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.

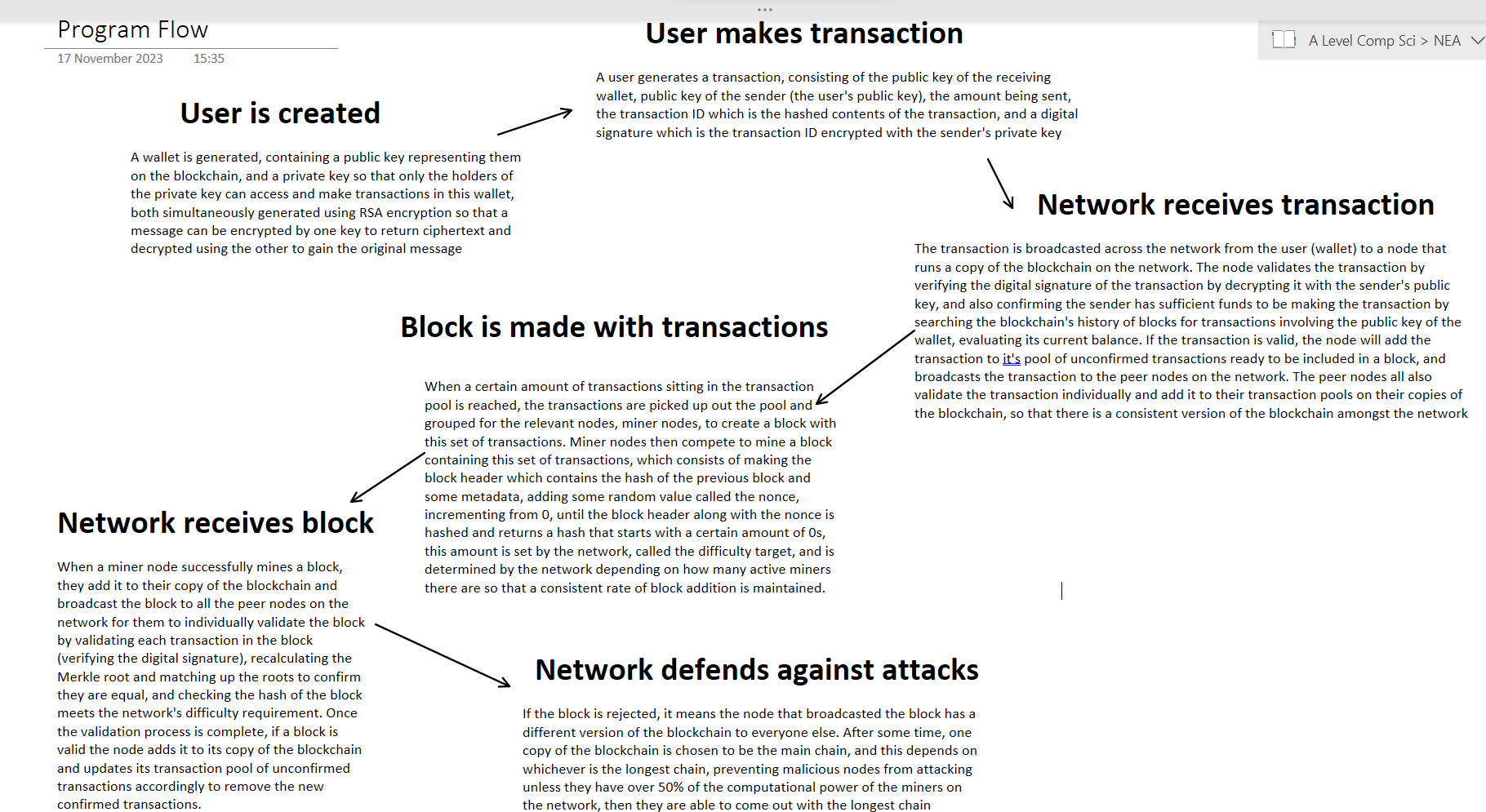
**Objectives**

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. Block Structure – Block Header
17. Block Structure – Block Header Hashing Function
18. Merkle Tree Data Structure, Tree Generation
19. Merkle Root Generation
20. Merkle Proof Generation
21. Block Structure – Merkle Root Calculation Function
22. Block Structure – Mining Function
23. Block Serialisation & Deserialization Functions
24. Child Class (Inherited from Node Class) Miner Node Class Functions
25. Network Communication – Block Template Generation
26. Network Communication – Difficulty Target Adjusting Algorithm
27. Node Class – Block Validation (Transaction Verifications & Valid Hash Check)
