling a currency. Blockchain makes currency decentralised, removing the element of trust in a third party. This allows for currency to operate on a distributed ledger, not governed by a central authority. The concept of a distributed ledger allows for this because anyone who wants to make transactions can make transactions without the need for a central intermediary. The blockchain is essentially a public record of transactions, and there are nodes, which anyone can operate, that carry a copy of the public record of transactions. Nodes can create transactions and add the transaction to their copy of the blockchain, then broadcast it onto the blockchain network for all the other peer nodes to pick up on, and add the transactions to their copies of the blockchain individually, so that all nodes are up to date and have the same copy of the blockchain. This whole process is done without the need of a central authority to regulate any transactions. There are many intricacies to this, as it sounds as though a lot of trust is still required, since malicious nodes could ruin the integrity and security of the blockchain, but there is a lot of mathematics behind blockchain that make it near impossible to make transactions in other user’s names, make transactions outside one’s balance, change transactions once they are added to the blockchain and broadcasted to all other nodes on the network, and also allows for privacy of users, which I will go into later in this document

Research

There are many existing system’s that demonstrate blockchain technology, the two I have researched are Bitcoin and Ether, the cryptocurrency belonging to Ethereum. Bitcoin was created as a digital currency, for the store of value and a medium of exchange. The focus of bitcoin is to provide a decentralised and secure way to transfer and store value, in the form of bitcoin cryptocurrency. Ethereum is also blockchain technology, but it has a much wider application, I will be focusing on its cryptocurrency, Ether, which has the same purposes as Bitcoin but through slightly different methods.

Transactions

The Bitcoin network operates similarly to how I mentioned above, it is a network, where individual nodes have their own copies of the blockchain, and can add transactions to the blockchain. Bitcoin solves many issues of security and integrity that arise from having a decentralised system like this for transactions through a lot of different ways. Starting from the core, transactions. A node can create transactions but how does Bitcoin make sure people are making transactions in their own name? Bitcoin utilises the RSA encryption algorithm, this algorithm generates a public and private key pair that are mathematically linked. Each user on the network has a public and private key. A user is represented on the network and referenced by their wallet address, which is derived from the public key. (in simpler systems, the public key can be used to represent a user on the blockchain, but in Bitcoin the wallet address is the public key put into a more standardised format, as a wallet can make transactions with different cryptocurrencies, not just Bitcoin, and different blockchain systems may generate the public key in different ways, lengths and such. Also RSA typically returns the public key in hexadecimal or binary string format, which is difficult to type out for users). When a transaction is made by a user, they must authenticate the transaction using a digital signature. A transaction includes the public key or wallet address, the amount being send, the public key or wallet address of the recipient receiving the transferred cryptocurrency, and a digital signature, which is made with the private key of the sender, and the cryptographically hashed contents of the transaction. (Bitcoin and blockchains in general use cryptographic hashing, specifically SHA-256, often for data as it is computationally deemed irreversible) The transaction is then broadcasted to the network and is added to the mempool, short for memory pool, a list of unconfirmed transactions (a transaction is confirmed when it is incorporated into the blockchain for long enough – more on this later). When a node picks up on the broadcast to add it to the transaction pool, it verifies the transaction. The public key and private key are mathematically linked in a way, due to RSA encryption to generate the key pairs, such that you can use the public key to verify a digital signature using a verification algorithm. It goes like this, the node takes the transaction, hashes it, and applies the public key on the digital signature using the verification algorithm (think of it like the public key is undoing the effect the private key has on the hashed transaction, because they are like inverses of each other). The node then compares the result to the hashed transaction, and if they are the same, then the user is in hold of the private key. This verifies and authenticates a transaction and is called asymmetric encryption, the node now just has to validate the transaction by making sure the user has sufficient funds by searching their transaction history in the blockchain, since the blockchain stores all transactions too. This process assumes the user keeps their private key private, as the holder of the private key can create transactions in the name of the public key linked to it. Although they are mathematically linked, the private key cannot be derived from the public key without infeasible computational power, due to the nature of the RSA encryption algorithm of which the mathematics are explained in more detail later.

Mining (proof-of-work)

Once a node verifies and validates a transaction, they add it to the transaction pool and also broadcast it to all other nodes, for all other nodes to validate the transaction individually, so that they can add it to their individual transaction pools, since the network needs to maintain equality. Decentralisation shows heavily here, as all nodes need to validate the transaction for themselves, they cannot just trust another node since anyone can run a node. Once a transaction in in the transaction pool it can be picked up by a specialised node called a miner. This is where Bitcoin begins to differ from Ether. All blockchains have their own set of rules that each node needs to maintain for the distributed ledger system to work. How the actual blockchain works is that the blockchain is a list of blocks, where each block is linked to the block before it. Each block contains transaction data, and some other identifying metadata, this is called the block header. The block header is hashed, resulting in a number (Bitcoin uses the SHA-256 hashing function which returns a fixed length of 256 bits, as a hexadecimal string, no matter the input) this hash represents the block. The hash is then included in the next block. A block contains the hash of the previous block in its own hash calculation, linking the blocks together in a chronological fashion. A diagram of a blockchain

Description automatically generated

These contents of the block header are used to calculate the hash that represents the block

If you go back and change the contents of a block, for example its transaction data, you will have to recalculate the hash of every block after it, because if you change the transaction data of a block, the hash of the block changes, and this changes the hash of the next block since the hash of the next block uses the hash of the previous block to calculate its hash. This isn’t too computationally heavy, making it easy to rewrite history essentially, which is where Bitcoin’s Proof-of-work consensus comes into play. A miner node picks up transactions out of the transaction pool and creates a block with it. If a miner could create a block quickly, it would be easy to go back and rewrite the entire blockchain, but a miner cant create a block quickly because they have to solve a hash puzzle. There is a value in each block called the nonce (short for number only used once) and the blockchain network will have a difficulty target, a value which determines how difficult it is to solve the hash puzzle. The hash puzzle is essentially a certain amount of 0s the hexadecimal string of the block hash must begin with. There is no way to solve this other than guessing random values for the nonce which will keep changing the hash value, until the hash value of the block meets this requirement of 0s set by the difficulty target, this is the computationally expensive part of being a miner node. Once a miner meets the requirement, they are then allowed to create a block and add it onto the blockchain, and broadcast it to the network, allowing for other nodes to add the block onto their blockchain (after validating the block). In Bitcoin specifically, a miner node is incentivised to add blocks because creating blocks will reward you with a bitcoin reward, and transactions have their own transaction fees which are rewarded to the miner. Miners all compete for the same block height, so the process of mining restarts when a block is uploaded to the network, as that specific block height is now filled with a block.

Immutability of Blockchain (attacks on Blockchain)

A malicious node will attempt to go back in the blockchain and change the contents of a block, but doing that means they will have to remine every single block in the chain. The history of the blockchain is already set, attempting to remine old blocks due to changing their contents will create a fork in the chain. Imagine the chain of blocks, and a fork coming off the first block to be changed with a list of blocks ahead of it that have been newly created (they are the remined blocks). Another part of the consensus of bitcoin is that if there is a fork, the one with the most computational work put in it is accepted as the blockchain. This means that the malicious node will have to remine the blocks at a rate faster than the creation of all blocks by all the other nodes together. This is called a 51% attack, because for a malicious node to successfully win the fork, they need the majority of the computational power of all nodes on the network, because if they don’t have that, then the rate at which nodes are adding blocks onto the main chain will be too fast and the shorter fork prong will not be deemed as the real blockchain. If a malicious node successfully remines blocks fast enough, it will have been able to change transaction history, and is now able to double spend currency that has already been transferred. As talked about earlier, a transaction is deemed confirmed in the blockchain once it has a transaction count of 6, the transaction count is the depth of the transaction in the blockchain, how many blocks deep the transaction is in. Bitcoin says that after 6 blocks it is confirmed as a transaction because it is unlikely that it is going to be able to get changed. A reminder that what is going on in the copy of the blockchain held by one node is not instantly regarded as the overall blockchain, since all other nodes have their own copy and validate transactions and blocks broadcasted from other nodes before adding it to their own copy, increasing the security and integrity of the decentralised distributed ledger of the blockchain network. Bitcoin is decentralised but it has lost its meaning slightly because there are a few nodes that are majorly responsible for the mining computational power, and when a lot of transactions from users are broadcasted to a select few nodes so that they broadcast the information to all other nodes, it contributes to centralisation.

Efficient Data Storage

Another component of blockchain that both Bitcoin and Ether have in common is the use of a data structure called Merkle Trees. When peer nodes in the network receives a block broadcasted from a miner node, they must validate the block and make sure it follows the rules of the blockchain network, they validate its Proof-of-Work in Bitcoin (difficulty target set by then network is reached), and Proof-of-Stake in Ether, a different consensus algorithm. After they verify the proof of work, they may add a block to the chain. Transactions are stored in the block, but a block header also includes something called a Merkle Root. All transactions in a block are organised in a Merkle Tree, a binary tree data structure where the leaf nodes are occupied by all the transactions, and each node above is the hash of the concatenation of the previous two nodes, called child nodes. The parent nodes are then paired up and used as the child nodes for the next level of the binary tree, the same process of concatenation and hashing. Each level is half the length of the previous level from the leaf node up to the point where there is only one node remaining in the tree, called the root. In this context of trees this is called the Merkle Root, which represents all the transactions without showing any transactions. Merkle Trees can be used for many purposes, in the context of Bitcoin it allows for a few things. One is efficient verification that a certain transaction is included in a block. Instead of searching the entire dataset of transactions, parties can use a Merkle Proof to verify that a specific transaction is in the tree. A transaction is stored in a leaf node of the tree, and there are a set of nodes related to this transaction, they are the nodes directly concatenating with the child nodes originating from the transaction. A Merkle proof is the set of sibling nodes that are directly concatenated with the transaction or parent node of the transaction for the hash of future nodes. In short, the Merkle proof is the path of nodes from the transaction in question to the Merkle root. A diagram of a structure

Description automatically generatedblue – set of nodes in Merkle proof, green – transaction in question

The transaction is then proved after concatenating and hashing to reach Merkle root, it is proved if the Merkle root generated from the set of nodes in the Merkle proof is the same as the Merkle root included in the block, because it means the transaction was a part of the original tree structure.

This process to confirm inclusion of data in the tree is much more efficient than searching the whole tree, with a time complexity of log(n) compared to the time complexity of n from just searching the tree. The nodes in Merkle trees are hashed for the immutability of blockchain, because of the nature of hashing algorithms, when some data is tampered with, the Merkle root included in the block header will change, requiring the Merkle root to be recalculated and the block would have to be remined. To be specific, transactions are represented by the transaction ID, the hash of the contents of a transaction object, this is how transactions are referenced in the blockchain, for the immutability of the transactions.

Ethereum Differences (proof-of-stake)

Ethereum does things slightly differently, it does not use a Proof-of-Work consensus algorithm, (mining being a hash puzzle, nonce values, etc) but it uses a Proof-of-Stake consensus algorithm. A consensus algorithm is used to reach agreement on how blocks should be added and validated to the blockchain, so that all nodes on the network can operate in a decentralised fashion but still follow the same rules to maintain the distributed ledger. In proof of stake, validators are responsible for confirming and adding transactions to the blockchain. Who validates the block is based off which node holds the most stake in the network, the more cryptocurrency the wallet of a node holds, the higher stake they have, the higher the chances of being selected as the validator of the block. This stake is temporarily locked in a smart contract as collateral, meaning it cannot be moved or spent or withdrawn in any way until they leave the blockchain network. This prevents malicious nodes as malicious nodes are penalised for any malicious actions and behaviour, in the form of slashing. This refers to portions of their stake being confiscated as a penalty of malicious actions. The incentive for a node to increase their stake in the blockchain and create blocks as a validator is through transaction fees and block rewards for creating blocks in the form of cryptocurrency. Bitcoin also gives out these block rewards for miners mining blocks, bitcoin actually creates cryptocurrency in the form of these rewards, cryptocurrency is generated when a block is made, however bitcoin has a cap of 21 million BTC, so since the creation of bitcoin, the reward has been halving every 4 years (or every 210 thousand blocks) or so. This cap was placed to add scarcity onto bitcoin in the future. The block reward currency for mining currently in bitcoin is 6.25 bitcoins, which is currently valued very highly but due to the nature of mining, and how computationally expensive it is, there are some fine margins. It is computationally expensive to mine because Bitcoin adjusts the difficulty target such that a block is added every 10 or so minutes, this essentially makes changing the history in the blockchain impossible if the block is far back due to the time It would take in combination with the fork resolution methods bitcoin takes that I talked of earlier, and also the expense of mining itself.

Features I will be using

The features I will use from these existing systems include the proof-of-work consensus algorithm from Bitcoin. This defines the security measures in the blockchain and the rules that each node follows in order to maintain consistency among the nodes in the decisions behind their copy of the blockchain, such as under what conditions is a block added, how are transactions verified, so on. The entire consensus algorithm goes like this: users initiate transactions that include a digital signature so that the transaction can be verified by various parties. Transactions are broadcasted to the network, and collected into the Mempool as unconfirmed transactions. Miners compete for the same block height to solve the cryptographic puzzle, this is the proof-of-work part of the consensus algorithm, but I am making slight changes to this. Miners have free will to pick which transactions they will include in their block, therefore they will be mining different blocks. My program will be different, the network will broadcast a block with preselected transactions to all the miners, and the miners will compete to solve the has puzzle of this block after validating the block and it’s transactions. The miner that meets the difficulty target set by the network and gets a small enough hash number will then broadcast the block to the network which will broadcast the block to all the peer nodes on the network, and these nodes will individually validate the block by checking that it meets the difficulty target, and recalculating the Merkle root as to confirm the transactions haven’t been tampered with, leading to a different Merkle root than what was broadcasted by the network for the miners to mine the block. Each transaction is also individually validated, checking the digital signatures, this is also done much earlier by a node when a user broadcasts a transaction to a node, after validating the transaction the node broadcasts the transaction to the network for it to be added to the mempool (unconfirmed transaction list). Back to the block validation that a node performs when receiving a block broadcasted from the network after a miner node broadcasts said block to the network, a node will add the block to it’s copy of the blockchain, and update its mempool accordingly. The difficulty target of my network will be adjusted accordingly by the network such that a block is added every 10 minutes (it takes 10 minutes for one miner out of the group of active miners to mine the block). This is adjusted by taking into account how many miners are working to mine a block (how many miners have been broadcasted the block for mining). I will be using the same hashing algorithm that Bitcoin uses, SHA-256, which returns a fixed length 256 bit hexadecimal string, and I will be using the same public-private key pair algorithm, RSA encryption.

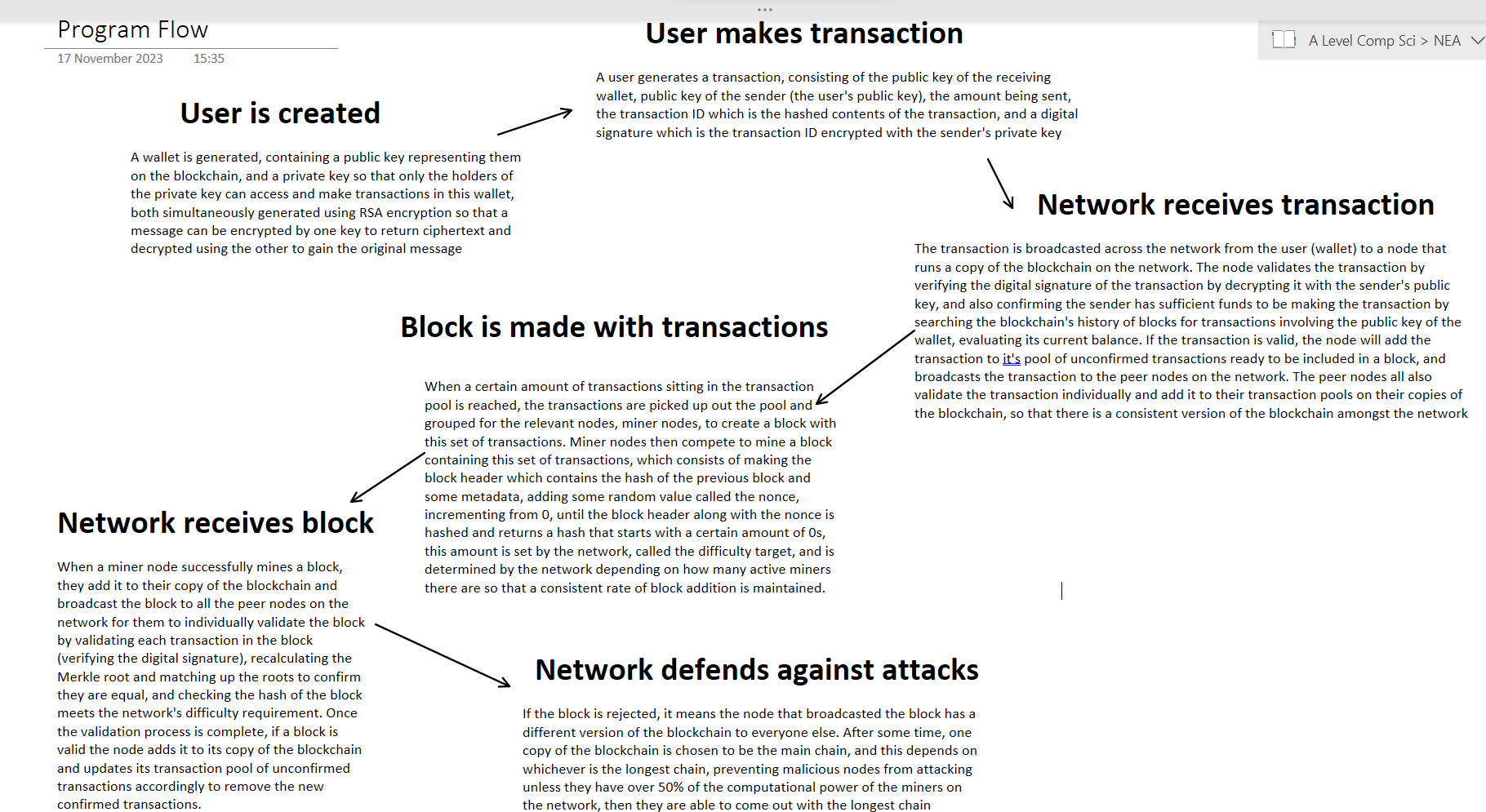
**Objectives**

1. RSA Class – Class Functions
2. RSA Class- Key Pair Generation
3. RSA Class - RSA Encryption & Decryption
4. Transaction Class – Transaction Structure (initialisation)
5. Transaction Class - Transaction Signing & Verifying Functions
6. Transaction Class and Wallet Class – Transaction History and Balance Checking
7. Transaction Class and Wallet Class – Sufficient Funds Validation
8. Transaction Class - Transaction Serialisation & Deserialization Functions
9. Network Communication Class – User to Node (to network to nodes later)
10. Transaction Class - Transaction Broadcasting Function
11. Blockchain Data Structure – Chain Mutation Functions
12. Blockchain Data Structure – Hard Coded Genesis Block
13. Blockchain Data Structure – Transaction Pool (Mem-pool)
14. Node Class – Validate Transaction
15. Node Class – Transaction Pool Functions
16. Network Communication – Peer Node List
17. Network Communication – Broadcast Transaction to Peer Nodes
18. Node Class – Broadcast Transaction to Network
19. Block Structure – Block Header
20. Block Structure – Block Header Hashing Function
21. Merkle Tree Data Structure, Tree Generation
22. Merkle Root Generation
23. Merkle Proof Generation
24. Block Structure – Merkle Root Calculation Function
25. Block Structure – Mining Recursion Function
26. Block Serialisation & Deserialization Functions
27. Child Class (Inherited from Node Class) Miner Node Class Functions
28. Network Communication – Block Template Generation & Broadcasting
29. Network Communication – Difficulty Target Adjusting Algorithm
30. Node Class & Blockchain Class – Received Difficulty Target Adjust
31. Miner Node Class – Broadcast Block to Blockchain
32. Node Class – Block Validation (Transaction Verifications)
33. Node Class – Block Validation (Proof-of-work Verification)
34. Network Communication Class – Block Broadcasting
35. Node Class – Broadcast Block to Network
36. Node Class – Fork Resolution
37. Node Class & Blockchain Class – Overspend Balance Deriving Function
38. Mining Block Reward + Fee Allocations
39. Blockchain Database (Saving Blockchain History)
40. Node Client Interface (Blockchain Visualisations

**Modelling**

Flow of the Program

The overall flow of using my system will go like this. A user is registered as on the blockchain once they have a wallet, which includes their wallet address (public key) and private key. These are generated by RSA encryption and are mathematically linked in such a way that a message can be encrypted with the private key and decrypted with the public key. In my blockchain context, this means that a transaction is hashed, and is ‘signed’ with the private key by the sender of the transaction using an RSA encryption operation. The result of this operation is the digital signature, which is included in in the transaction. The transaction is broadcasted to a node, as users typically will operate on a client which depends on the node, however you can broadcast directly to the network as a node. The node then verifies the transaction by decrypting the digital signature with the public key of the sender, which should return the hash of the transaction, called the transaction ID, if the private key used to sign the transaction and generate the digital signature is mathematically linked to the public key used to decrypt said digital signature, then the hash of the transaction is returned, verifying the authenticity of the sender (assuming the private key is kept secret). Once the transaction is verified by the node, it is added to it’s copy of the mempool and it is also broadcasted to the network, so that it is broadcasted to all the peer nodes for them to validate the transaction and add it to their copies of the blockchain individually. The network picks up transactions from it’s copy of the mempool and creates a block template with them, with some of the metadata filled out. This block is broadcasted to all the miner nodes, and they will compete to find the nonce value such that the resulting hash of the block’s contents paired with the nonce value is a small enough number to meet the difficulty target. Once a miner solves the hash puzzle by finding this value, they have successfully mined the block. At this point the miner node will broadcast the block to the network, broadcasting the block to all the peer nodes, who will validate the block individually to make sure it meets the consensus rules, and hasn’t had the transactions tampered with. If the validation is successful, the block is added to their copy of the blockchain, confirming the transactions. If not, the block is rejected and the transactions are returned to the transaction pool to be picked up again for later blocks. My blockchain program has several ways of dealing with malicious activity. If a block that has already been added to the block is altered, the hash of the block changes. This means the attacker must remine the block, and because the blockchain is essentially a linked list of hash pointers, every block onward must be remined. This is very computationally intensive, and even if successfully done, this will create a fork in the blockchain. What this means is one node will have a copy of the blockchain where from a certain block, all the blocks are different to a different copy of the blockchain on another node. When a malicious node remines a block it is broadcasting the block to all the other nodes, the nodes will accept the longest chain of the fork, the one with the most computational effort put in it. This means that in order for the malicious node to be successful, it will have to have at least 51% of the computational power of all the miners on the network, otherwise it will not be able to generate a longer chain in the fork compared to the main chain all the other non-malicious miners in the network are working on. This also prevents double-spending. Users spending more than they have is prevented by the program, when a user broadcasts a transaction a node validates the transaction also by checking the user’s history of transactions through the history of transactions on the blockchain, deriving the balance from there. If successfully validated, the transaction is added to the mempool and broadcasted to all other peer nodes for them to validate it. All objects being broadcasted through the network are serialised into a string, for later deserialization back to object form, allowing for more efficient network communication.