28. Node Class & Blockchain Class – Overspend Balance Deriving Function
29. Mining Block Reward + Fee Allocations in Transaction History
30. Blockchain Database (Saving Blockchain History)
31. Node Client Interface – Switching Between Users
32. Node Client Interface – Node Privileges (Viewing)
33. User Interface – All Functionality

**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.

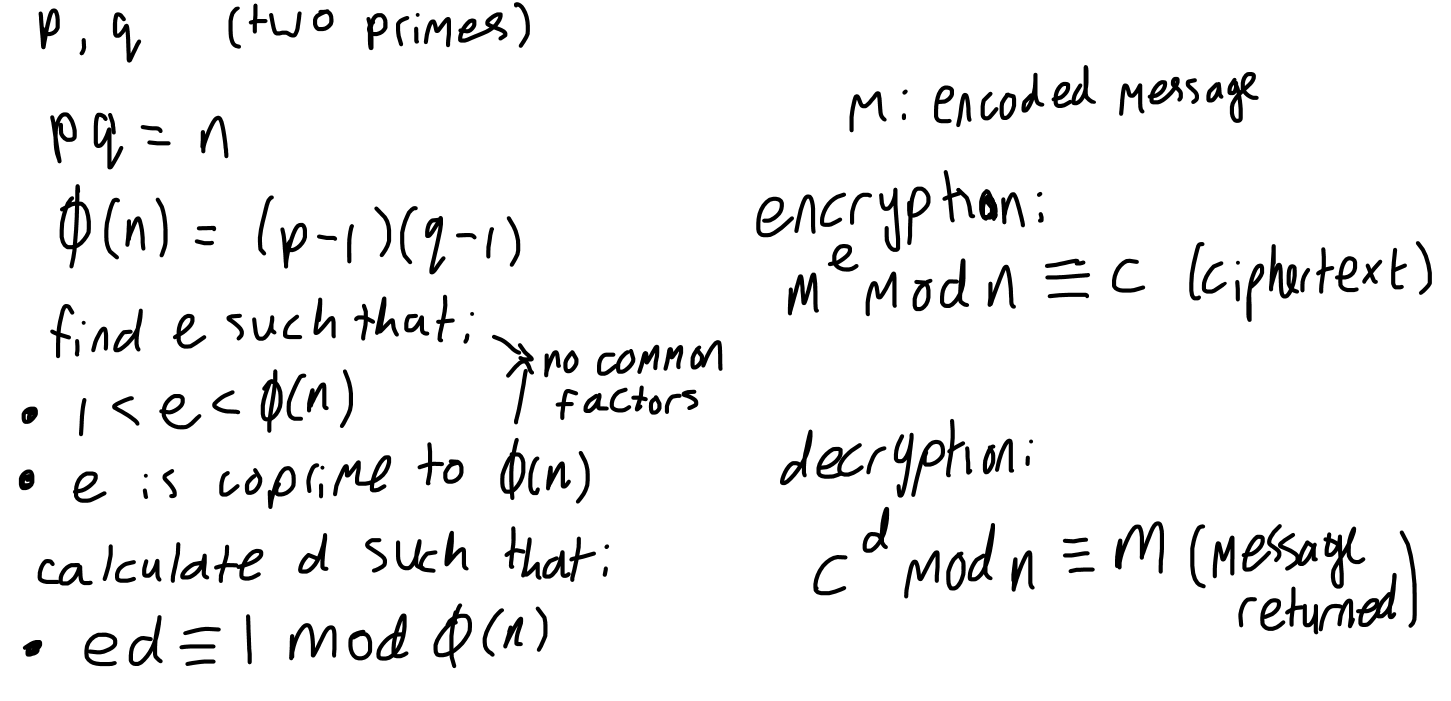


Back to the RSA encryption algorithm, it involves creating a public private key pair that are mathematically linked in such a way that if you encrypt a message with one key, you can decrypt it to gain back the original message from ciphertext with the other key. Typically in RSA encryption, the public 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.