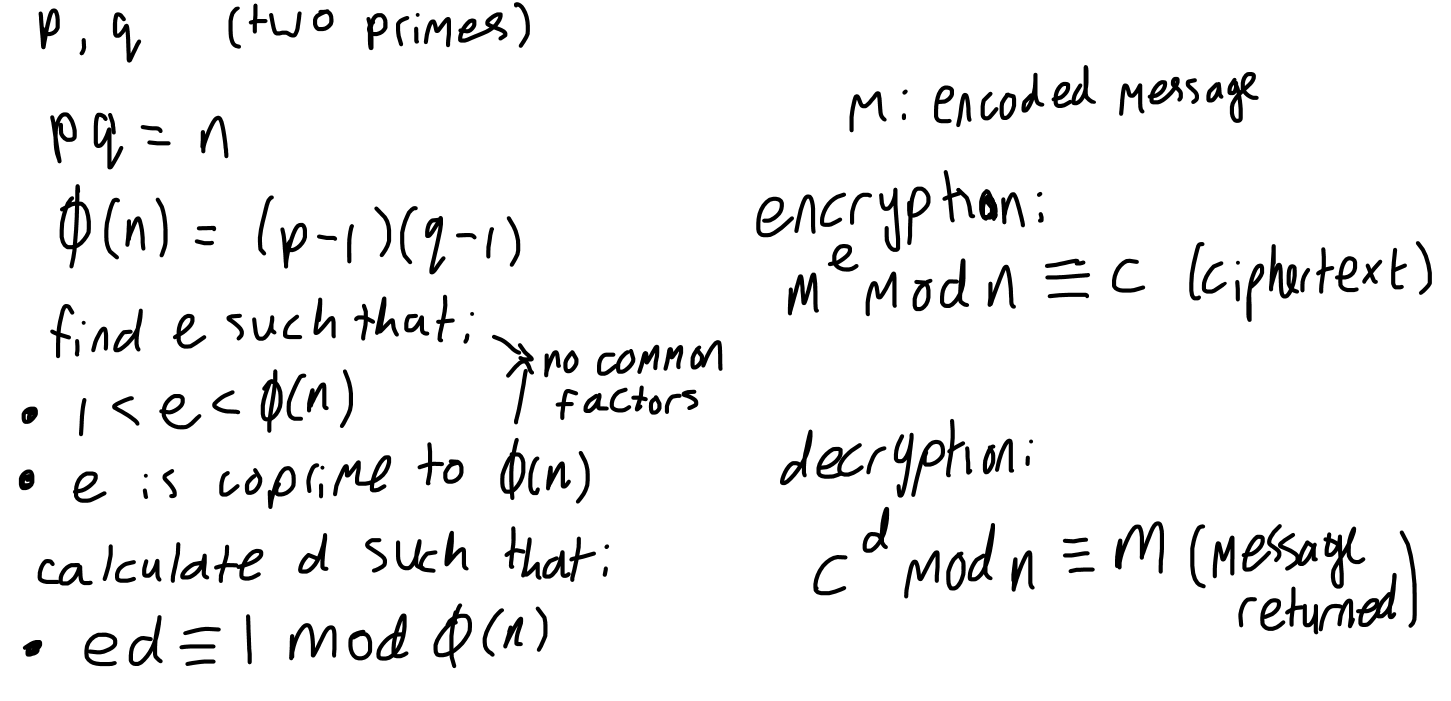


Back to the RSA encryption algorithm, it involves creating a public private key pair that are mathematically linked in such a way that if you encrypt a message with one key, you can decrypt it to gain back the original message from ciphertext with the other key. Typically in RSA encryption, the public key of the receiver is used to encrypt the message, returning ciphertext, and the receiver uses their private key to decrypt the message, this way no one can know what the message is without the private key. In blockchain technology though, RSA encryption is used the other way around, encrypting with the sender’s private key, and decrypting with the sender’s public key. This is because if the sender (creator of the transaction, transferring currency from their wallet) encrypts the transaction with their private key, the receiver of the transaction can then use the public key of the sender, which is public information included in the transaction, to decrypt. If decrypting returns the transaction information, it means that the sender is who they say they are as they have access to both the public and private key of the sender’s wallet, assuming all wallets keep their private key private. This process of encrypting is called signing the transaction, and the process of decrypting to check if it returns the transaction back is called verifying.

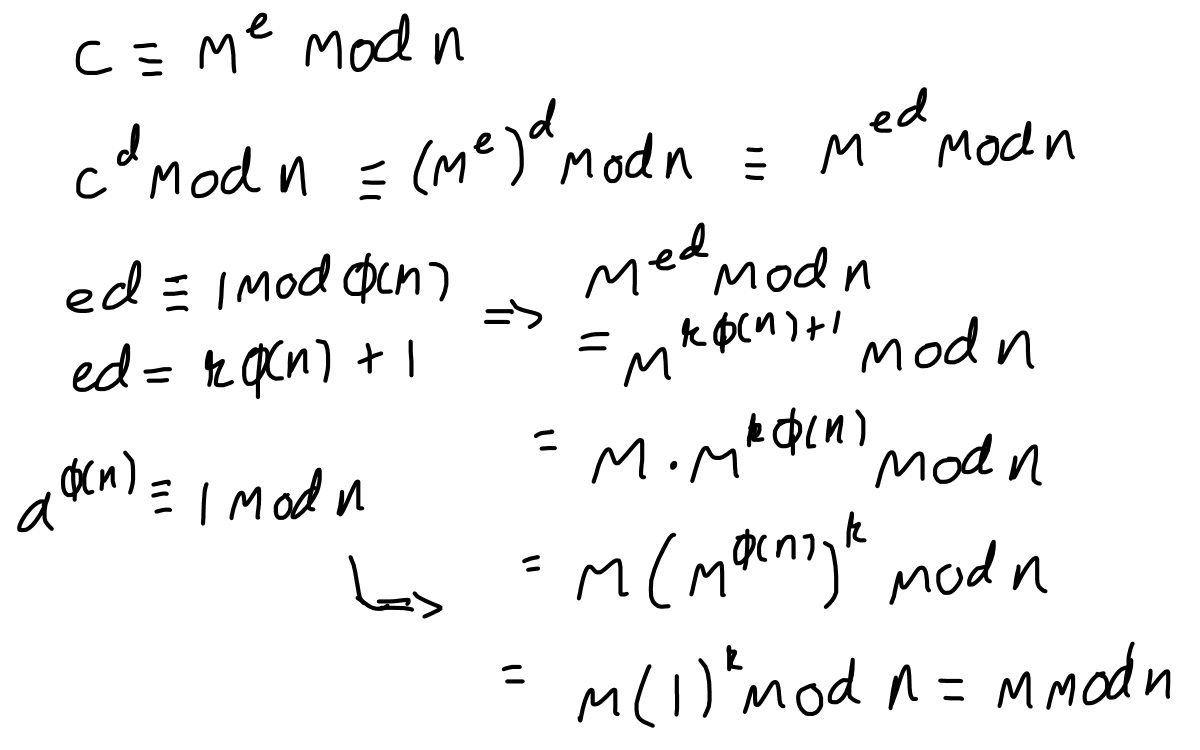
Now going into how the RSA encryption algorithm can generate these two mathematically linked keys, it uses a lot of modular arithmetic and prime numbers. The main underlying concept behind it is the fact that currently, we have a lot of methods to find prime numbers, but not efficient ways to break up a number into it’s prime factors. What this means is, given two primes p and q, it is easy to find n which is equal to pq. But given n, it is difficult to then find pq without knowing either p or q. The other important concept is the concept of modular arithmetic, which we use in our day to day lives ourselves, like for reading a clock. When our phone tells us its 15:00, we say that it is 3, a clock works on a mod 12 system, essentially any number after 12 wraps around the clock and assigns itself onto the congruent value, for example, 15 is 3 units past 12, so it is congruent to 3 mod 12. This value 12 is called the Modulus, it can be imagined as the wrap-around point, where when you reach a certain value you start essentially assigning the following values to the values from 1 to the modulus, like wrapping around the number line on a clock after it reaches the 12th hour. More on how this is relevant later. The algorithm starts with generating the key pair, such that encrypting with one means you can decrypt with the other.

Mathematics Behind RSA Encryption

Key generation: choose two large distinct prime numbers p and q, which can be done by generating a random large odd number, since all even numbers have a factor of 2 meaning they aren’t prime, and using a primality test on the random large number, like the Miller-Rabin test (RSA typically uses probable prime testing like the Miller-Rabin test, meaning there is a chance the test returns a false positive, declaring a number as a prime when it isn’t, so RSA encryption does multiple iterations of testing to ensure primality). Calculate the modulus n, which is the product of p and q. The encryption and decryption work by encoding the message as a number, and raising this number by another number called the public exponent, and doing this operation mod n to encrypt, then to decrypt, the ciphertext is raised to another number called the private exponent, mod n, which will return the original message. After calculating the modulus, n, calculate the totient function of n, represented by Phi(n) which is (p-1)(q-1) and find some value e, which will be our public exponent, and is bigger than 1, smaller than Phi(n), and is coprime to Phi(n), meaning it has no common factors with Phi(n). This value is public known information and is used to encrypt a message to return ciphertext by raising an encoded message to the power of this public exponent, e. Now the private exponent used to decrypt the ciphertext is derived from the public exponent. Calculate some value d such that the product of e and d is congruent to 1 mod Phi(n), meaning the product of e and d takeaway 1 is a multiple of Phi(n). Raising the a value to the power of e, then raising it to the power of d will just return back the original value. This works due to the properties of modular exponentiation and modular inverses. It is, computationally speaking, impossible to derive the private exponent from only knowing the public exponent and n, because it would require knowing Phi(n), and that would require knowing the individual primes that make up n, which as stated before, is computationally impossible to figure out given a large enough value of n, because while it is easy to find n (the product of two primes p and q) when you have p and q, there are no currently known methods to derive p and q given n, without just checking all the numbers that are prime from 1 to n. In RSA these primes are typically 300 digits long.



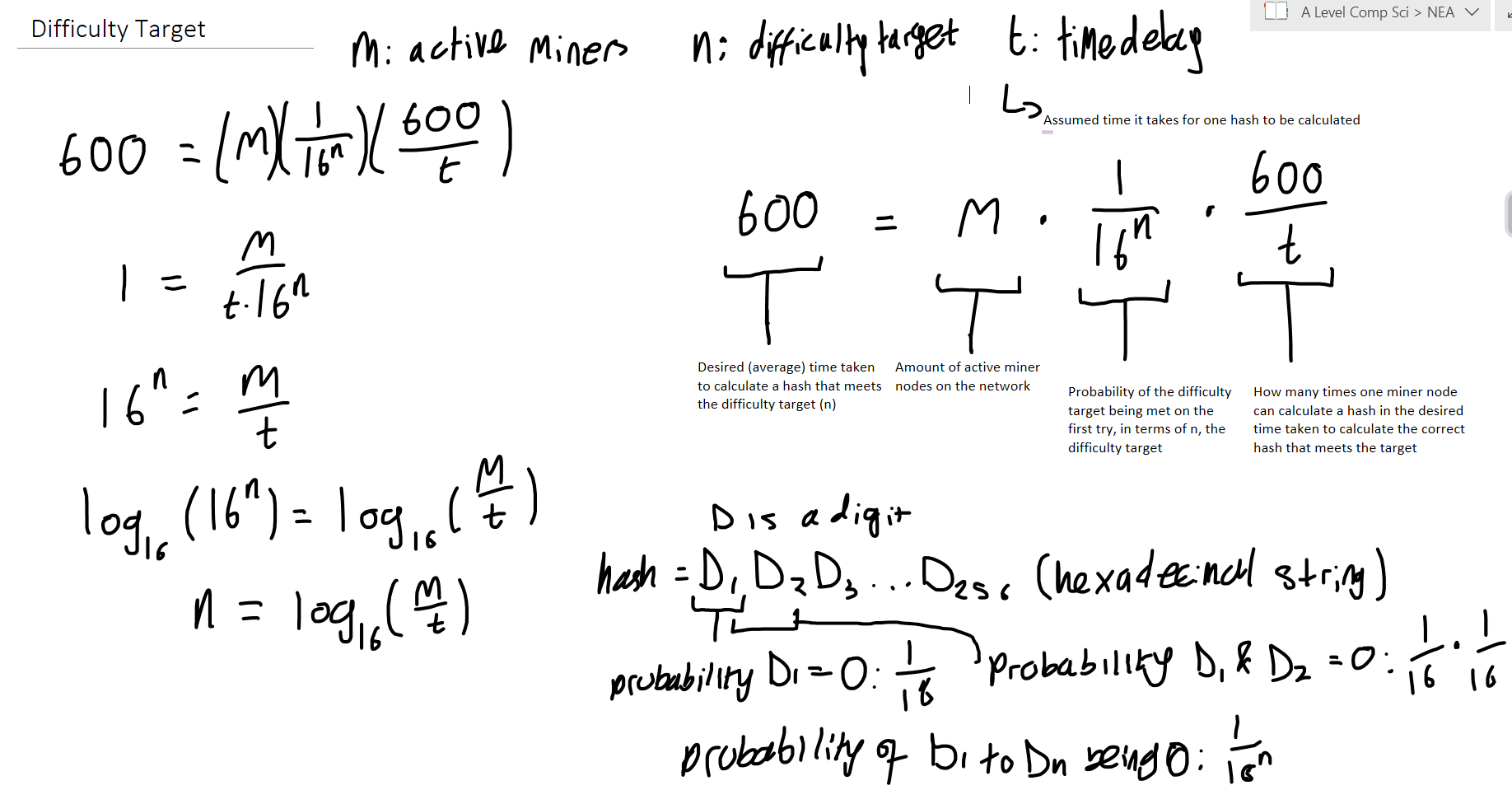
The reason this works is because of Fermat’s Little Theorem, which states that if p is a prime number, and a is an integer not divisible by p, then ap-1 1 mod p, which can be extended to Euler’s Totient Theorem, to work for composite numbers (like n being a composite of p and q) by considering the totient function Phi(n). The theorem is stated as follows, for any positive integer n and any integer a, coprime to n (meaning gcd(a, n) = 1, greatest common divisor) aPhi(n) 1 mod n, where Euler’s Totient Function Phi(n) represents the count of positive integers less than or equal to n that are coprime to n, or in the case of composite values of n made up of two distinct primes, Phi(n) = (p-1)(q-1). If we have some message m, and raise it to the public exponent e, modulus n, and then, raise the resulting ciphertext to the private exponent d, modulus n, it is equal to raising m to the power of e multiplied by d, modulus n, due to exponentiation laws and modular exponentiation laws. The product of the exponents is congruent to 1 mod Phi(n) which is a requirement and condition in the generation of the second exponent, meaning the product of the exponents = some multiple of Phi(n) + 1, so m is raised to the power of some multiple (k) of Phi(n) + 1, which is equal to m to the power of 1, which is just m, multiplied by m raised to the power of k x Phi(n), m raised to the power of k x Phi(n) is equal to m to the power of Phi(n) to the power of k, and as per Euler’s Totient Theorem, any integer to the power of Phi(n) is congruent to 1 mod n, so the resulting value is just m mod n



The primality test used to check primes when generating large primes is the Miller-Rabin Primality Test. The concept is based on Fermat’s Little Theorem which we used for Euler’s Totient Theorem earlier. If n is a prime number, then for most values of the integer a between 2 and n-2, this relation holds: . However the converse is not always true, meaning that if , n is not always prime, because there exists non prime values for which this relation holds for all a not divisible by n, so the Miller-Rabin test is used to increase confidence in this relation, through multiple witnesses of the constant a for different iterations of the test. The test goes as follows: write n-1 as (2r)(d) where r is the largest power of 2 that divides n-1, and d is an odd number. Then a random integer is picked for the constant a between 2 and n-2. Then x = ad mod n is computed. If x is congruent to 1 or x is congruent to n-1, then n may be prime. Now we continue to the next iteration. The following actions are repeated r-1 times: compute x is congruent to x2 mod n, if x is congruent to n-1, n may be prime (continue onto next iteration). If the congruency doesn’t hold for any iteration, the n is definitely not prime. This works because for choosing different random values of the constant a and repeating the test, the probability of incorrectly classifying a value as a prime number decreases exponentially, so with more tests, the confidence in a positive primality result increases.

Difficulty Target Calculation

Another part of math of this program is the difficulty target. The difficulty target represents a value, for example 5, and it means that the hash of the block must begin with 5 0’s. This is to control how long it takes for the active miners on the network to be able to mine a block, and bitcoin does it such that a new block is added every 10 minutes, which contributes to the immutability of the blockchain history as it takes a lot of time to mine a block even if you had the resources to overpower the rest of the miners and remine all the blocks, so if a block is far back enough in the blockchain it is infeasible for it to actually be mined as it would take to long to mine the rest of the blocks ahead of it. The math behind controlling it to a 10 minute pace is dependant on probability. The SHA-256 hashing algorithm will appear to be an essentially random 256-bit hexadecimal string, even though it isn’t random. This means we can approximate the probability of a single digit of the string being a specific value. Hexadecimal is base-16, so the odds of the first digit being a 0 if you hash once is 1 in 16. The odds of the next digit also being 0 alongside the first digit being 0 if you hash once is 1 in 162. For the all digits from the first to the n-th digit being 0 on the first attempt, the probability is 1 in 16n, which becomes very improbable very fast. The network will take into consideration how long it takes to generate a hash, which in my program, these hashes are calculated almost instantaneously, so I implemented a time delay between iterations of incrementing the nonce value to find the correct hash, and assumed the time it takes to generate one hash is this time delay (an alternative way to do this would be to look at hashes generated per second which would be more accurate to the specific machine, but isn’t as general to all miner nodes). Using the assumption of how long it takes to generate a hash, how many miner nodes are active (if there are m amount of miners then the probability of the correct hash being chosen on the first attempt is m times more likely), and that we want it to take them 600 seconds, we can use algebra to find what the difficulty target should be.



The desired time taken to calculate a hash that meets the difficulty target (in seconds) is the probability that the hexadecimal hash string starts with n-amount of 0s (the first n digits all being 0) multiplied by the amount of hashes one miner node can calculate in the desired time (where one hash takes t-amount of time to calculate) multiplied by the amount of miner nodes active on the network. This way we can rearrange for the difficulty target in terms of the amount of active miners on the network and the time it takes to generate a hash which is assumed to be the time delay I add.

Blockchain Database ERD – Not Finished

Data Saving and Loading – Not Finished

Prototype

The prototype has a few key differences to my main program. The prototype does not store the blockchain’s data on a database, meaning it is not saved. The prototype does not have a network system, only the functionality of a blockchain on one client, meaning nothing is broadcasted between different nodes, and essentially only one node can run at a time. There is no wallet address and the public key is directly used to represent users on the blockchain. Terminal responses to navigating actions on the blockchain instead of an external interface. There are no serialisation and deserialization methods compacting and reconstructing objects for broadcasting efficiency since there’s no broadcasting across a network in this prototype. The private key is stored in the class in this prototype whereas in the final product it will be securely stored or not stored at all anywhere. Validating blocks only consists of verifying all the digital signatures of the transactions and confirming that the difficulty target has been met, which is constant because there’s no network class in this prototype so no class to adjust the difficulty target based on active miners to keep a consistent rate of blocks coming in, the difficulty target is set to a low number for debugging purposes (makes mining times a lot shorter). The prototype does not include much exception handling, so the program will terminate because of errors, whereas in the final version, errors wont cause the program to terminate as it might be apart of the program like defending against malicious attacks should not lead to the program closing. In this prototype the transaction is validated when received by the blockchain from the user by making sure the user has the sufficient funds for this, which will be the same in the final design, but the method of checking is different in the prototype, each wallet will keep record of it’s transactions, and the blockchain will evaluate their balance based of that record of transactions, whereas in the final design the balance of a user is evaluated by searching the entire blockchain for transactions involving this user, which is a lot less efficient but guarantees the user is not spending more than they can. On the prototype the private key is stored in an attribute but in the final design the private key wont be stored anywhere or it will be stored securely using external password protection programs.

Where does the Prototype fit in with my final program

The prototype includes the core of the design, and is completely ready for use if all nodes on the network were to be on one computer, so not really a network. The final design will bring in the prototype with networking, allowing for the blockchain to be used for its intended purpose, a decentralised network for trust-less transactions between users across the network, the only things missing are anything related to broadcasting message between user to node and node to node, serialisation and deserialization of objects (where an object is deconstructed into a compact form and reconstructed back into an object) for more efficient transmission of messages over the network, and some other minor features outlined above.

What data does this program need to start

The only required data is information for the servers (nodes) to start up such as socket information

1    import hashlib

2    from datetime import datetime

3    import math

4    import random

5    import time

6    import sys

7

8    sys.setrecursionlimit(10 \*\* 6)  *# mining is done recursively and may have many tens of thousands of recursions*

9

10

11   class ExampleDataset:

12       *'''generate example datasets in place of transactions for testing'''*

13

14       def \_\_init\_\_(self, length):

15           self.dataset = []

16           self.length = length  *# desired length of example dataset*

17           self.data\_gen()

18

19       def get\_dataset(self):

20           return self.dataset

21

22       def data\_gen(self):  *# generate data (unique strings in place of transactions)*

23           for i in range((self.length + 1)):

24               string = f'Data{i}'

25               self.dataset.append(string)

26

27

28   class RSA:

29       *'''contains functions for implementing RSA encryption to generate a public-private key pair for wallets'''*

30

31       def \_\_init\_\_(self, key\_length=1024):

32           self.key\_length = key\_length  *# desired length of keys (longer keys are more computationally intensive to crack)*

33

34       def is\_prime(self, n, k=5):

35           *"""Miller-Rabin primality test."""*

36           if n <= 1 or n % 2 == 0:

37               return False

38           if n == 2 or n == 3:

39               return True

40

41           *# Write n as 2^r \* d + 1*

42           r, d = 0, n - 1

43           while d % 2 == 0:

44               r += 1

45               d //= 2

46

47           *# loop for trying different values of a*

48           for \_ in range(k):

49               a = random.randint(2, n - 2)

50               x = pow(a, d, n)

51               if x == 1 or x == n - 1:

52                   continue

53               for \_ in range(r - 1):

54                   x = pow(x, 2, n)

55                   if x == n - 1:

56                       break

57               else:

58                   return False

59           return True

60

61       def generate\_prime(self, bits):

62           *"""Generate a random prime number with the specified number of bits."""*

63           while True:  *# generate random numbers until the number passes Miller-Rabin primality test*

64               num = random.getrandbits(bits)

65               if self.is\_prime(num):

66                   return num

67

68       def egcd(self, a, b):

69           *"""Extended Euclidean Algorithm for finding modular inverses."""*

70           if a == 0:

71               return (b, 0, 1)

72           else:

73               g, x, y = self.egcd(b % a, a)

74               return (g, y - (b // a) \* x, x)

75

76       def modinv(self, a, m):

77           *"""Modular multiplicative inverse."""*

78           g, x, y = self.egcd(a, m)

79           if g != 1:

80               raise Exception('Modular inverse does not exist')

81           else:

82               return x % m

83

84       def generate\_keys(self):

85           p = self.generate\_prime(self.key\_length // 2)  *# Generate two large random prime numbers*

86           q = self.generate\_prime(self.key\_length // 2)

87           n = p \* q  *# Compute n (modulus)*

88           phi = (p - 1) \* (q - 1)  *# Compute totient (phi)*

89           e = 65537  *# Choose public exponent (65537 is a Commonly used value in RSA)*

90           d = self.modinv(e, phi)  *# Compute private exponent d*

91           public\_key = (e, n)  *# Public key (e, n)*

92           private\_key = (d, n)  *# Private key (d, n)*

93

94           return public\_key, private\_key

95

96       def encrypt(self, plaintext, d, n):

97           *'''encryption for signing transactions'''*

98           cipher\_text = [pow(ord(char), d, n) for char in plaintext]  *# pow function is exponentiation*

99           return cipher\_text

100

101      def decrypt(self, cipher\_text, e, n):

102          *'''decryption for verifying digital signatures'''*

103          plain\_text = ''.join([chr(pow(char, e, n)) for char in cipher\_text])  *# pow function is exponentiation*

104          return plain\_text

105

106

107  class Wallet:

108

109      def \_\_init\_\_(self):

110          self.public\_key = None

111          self.\_private\_key = None

112          self.transactions = []

113          self.\_balance = 0

114

115      def generate\_keypair(self):

116          *'''generate public and private key pair used to represent the user and sign transactions respectively'''*

117          self.public\_key, self.private\_key = RSA().generate\_keys()

118

119      def create\_transaction(self, recipient\_wallet, amount):

120          recipient\_pk = recipient\_wallet.reveal\_pk()

121          transaction = Transaction(self.public\_key, recipient\_pk, amount,

122                                    self.private\_key)  *# automatically signs transaction*

123          return transaction

124

125      def validate\_transaction(self, broadcaster, digital\_signature, transactionID):

126          *'''verify digital signature, authenticating the user'''*

127          broadcaster\_pk = broadcaster.public\_key

128          plain\_text = ''.join([chr(pow(char, broadcaster\_pk[0], broadcaster\_pk[1])) for char in

129                                digital\_signature])  *# pow function is exponentiation*

130          if plain\_text == transactionID:

131              return True

132          else:

133              return False

134

135      def evaluate\_balance(self):

136          *'''check record of transactions involving wallet and evaluate a final balance'''*

137          balance = 0

138          for transaction in self.transactions:

139              if transaction.sender\_pk == self.public\_key:  *# the amount from outgoing transactions is deducted from balance*

140                  balance -= transaction.amount

141              elif transaction.recipient\_pk == self.public\_key:  *# the amount from ingoing transactions is added to balance*

142                  balance += transaction.amount

143          self.\_balance = balance

144          return balance

145

146      def sufficient\_bal(self, amount):

147          *'''check if the user has the sufficient funds to make transaction'''*

148          balance = self.evaluate\_balance()

149          if balance >= amount:

150              return True

151          elif balance < amount:

152              return False

153

154      def add\_transaction(self, transaction\_obj):

155          self.transactions.append(transaction\_obj)

156

157      def identify\_pk(self, pk):

158          if self.public\_key == pk:

159              return self

160

161      def get\_bal(self):

162          return self.\_balance

163

164      def reveal\_pk(self):

165          return self.public\_key

166

167      *# check balance*

168

169

170  class Transaction():

171

172      def \_\_init\_\_(self, sender\_pk, recipient\_pk, amount, private\_key):

173          self.sender\_pk = sender\_pk

174          self.recipient\_pk = recipient\_pk

175          self.amount = amount

176          self.timestamp = datetime.now().strftime("%H:%M:%S")

177          self.transactionID = self.calculate\_transactionID()

178          self.digital\_signature = self.sign\_transaction(private\_key)

179

180      def calculate\_transactionID(self):

181          *'''Calculate hash of the transaction's contents to represent transaction when referenced on blockchain'''*

182          transaction\_data = f"{self.sender\_pk}{self.recipient\_pk}{self.amount}{self.timestamp}"

183          transactionID = hashlib.sha256(

184              transaction\_data.encode('utf-8')).hexdigest()  *# encoded -> hashed (binary) -> converted to hexadecimal*

185          return transactionID

186

187      def sign\_transaction(self, private\_key):  *# encryption and decryption mathematics explained in doc*

188          *'''encryption for signing transactions'''*

189          digital\_signature = [pow(ord(char), private\_key[0], private\_key[1]) for char in

190                               self.transactionID]  *# pow function is exponentiation*