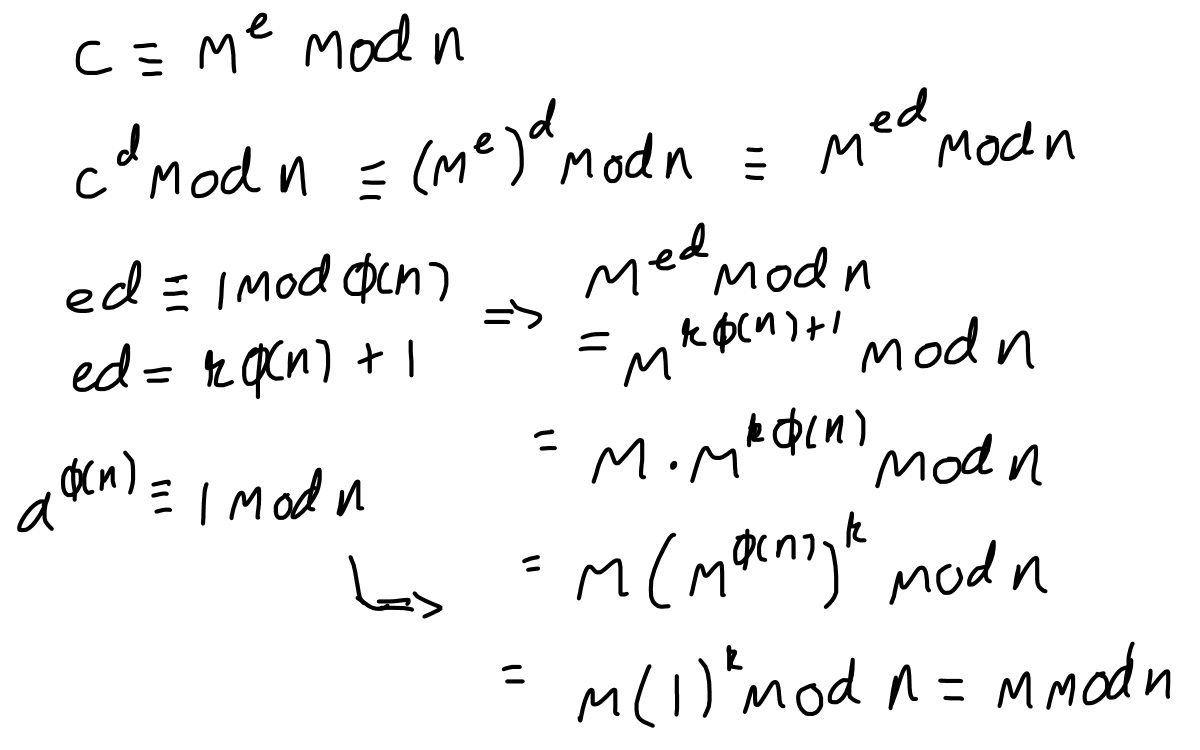
Now going into how the RSA encryption algorithm can generate these two mathematically linked keys, it uses a lot of modular arithmetic and prime numbers. The main underlying concept behind it is the fact that currently, we have a lot of methods to find prime numbers, but not efficient ways to break up 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 dependent 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 a 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.