191          return digital\_signature

192

193      def validate\_transaction(self):

194          *'''decryption for verifying digital signatures, and checking if user has sufficient funds'''*

195          *# decrypt the encrypted transaction ID and compare to transaction ID of the transaction to see if decryption worked (keys are linked)*

196          decryption = ''.join([chr(pow(char, self.sender\_pk[0], self.sender\_pk[1])) for char in

197                                self.digital\_signature])  *# pow function is exponentiation*

198          if decryption == self.transactionID:

199              return True

200          else:

201              return False

202

203      *# checking funds and updating records of wallets done through transaction class for ease of validation purposes*

204

205      def check\_funds(self, sender):

206          *'''check if the user has the sufficient funds to make transaction'''*

207          check = sender.sufficient\_bal(self.amount)

208          return check

209

210      def update\_records(self):

211          *'''update the list of transactions made for both sender and recipient by finding them through their public keys'''*

212          sender\_obj = Wallet.identify\_pk(self.sender\_pk)  *# find wallets of sender and recipient by checking*

213          receiver\_obj = Wallet.identify\_pk(self.recipient\_pk)

214

215          sender\_obj.add\_transaction(self)  *# update records of sender and recipient*

216          receiver\_obj.add\_transaction(self)

217

218      def \_\_repr\_\_(self):

219          return (

220              f"Transaction(sender\_pk={self.sender\_pk}, "

221              f"recipient\_pk={self.recipient\_pk}, "

222              f"amount={self.amount}, "

223              f"timestamp={self.timestamp}, "

224              f"transactionID={self.transactionID}, "

225              f"digital\_signature={self.digital\_signature})"

226          )

227

228

229  '''Wallet Generation & Transactions Testing'''

230

231  *# user / wallet generation*

232  Me = Wallet()

233  Me.generate\_keypair()

234  You = Wallet()

235  You.generate\_keypair()

236

237  *# transaction between users*

238  NewTransaction = Me.create\_transaction(You, 5)  *# sending transaction*

239  print(NewTransaction)  *# string representation of transaction (digital signature is very large)*

240

241  *# validating transaction*

242

243  print(NewTransaction.validate\_transaction())

244  print(NewTransaction.check\_funds(Me))  *# will return False as user has balance of 0*

245

246

247  class MerkleNode:

248      *'''represents one node made up of the hash of two concatenated child nodes'''*

249

250      def \_\_init\_\_(self, left\_node, right\_node,

251                   hash\_value):  *# tree is made by merkle nodes linking to eachother through attributes*

252          self.left\_node = left\_node

253          self.right\_node = right\_node

254          self.hash = hash\_value

255

256      def get\_hash(self):

257          return self.hash

258

259

260  class MerkleTree:

261

262      def \_\_init\_\_(self, dataset):

263          self.dataset = dataset

264          self.tree = self.build\_tree()

265          self.root = self.get\_root()

266

267      def calculate\_hash(self, left,

268                         right):  *# may be used to make leaf nodes (left and right are from dataset) or other nodes (L and R are hashes)*

269          *'''takes two elements, converts them to strings, concatenates them, and calculates the hash of this concatenation'''*

270          hash\_input = str(left) + str(right)

271          hashed = hashlib.sha256(hash\_input.encode('utf-8')).hexdigest()

272          return hashed

273

274      def build\_tree(self):

275          *'''builds the merkle tree of merkle nodes, providing a merkle root representing the hash of all nodes'''*

276          leaf\_nodes = []

277          *# add hashed dataset values into leaf level in string form*

278          for data in self.dataset:

279              hash\_input = str(data)  *# convert to string*

280              hashed\_data = hashlib.sha256(hash\_input.encode()).hexdigest()

281              leaf\_nodes.append(hashed\_data)

282

283          tree = [leaf\_nodes]

284          *# generate parent nodes from child nodes in previous level*

285          while len(tree[-1]) > 1:  *# generate next level until the root is reached  (level of length 1)*

286              parent\_nodes = []

287              for node in tree[-1][0:len(tree[-1]):2]:  *# tree[-1] is the current level of the tree*

288                  left\_node = node

289                  if left\_node != tree[-1][-1]:  *# if left node isnt the last node then there is a right node*

290                      right\_index = tree[-1].index(node) + 1

291                      right\_node = tree[-1][right\_index]

292                  else:

293                      right\_node = None

294                  parent\_hash = self.calculate\_hash(left\_node, right\_node)

295                  parent = MerkleNode(left\_node, right\_node, parent\_hash)

296                  parent\_nodes.append(parent.get\_hash())

297              tree.append(parent\_nodes)

298          return (tree)

299

300      def get\_root(self):

301          root = self.tree[-1][0]

302          return root

303

304      def merkle\_proof(self, target\_node):

305          *'''generates the sibling nodes that are in the path the target node takes to the root'''*

306          target\_node = hashlib.sha256(

307              str(target\_node).encode('utf-8')).hexdigest()  *# get target node into its leaf level form*

308          proof\_path = []

309          root\_reached = False

310          current\_level = 0  *# index of current level*

311          while root\_reached == False:  *# traverse tree from target node to root*

312              *# pick up sibling nodes during traversal and add to proof path*

313              for node in self.tree[current\_level][

314                          0:len(self.tree[current\_level]):2]:  *# look at every other node (first node of a pair)*

315                  left\_node = node

316                  if self.tree[current\_level][-1] != node:  *# if left node isnt last node in tree*

317                      right\_index = self.tree[current\_level].index(

318                          left\_node) + 1  *# one index after left node in the current level*

319                      right\_node = self.tree[current\_level][right\_index]

320                      *# check if target node is either of the nodes just defined in the pair*

321                  if left\_node == target\_node:

322                      proof\_path.append(right\_node)

323                      target\_node = self.calculate\_hash(left\_node,

324                                                        right\_node)  *# target node for next level (hash of child nodes)*

325                  elif right\_node == target\_node:

326                      proof\_path.append(left\_node)

327                      target\_node = self.calculate\_hash(left\_node,

328                                                        right\_node)  *# target node for next level (hash of child nodes)*

329              if len(self.tree[current\_level + 1]) == 1:  *# if the next level is the root*

330                  root\_reached = True  *# dont search next level (not needed for proof path)*

331              else:

332                  current\_level += 1  *# search next level*

333          return proof\_path

334

335      def verify\_proof(self, target\_node, proof):

336          *'''takes a proof path and reconstructs the root with it, comparing the roots to verify if the proof is valid, verifying the target node'''*

337          target\_node = hashlib.sha256(

338              str(target\_node).encode('utf-8')).hexdigest()  *# get target node into its leaf level form*

339          for node in proof:  *# contatenate and hash target node with proof node, concatenate and hash the previous hash with next proof node, so on*

340              current\_level = proof.index(

341                  node)  *# works because there is only one sibling node per level in the proof path*

342              if self.tree[current\_level].index(node) % 2 == 0:  *# all left childs of pairs have even node index in level*

343                  target\_node = self.calculate\_hash(node, target\_node)  *# node is left child*

344              elif self.tree[current\_level].index(

345                      node) % 2 == 1:  *# all right childs of pairs have odd node index in level*

346                  target\_node = self.calculate\_hash(target\_node, node)  *# node is right child*

347          if target\_node == self.root:  *# check if root generated from proof is equal to actual root*

348              return True

349          else:

350              return False

351

352

353  '''Merkle Tree Testing'''

354

355  dataset1 = ExampleDataset(16).get\_dataset()  *# generate example dataset*

356  tree1 = MerkleTree(dataset1)  *# generate merkle tree from example dataset*

357  print(tree1.tree)

358  proof = tree1.merkle\_proof("Data3")  *# generate proof path given a target node*

359  print(proof)

360  print(tree1.verify\_proof("Data3", proof))  *# verify that target node is in merkle tree through proof path*

361

362

363  class Block:

364      *'''basic structure of a block, block manipulation methods, block mining, block validation'''*

365

366      def \_\_init\_\_(self, transactions, blockchain):

367          self.transactions = transactions

368          self.block\_height = len(blockchain.get\_chain())  *# index of latest block + 1 in chain*

369          if blockchain.get\_chain() == []:  *# if blockchain is empty, create genesis block*

370              self.previous\_hash = 0  *# genesis block creation*

371          else:

372              self.previous\_hash = blockchain.get\_chain()[-1].get\_block\_hash()  *# block hash of last block in chain*

373          self.timestamp = datetime.now().strftime("%H:%M:%S")

374          self.merkle\_root = self.calculate\_merkle\_root()  *# used to check if a specific transaction is in the block efficiently (merkle proof)*

375          self.nonce = 0  *# incremented for mining*

376          self.difficulty\_target = blockchain.get\_difficulty\_target()

377          self.block\_header = f'''block\_height = {self.block\_height},

378                              previous\_hash = {self.previous\_hash},

379                              timestamp = {self.timestamp},

380                              merkle\_root = {self.merkle\_root},

381                              transactions = {self.transactions},

382                              difficulty\_target = {self.difficulty\_target}'''  *# ready format for hashing*

383          self.block\_hash = None

384

385      def calculate\_merkle\_root(self):

386          *'''calculate merkle root from transaction list'''*

387          this\_merkle\_tree = MerkleTree(self.transactions)

388          *# generate merkle tree and return merkle root*

389          return this\_merkle\_tree.get\_root()

390

391      def calculate\_block\_hash(self):

392          *'''take block header and hash it, if hash meets difficculty target, return, if not, increment nonce and repeat'''*

393          hash\_input = self.block\_header + str(self.nonce)

394          block\_hash = hashlib.sha256(hash\_input.encode("utf-8")).hexdigest()

395          while block\_hash[0: (

396          self.difficulty\_target)] != "0" \* self.difficulty\_target:  *# keep mining while difficulty target is not met*

397              print(self.nonce)

398              self.nonce += 1  *# increment nonce and mine again*

399              hash\_input = self.block\_header + str(self.nonce)

400              block\_hash = hashlib.sha256(hash\_input.encode("utf-8")).hexdigest()

401              print(block\_hash)

402          self.block\_hash = block\_hash

403

404      def is\_block\_valid(self):

405          *'''validate block by checking block hash meets difficulty target, and that each transaction is valid (verify each transaction in set)'''*

406          check = []

407          *# block header information and structure is correct*

408          hash\_portion = self.block\_hash[0: self.difficulty\_target]  *# check if hash meets difficulty target*

409          if str(hash\_portion) == "0" \* self.difficulty\_target:

410              check.append(True)

411          else:

412              check.append(False)

413

414          def validate\_transactions():  *# validate each transaction (verifying digital signatures)*

415              pass

416

417          *#     for transaction in self.transactions:*

418          *#         check.append(transaction.validate\_transaction())*

419          *# validate\_transactions()*

420          *# transaction double spending prevented (no duplicate transactions)*

421

422          print(f'is block valid: {all(check)}')  *# all() returns true if all elements are true*

423

424      def get\_block\_hash(self):

425          return self.block\_hash

426

427      def get\_transactions(self):

428          return self.transactions

429

430      def get\_block\_header(self):

431          return self.block\_header

432

433      def adjust\_difficulty(self, difficulty):

434          self.difficulty\_target = difficulty

435

436      def transaction\_check(self, transaction):  *# check if transaction is in the block efficiently (merkle proof)*

437          *'''check if a transaction is in a block using merkle proofs (verifying dataset has not been tampered with too)'''*

438          this\_merkle\_tree = MerkleTree(self.transactions)

439          proof\_path = this\_merkle\_tree.merkle\_proof(transaction)

440          verify = this\_merkle\_tree.verify\_proof(proof\_path, transaction)

441          print(f'is transaction in transactions: {verify}')

442

443

444  class Blockchain():

445      *'''the data structure that all nodes base their copy of the blockchain off, and manipulating incoming / outgoing messages of the network'''*

446

447      def \_\_init\_\_(self):

448          self.chain = []

449          self.transaction\_pool = []  *# unconfirmed, verified transactions*

450

451      def add\_transaction(self, transaction):

452          self.transaction\_pool.append(transaction)

453

454      def genesis\_block(self, issuance):  *# issuance is the first amount of currency the program starts with*

455          genesis\_block = Block(issuance,

456                                self)  *# generates the first currency on the program and has no previous hash so it must be hardcoded in*

457          return genesis\_block

458

459      def get\_chain(self):

460          return self.chain

461

462      def add\_block(self, block):

463          *'''adds a new block to the chain'''*

464          self.chain.append(block)

465

466      def get\_latest\_block(self):

467          *'''retrieves the latest block in the chain'''*

468          return self.chain[-1]

469

470      def mine\_block(self, block):

471          *# initiate mining process, solving hash puzzle (called by miner node)*

472          block.calculate\_block\_hash()

473          self.transaction\_pool = []  *# empty transaction pool*

474          return block

475

476      def confirm\_transaction(self, transaction):  *# constantly run by network for each transactions*

477          *# confirm inclusion of a transaction in a block by incrementing confirmation count for each block that is added after block of said transaction (6=confirmed)*

478          for block in self.chain[::-1]:

479              if transaction in block.get\_transactions():  *# searches for block containing transaction USE MERKLE PROOF INSTEAD*

480                  transaction\_depth = len(self.chain) - self.chain.index(

481                      block)  *# length from end of chain to block containing transaction*

482                  if transaction\_depth >= 6:

483                      return True

484          return False  *# returns false if required transaction depth has not reached*

485

486      def get\_difficulty\_target(self):

487          *# listen to difficulty target from network*

488          return 1  *# example for prototype*

489

490

491  '''Block & Blockchain Testing'''

492

493  *# blockchain and genesis block creation*

494  Blockchain1 = Blockchain()

495  genesis\_dataset = ExampleDataset(8).get\_dataset()

496  genesis\_block = Blockchain1.genesis\_block(genesis\_dataset)

497  Blockchain1.add\_block(genesis\_block)  *# create the first block and add it to blockchain*

498  print(Blockchain1.get\_chain())

499

500  *# block mining and creation*

501  ExDataset1 = ExampleDataset(8).get\_dataset()

502  block01 = Block(ExDataset1, Blockchain1)  *# create the block*

503  block01.calculate\_block\_hash()  *# mine the block*

504

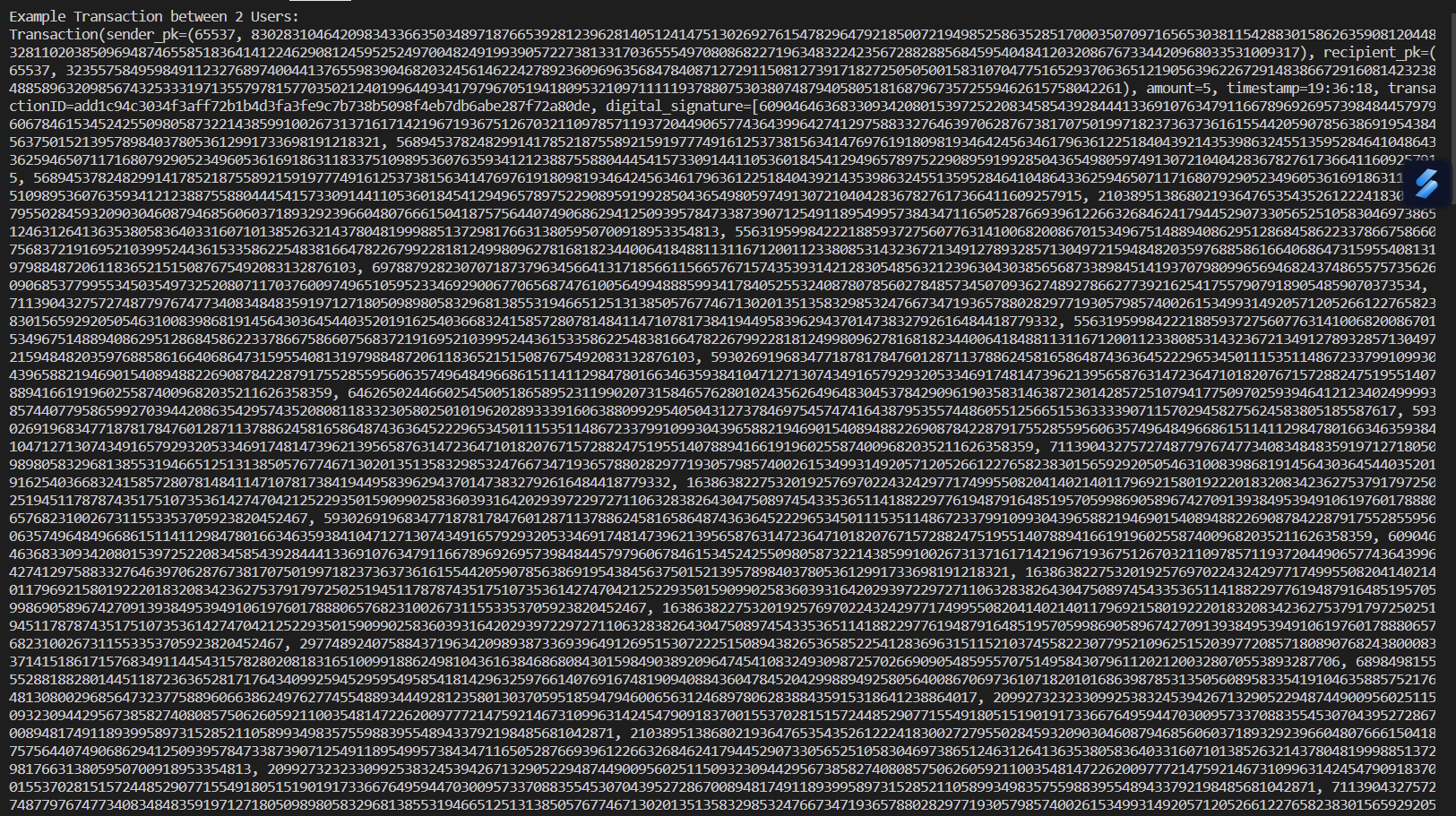
505  *# adding block to the blockchain*

506  print(block01.is\_block\_valid())  *# check validity*

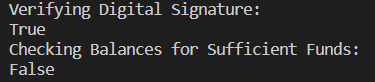
507  Blockchain1.add\_block(block01)  *# add block*

508  print(Blockchain1.get\_chain())  *# show chain*

Testing – Wallet Generation (RSA Testing), Transactions, Transaction Validation (Line 220)



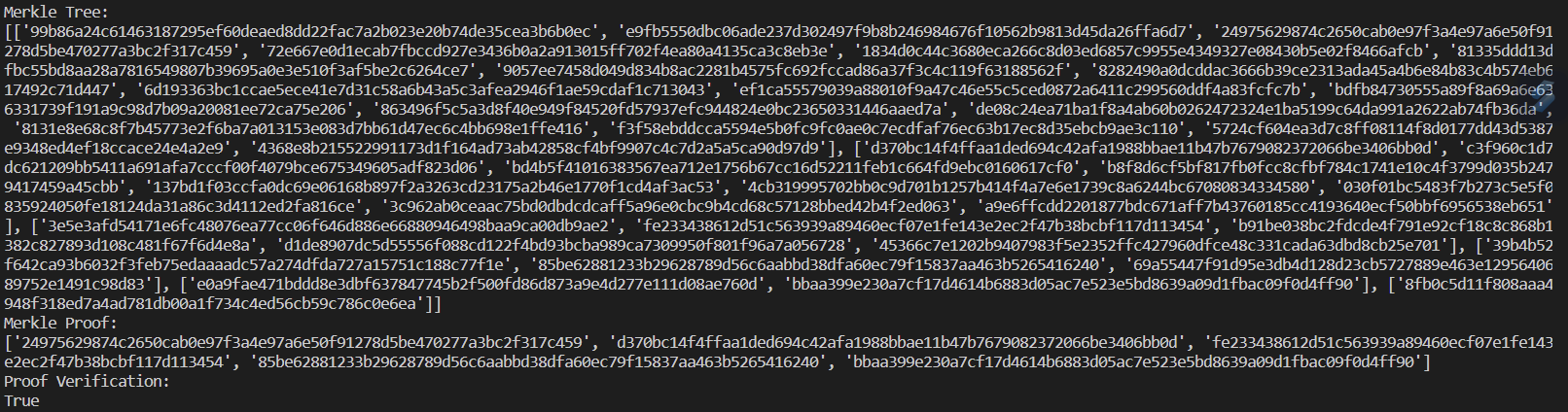
Shows the transaction object attributes describing the transaction taking place and also the digital signature which is a very long number due to the nature of encrypting with the private key through RSA encryption (raising the encoded transaction ID to the power of the private key which in itself is already a very long number for more security)



Digital signature is verified proving the user saying they are the owner of the public key they are making the transaction with really is the owner as decrypting the digital signature with the public key gave back the transaction ID meaning the keys are linked (relies on private key being kept secret)

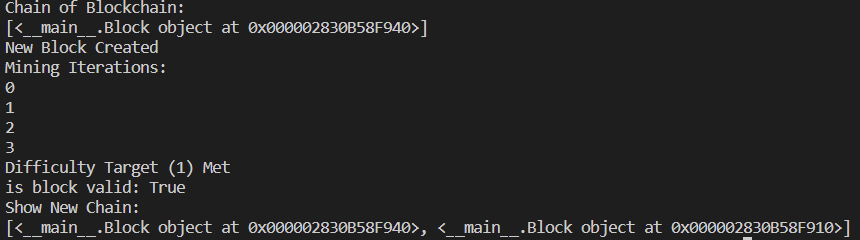
Transaction was then validated further by checking if the user had the sufficient funds, which they didn’t as the wallet was generated and had no currency allocated to it’s balance beforehand

Testing – Merkle Tree Creation, Merkle Proof, Proof Verifying (Line 333)



A Merkle tree is generated from an example dataset of unique strings in place of transactions. The Merkle tree is stored as a list of lists that represent the levels, with the last list in the list being the root node and the first list in the list being the level of leaf nodes. A Merkle proof is generated taking in an element of the dataset for inclusion to be proved. The Merkle proof is a list of nodes the element in question hashes directly with in its path to the root. The proof is verified by taking the element in question and hashing it with elements of the proof in order, which results in the root node if the data in question is really in the Merkle proof. It then compares the generated root to the actual root and they are equal in this case so the verification returns True

Testing – Block & Blockchain Creation, Mining, Block Validation (Line 466)



A blockchain is generated with the genesis block hardcoded in. A new block is created with the transactions being an example dataset like used above. The nonce value is incremented until the difficulty target, in this case 1, is reached meaning the hash of the block is one such that it starts with 1 zero. The difficulty target has been set to a low value for simplicity’s sake, but it will be set to a value such that it takes, on average, 10 minutes of incrementing this nonce value until the hash is reached (taking into account how many other miners would be on the network and such – explained far above in the section about the algorithm that determines the difficulty target). The block is validated by checking the hash value of the block meets the difficulty target, which it will because we just mined it but this is for blocks that are being broadcasted in from other nodes across the network because of the trust-less decentralised system of a blockchain, any incoming block needs to be validated. The chain is a list of block objects.

**Identification of End Users**

My brother, Sam Mirnejhad, is looking for a program he can use to launch his cryptocurrency idea with. He is someone that wants to have a secure way of making transactions through the immutability of the blockchain that stores the history of transactions, and also a way for users to make transactions and store their currency on the program without having to worry about fraud or having their money stolen.

Interview

Q1: what are you looking to get out of a blockchain program

A1: Transactions are government controlled, centralised. I want a currency system that is decentralised, not controlled by a central authority, but still have a set of rules that are adhered to

Q2: what security measures are you looking for

A2: I want transactions to be authenticatable and verifiable so that no one can fraudulently make transactions in another users name, and I want it so that once a transaction is made, you cannot change it, as to prevent double spending of currency

Q3: how else is this going to be any different from usual currencies that are centralised

A3: centralised currencies such as US Dollars and Great British Pounds are heavily subject to inflation, which erodes the value over time. I want my cryptocurrency to not be hit as hard by inflation by controlling the amount of currency in system, as part of inflation is due to the central authorities just printing out money when they feel like it

Q4: why would people want to choose cryptocurrency over just usual currency

A4: I want this program to be financially inclusive, so that anyone with an internet connection has access to financial services, because centralised currencies require access to a bank, and I want this program to give more privacy as well

Analysis of Interview

The end user wants a currency that is decentralised. We can make this by allowing anyone to add blocks to the blockchain and allow anyone to run a copy of the blockchain, users with this power are called nodes. While adding blocks to the blockchain can be costly, anyone can run a copy of the blockchain, being apart of the network, what nodes do is listen for transactions from users and broadcast them to other nodes once they’ve validated it (for the other nodes to also validate it too for themselves, as it is decentralised and trust-less) and also they validate and add broadcasted blocks (from other nodes) onto their chain (a miner node can make blocks). Transactions can be authenticated through RSA encryption’s public and private key pair generation, a user must sign a transaction when they make the transaction. Which proves their identity and that they are not making a fraudulent transaction, once the transaction is validated. Transactions are easily validated due to the nature of RSA encryption and the method of asymmetric key encryption. Transactions, once added into the blockchain through block creation, cannot be changed because of how mining blocks works. On average in my program it will take 10 minutes to mine a block (due to the algorithm for mining difficulty target) and due to the ‘linked list of hash pointers’ nature of the blockchain, every block after the block you alter must be remined. Even if this is successfully done, this will mean that one copy of the blockchain on one node will have a completely different copy of the blockchain to all other nodes. The other nodes will only accept this copy and make their copies like the new one if it is a longer chain, due to the decentralised nature of blockchain. This will only happen if the node is mining blocks faster than all other miners combined, otherwise they will be working towards a longer chain (the real chain). This is called a 51% attack because the malicious node needs the majority of the computational power of all the miners on the network combined for this to work, this is the defence against changing transactions in the blockchain.

Inflation will be tackled by having a cap on the total possible amount of cryptocurrency on the network. Bitcoin is capped to 21 million, and they do this by having a system where new currency is only generated through block mining reward. This is the reward that miners are awarded for creating a new block and adding it to the blockchain, and it is halved approximately every 4 years depending on rates of block creation. My program will have the same system, but the cap will be a lot lower.

Privacy will be far stronger in my program through the fact that almost everything is hashed first before being stored on the public chain and is publicly represented by it’s hash.

# Design

**Mission Statement**

My project needs to generate a public and private key for users that are linked in a way such that one encrypts a message and the other decrypts a message but you cannot derive one from the other (RSA encryption), allow users to create transactions, authenticate their identity through signing the transaction with the private key (encrypting the transaction ID with the private key), broadcast the transaction across the network to a node for the node to verify the transaction (checking that the user has the sufficient funds to make this transaction and checking the digital signature on the transaction by decrypting it with the sender’s public key to see if it returns the transaction ID meaning they are who they say they are) and hold it in the transaction pool while broadcasting the transaction to all the other nodes on the network for them to verify and add it to their copies of the transaction pool (list of unconfirmed transactions). Then my program needs to gather the transactions into a set when the length of the transaction pool has reached a certain value, this set being ready for miner nodes on the network to pick up and create a block with this set of transactions. Once they fill out the metadata for the block (automatically done upon block creation) they can mine the block where they are trying to find a value (called the nonce) such that the overall hash of the block’s content including this value starts with a certain amount of 0s which is set by the network, called the difficulty target which is calculated algorithmically such that on average it takes 10 minutes to mine the block given the time it takes to mine a block, and the amount of active miners on the network. Once the block is mined, the miner node will add it to their copy of the blockchain and broadcast it to all other nodes on the network for them to validate the block by validating each of the transactions in the block, and verifying that the hash of the block meets the difficulty target. If the block is validated successfully, it is added to their copy of the blockchain too, if the block is rejected then they will not add it and it means there is a malicious node either broadcasting a malicious block or rejecting a valid block. Either way, this means one node will have a different copy to the rest of the nodes and will have their chain rejected by the other nodes unless they successfully perform a 51% attack as outlined at the end of the analysis, which would require them to hold the majority of the computational power out of all the miner nodes on the network. Blocks also contain information like the Merkle root which helps strengthen the integrity of a block as the Merkle root will change if any of the transactions of a block are tampered with since the Merkle root represents the hash of all the transactions in the block.

**Design Introduction**

* Design (each component in my program explained in the analysis)
* TO-DO: Hierarchy Chart for Functions

**Data Structures**

Linked List of Hash Pointers

The blockchain’s chain is a linked list of hash pointers. A linked list consists of nodes which are made up of the data element and a reference (called a link or pointer). The reference is what links the nodes, with one node having a link, linking it to the next node. In blockchain, the blocks have a hash pointer to the previous block. Every block will have a hash pointer pointing to the block before it in the chain, which is what links the blocks. This is fundamental to blockchain and has many purposes. It creates a chronological and cryptographic link between the blocks, forming a chain of blocks. Including the hash of the previous block in a block contributes heavily to the immutability of blockchain, as changing the contents of one block will change it’s hash, meaning the next block in the chain now has a new reference since the hash changed, which means the contents of that block now changes, this is an ongoing effect all the way to the last block in the chain, meaning changing the contents of one block will change the contents of every block after it, and due to the nature of mining, this is an intractable problem, therefore transactions cannot be altered after they are confirmed in the blockchain, providing security against tampering. This data structure for the chain also makes it a lot faster to resolve forks in the chain, when different nodes may have different copies of the blockchain from a certain block (due to a malicious node remining blocks from changing the content of a block). Every time a block is remined it is broadcasted to the network for the other nodes to pick up on it and validate it, while these remined blocks are valid they are obviously still products of malicious activity, so the network accepts the longer fork, the one with more computational work put in it, as the true chain. This data structure allows for quick fork resolution as it is fast to check which prong of the fork in the chain is the longest.

Merkle Tree (Form of Binary Tree)

The Merkle tree is a binary tree where the leaf nodes consist of the hashes of the transactions, and the next level of nodes, the parent nodes of the leaf nodes, are made by pairing the leaf nodes and hashing their concatenation to get the parent node. This process is repeated for the parent nodes to get the next level, and is recursively continued until the Merkle root is reached, which represents the hash of all the original transactions in some way. With Merkle trees comes the concept of Merkle proofs, which, given a specific transaction, can check if that transaction was included in the Merkle tree without searching the entire Merkle tree, only using the nodes along the path from the specific transaction to the Merkle root node. The list of nodes along this path is called the proof, and is used to reconstruct the Merkle root given the specified node. If the specified node is really in the Merkle tree, then the reconstructed Merkle root will be the same as the actual Merkle root, this also guarantees the integrity of the entire Merkle tree, proving that no transactions have been tampered with, because a change in the transactions would change the Merkle root and a change in the transactions in the proof would also change the reconstructed Merkle root. This is used to validate the integrity of the transactions in a block when a block is mined and broadcasted to all other nodes on the network for them to validate. Any altercation to a transaction would require the whole Merkle tree to be recalculated for the new Merkle root, which is intractable to do for every block past a block deep enough into the chain.

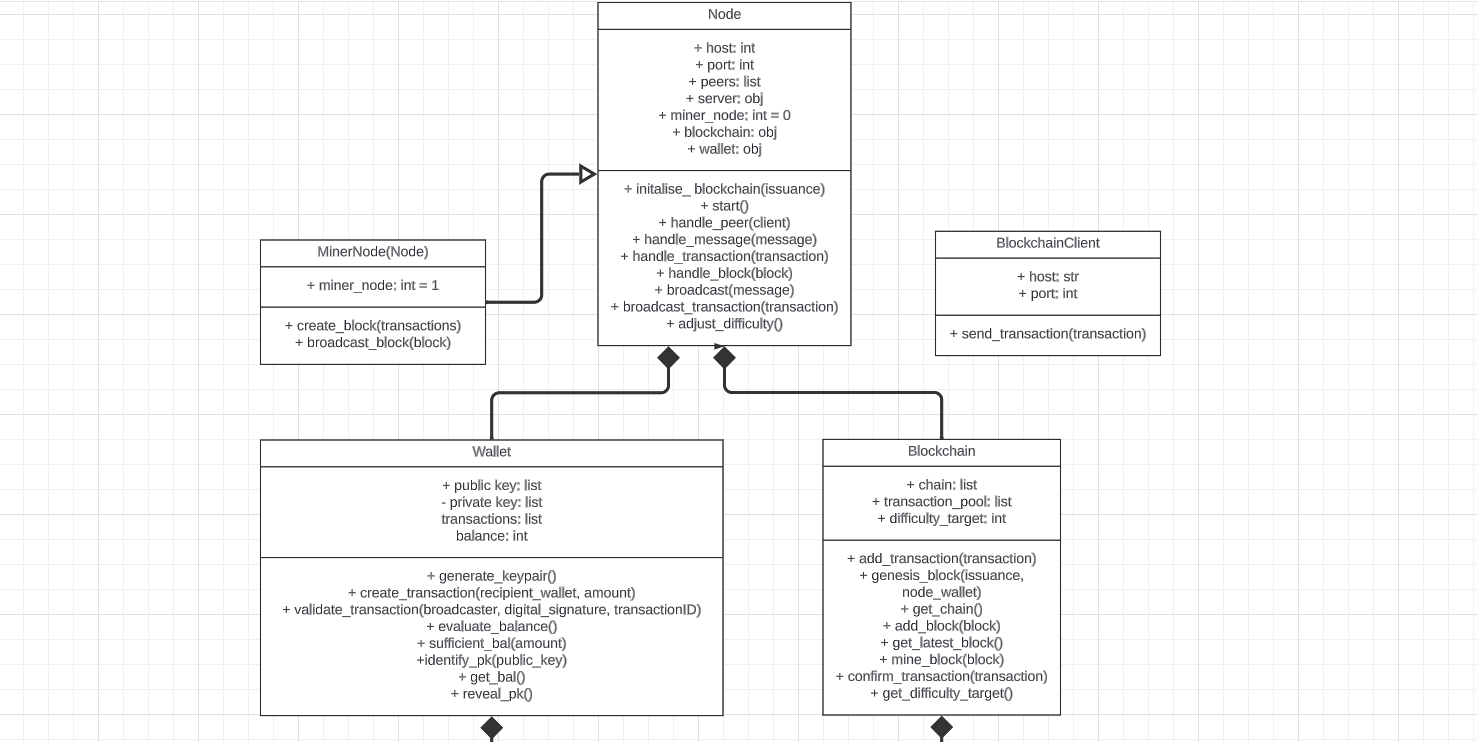
**Database / Data Storage Design**

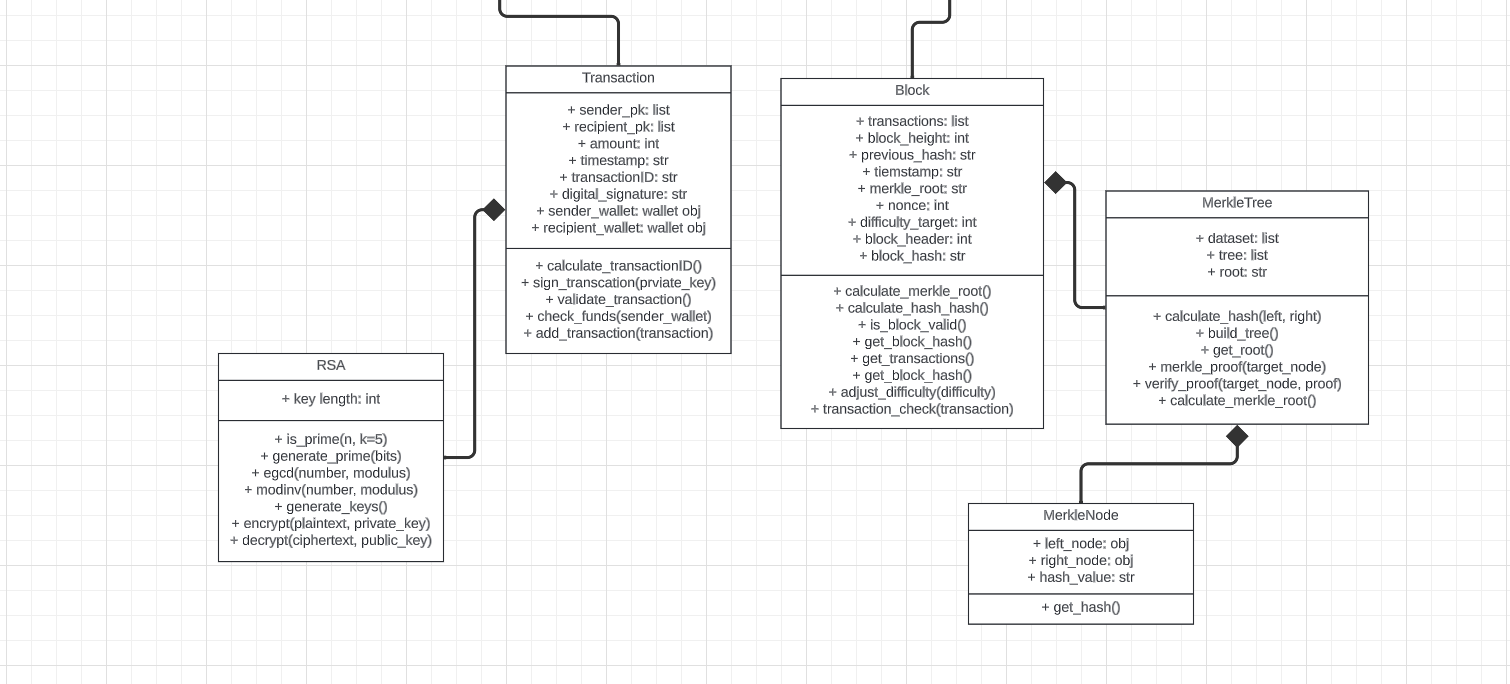
* TO-DO: ERD for database (implement blockchain load save file first)
* TO-DO: Data Dictionary (implement blockchain load save file first)
* TO-DO: Data File Structure (implement blockchain load save file first)

**Key Algorithms**

* Identify Algorithms
* Explain Role of Algorithms
* Psuedo-code or Flowcharts of Algorithms
* Data Packets Structure (Networking)
* RSA Encryption Algorithm, Digital Signature & Signature Verification Algorithm, SHA-256 cryptographic hash function, Extended Euclidean Algorithm, Miller-Rabin Primality Test, Merkle Tree Generation Algorithm, Merkle Proof Algorithm, Difficulty Target Adjusting Algorithm, Block Mining Algorithm

**Class Diagram**

****

****

**Function Listing**

* List of all functions in a table
* Headings: Func Name, Parameters, Returns, Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Parameters** | **Returns** | **Description** |
| get\_dataset | None | example dataset | returns example dataset |
| data\_gen | None | None | generates example dataset for testing different parts of the blockchain without having to make loads of transactions |
| is\_prime | number, iterations | True  or  False | uses the Miller-Rabin Primality test algorithm to test if a number is prime, iterations defines how many times the test is repeated as the test is a probable primality test not perfect |
| generate\_prime | bits | a prime number | a while loop that generates a random number with a certain amount of bits, test if the number is prime using the is\_prime() method, if it isn’t then continue loop, if it is then return it |
| egcd | number, modulus | extended greatest common divisor | using the Extended Euclidean algorithm, the method takes in a number and a modulus, and returns the extended greatest common divisor |
| modinv | number, modulus | modular inverse  or  ‘it doesn’t exist’ | calculates the modular inverse of a number modulo the modulus by calling the egcd() method to get the extended greatest common divisor of the number modulo the modulus, and returns the modular inverse which is part of the egcd() return after checking that the first part of the egcd() return is equal to 1 which means a modular inverse exists, otherwise it returns saying it not exist |
| generate\_keys | None | public\_key  and  private\_key | this method uses RSA encryption to generate a public private key pair which involves generating primes using the generate\_prime() method and using the modinv() method to find part of the private key |
| encrypt | plaintext, private key | ciphertext | plaintext is encrypted with RSA encryption algorithm using the private key |
| decrypt | ciphertext, public key | plaintext | ciphertext that has been encrypted with the related private key will be decrypted using RSA trapdoor permutation with the public key, but will not be decrypted correctly if the keys are not linked |
| generate\_keypair\* | None | None | from the wallet class calls the generate\_keys() method in the RSA class, assigns keys to initialised attributes |
| create\_transaction | recipient\_wallet,  amount | transaction object | takes the wallet of the user who is receiving the transaction, and the amount being sent to them, creates a transaction object |
| validate\_transaction | broadcaster, digital signature, transactionID | True  or  False | takes the wallet of the user making the transaction, the digital signature made by the user, decrypts the digital signature with the user’s public key using RSA encryption and checks if the resulting plaintext is the transactionID because a digital signature is the transactionID encrypted with the private key |
| evaluate\_balance | None | balance | the wallet has an attribute called transactions which keeps track of every transaction referencing the user as a sender or recipient, this method checks each transaction in the list and calculates a final balance at the end |
| sufficient\_bal | amount | True or False | uses the evaluate\_balance method to get an up to date balance to compare amount to, checking if the user has the sufficient funds to spend the amount |
| identify\_pk | public key | self (wallet) or nothing | given a public key, if the public key given is the same as the wallet’s public key, return the wallet |
| get\_bal | None | balance | returns balance of wallet |
| reveal\_pk | None | public key | returns public key of wallet |
| calculate\_transactionID | None | transactionID | hashes the data of the transaction |
| sign\_transaction | private\_key | digital signature | takes the transaction ID and encrypts it with the private key of the sender using RSA encryption, resulting in the digital signature |
| validate\_transaction | None | True or False | takes the digital signature of the transaction and decrypts it with the sender’s public key using RSA, if it returns the transaction ID then the keys are linked, proving the sender is who they say they are assuming they keep the private key safe |
| check\_funds | sender | True or False | calls the sufficient\_bal method on the sender’s wallet and returns |
| add\_transaction | transaction | None | adds a transaction to the history of transactions a wallet has |
| update\_records | None | None | uses the identify\_pk() method on the wallets of both ends of the transaction to get the wallet objects, then uses the add\_transaction() method on the objects to add the transactions (self) to both users’ histories |
| get\_hash | None | hash | returns a Merkle tree node’s value |
| calculate\_hash | left, right | hash | will take a left object, and a right object, convert them into strings, and will concatenate them in this order then input the concatenation into the SHA-256 hashing algorithm |
| build\_tree\* | None | Merkle tree | using a dataset, the leaf nodes of the tree are generated by taking each element in the dataset and hashes them. The next level of the tree is generated by looking at every other node in the leaf level, taking that node and the node to the right of it, and generating MerkleNode objects for each pair which in the process uses the calculate\_hash() method taking in the left and right node. This level of parent nodes is used in the same way to generate the next level, and so on through a while loop until the Merkle root is reached, when the length of the level is equal to 1 (there is only 1 node in the level) |
| get\_root | None | Merkle root | returns the first index of the last index of the tree, which is the root |
| merkle\_proof | target node | proof path | given a target node amongst the leaf nodes, the method will start from the target node and traverse the tree until the root is reached, appending all nodes directly concatenating and hashing with a node along this path to the root, to the proof path through a while loop that moves through each level, that reassigns the target node as it’s parent node so that in the next iteration of the while loop, the method looks for the parent node to find the sibling node that concatenates and hashes with it to get that parent node’s parent node, until the root is reached and the proof path is a list of every node (one node per level) that concatenates and hashes with the nodes in the path from the original target node to the root node |
| verify\_proof\* | target node, proof path | True or False | this method takes a target node in the leaf level, and calculates the hash of it’s parent using the first element in the proof path which represents the sibling node used to generate the parent node with said target node. The target node is then reassigned to the parent node and it is checked against the next element in the proof path which should represent the sibling node that combines with the current target node to get the parent node. This process continues through a for loop looping through the proof path until the last element of the proof path and the latest iteration of the target node are put together to calculate the parent node. IF the original target node was ever actually in the tree then this current parent node will be the Merkle root, so the method returns True, if not then it returns false as the calculated Merkle root will be checked against the actual Merkle root and if they are not the same then the target node is not in the tree. This proves if a target node is or isn’t in a tree. |
| calculate\_merkle\_root | None | Merkle root | given the block’s set of transactions this method calculates the Merkle tree of the set of transactions by creating a MerkleTree object taking in the set of transactions which automatically generates the tree using calculate\_tree(). Then the get\_root() method is called on the object to return the root of the tree |
| calculate\_block\_hash\* | None | None | this method takes the block object’s block header and nonce value attributes and converts them into strings then concatenates them for hashing using the SHA-256 hashing algorithm. If the resulting hash meets the block’s difficulty target attribute, meaning if the difficulty target is the value n, the hash’s first n digits must be equal to 0, then the hash of the block is set to this hash, if not then a while loop repeats this process after incrementing the nonce value by 1. The while loop will break when a nonce value is found such that it’s pairing and hashing with the block header (which contains all the information of the block) returns a hash that meets the difficulty target |
| is\_block\_valid\* | None | True or False | this takes a block and checks that it’s hash meets the difficulty target by taking the nonce and block header and recalculating the hash of the block using the calculate\_block\_hash() method. The result is added to a list called check, which will be checked at the end to see if all the elements of the list are True. This method then takes the block’s set of transactions and validates each one individually using the validate\_transaction() method which will return either True or False for each transaction, each result is added to the list named check. After validating each transaction using a for loop, the all() function is used on the check list, which returns True if all it’s elements are True, otherwise it returns False. |
| get\_block\_hash | None | block hash | returns block hash of the block |
| get\_transactions | None | transactions | returns block’s set of transactions |
| get\_block\_header | None | block header | returns the block header of a block |
| adjust\_difficulty | difficulty | None | reassign’s block’s difficulty target |
| transaction\_check | transaction | True or False | checks if a transaction is in the block’s set of transactions by generating the Merkle tree of the set of transactions, using the merkle\_proof(transaction) method to find the proof path given the target transaction, then the verify\_proof(transaction, proof\_path) method is called given the just calculated proof path and the transaction, returning True or False depending on if the transaction is actually in the tree |
| add\_transaction | transaction | None | adds a transaction to the transaction pool of a blockchain data structure |
| genesis\_block\* | issuance,  node wallet | genesis block | given the wallet of the node running this copy of the blockchain, and the issuance which is the starting currency being introduced into the circulation, this method will create the first block of the blockchain which is hardcoded into the blockchain using this method because it is different to usual blocks, it does not have a previous hash and no transactions are possible when no one has any currency in their wallets. The balance of the node’s wallet is set to the issuance, a genesis transaction is made to record this, which is a usual transaction object stating a transaction from the node wallet to the node wallet where the amount is the issuance, and the genesis block is created as a block object with the genesis transaction as its set of transactions (the block will recognise the blockchain is empty and will set the previous hash to 0) |
| get\_chain | None | chain | returns the linked list representing the chain of the blockchain structure |
| add\_block | block | None | appends a block to the chain |
| get\_latest\_block | None | latest block | returns last block in chain |
| mine\_block | block | mined block | given a block, this method empties the transaction pool of unconfirmed transactions and calls the calculate\_hash() method on the block, returning the mined block |
| confirm\_transaction | transaction | True or False | given a transaction, checks how deep the transaction is into the chain by checking each block’s set of transactions from the last block. A transaction is considered confirmed once it has a depth of 6 into the chain, as any malicious nodes attempting to remine the chain due to changing a transaction in a block will most likely have their fork in the blockchain rejected by the other network nodes after 6 blocks |
| get\_difficulty\_target | None | difficulty target | returns blockchain’s difficulty target |
| initialise\_blockchain | issuance | None | called on a network node to initialise their copy of the blockchain, creating the genesis block with the genesis\_block() method, adding it to the chain using the add\_block() method and generating the keypair of the node’s wallet using generate\_keypair() |
| start | None | None | create a socket, bind node to the socket and start listening for incoming messages, with threads for each peer node on the network for more efficient handling of multiple messages broadcasted in |
| handle\_peer | client | None | receives incoming data from peer nodes (client in this case), deserialises the message back into object form, and calls the handle\_message(message) function |
| handle\_message | message | None | if the message’s object type is a transaction then it calls the handle\_transaction() method, if the message’s object type is a block then it calls the handle\_block() method |
| handle\_transaction\* | transaction | None | the transaction is validated using the validate\_transaction() method, if the transaction is valid then then the transaction is added to this node’s copy of the blockchain’s transaction pool using the add\_transaction(transaction) method, then the transaction is added to both the transaction’s sender and receiver transaction histories. The transaction is then broadcasted to the other nodes on the network using the broadcast\_transaction(transaction) method. If the length of the transaction pool of this node’s copy of the blockchain has reached the transaction pool size limit (which determines how many transactions one block can hold as blocks can only be mined when the pool is full) and if this node is a miner node, then the miner node calls it’s own method create\_block(transactions) taking in the transaction pool as the set of transactions, and then the miner node empties it’s transaction pool. If the node is not a miner node then it doesn’t create a block with the set of transactions and only empties the transaction pool. |
| handle\_block | block | None | validates the block using the is\_block\_valid() method and if the block is valid then it adds the block to the node’s copy of the blockchain using the add\_block(block) method |
| broadcast | message | None | given a message it serialises a message using the pickle library for efficient transmission of objects over the network, then for each peer node in the list of peer nodes which each node carries as an attribute, use the send(message) method on the socket of the peer node to send the message to them , which exception handling in case of server errors |
| broadcast\_transaction | transaction | None | message is created from data, being the transaction, and the message type, being TRANSACTION, because the message type is important for how the serialisation of the message goes. The broadcast(message) is called with the new message formed |
| adjust\_difficulty | None | None | the amount of active miners on the network is calculated by checking how many of the nodes on the list of peer nodes are miner nodes, using this value and the time delay between iterations of the while loop in the calculate\_hash() method, the difficulty is algorithmically determined such that mining the block on average will take 600 seconds given these parameters of time delay and active miners, the difficulty target is evaluated at the logarithm (base 16) of the amount of active miners divided by the time delay, and the blockchain difficulty target attribute is reassigned to this |
| initialise\_miner | None | None | miner node attribute set to 1, where it is 0 for regular nodes |
| create\_block | transactions | None | the difficulty target of this miner node’s copy of the blockchain is recalculated first for a more up to date difficulty target, then a block object is created taking in the transactions and copy of the blockchain. The calculate\_block\_hash() method is called on the block to mine it, and the block reward, set to 5, is added to the miner node’s wallet’s balance directly. Then the block is broadcasted to the network using the broadcast\_block() method. |
| send\_transaction | transaction | None | through a blockchain client, a user sends a transaction to the network through this method, creating the socket, connecting to the server, defining the message made up of the datae, being the transaction, and message type, being TRANSACTION, the message is serialised and then sent to the network using the send() method |

**HCI / User Interface**

* Designs of User Interface (no UI yet)
* Explain Design Choices – link to end user interview (no UI yet)

# Technical Solution

# Testing

# Evaluation