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

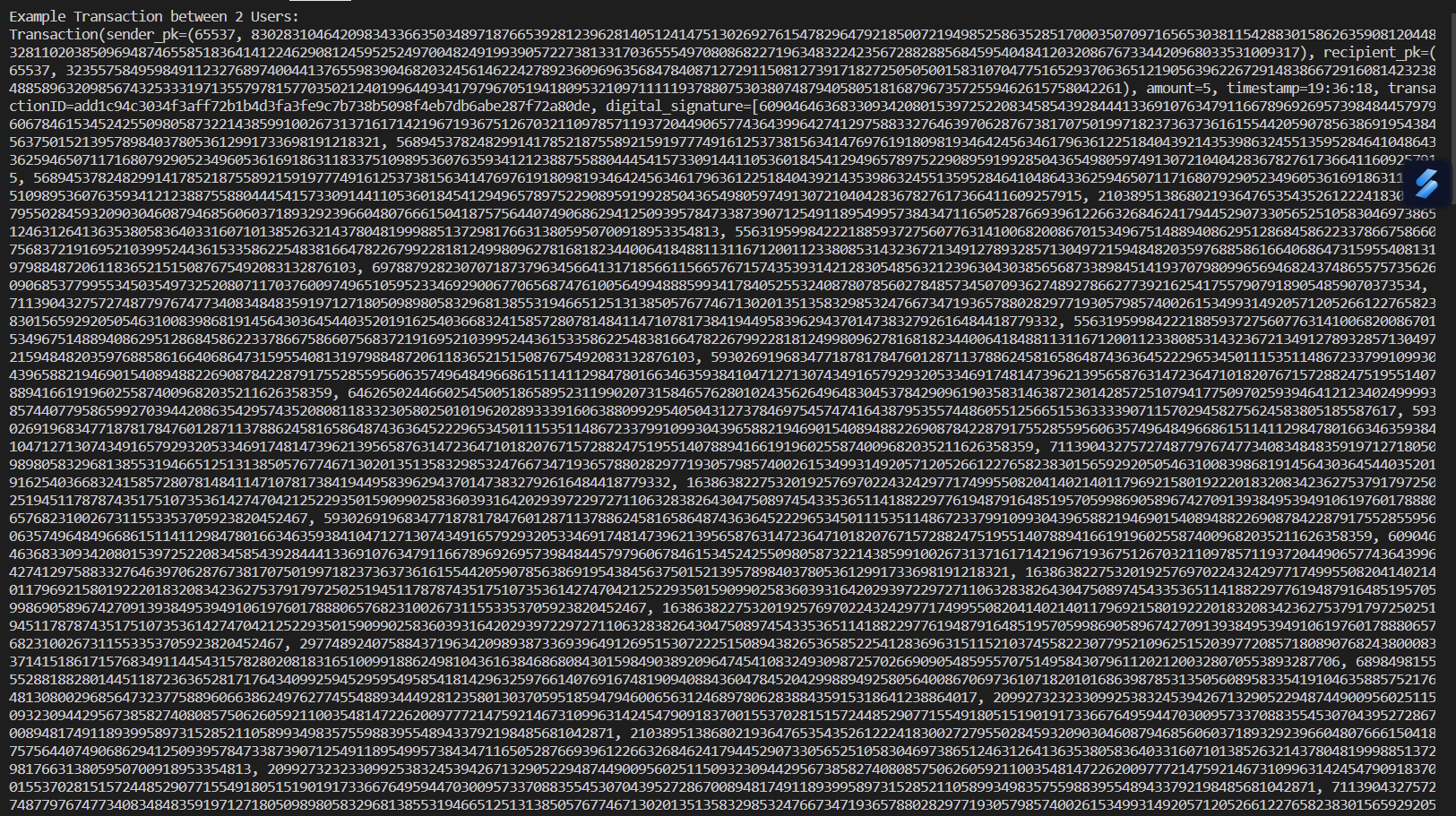
505  *# adding block to the blockchain*

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

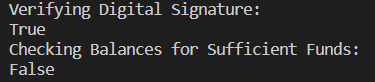
507  Blockchain1.add\_block(block01)  *# add block*

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

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



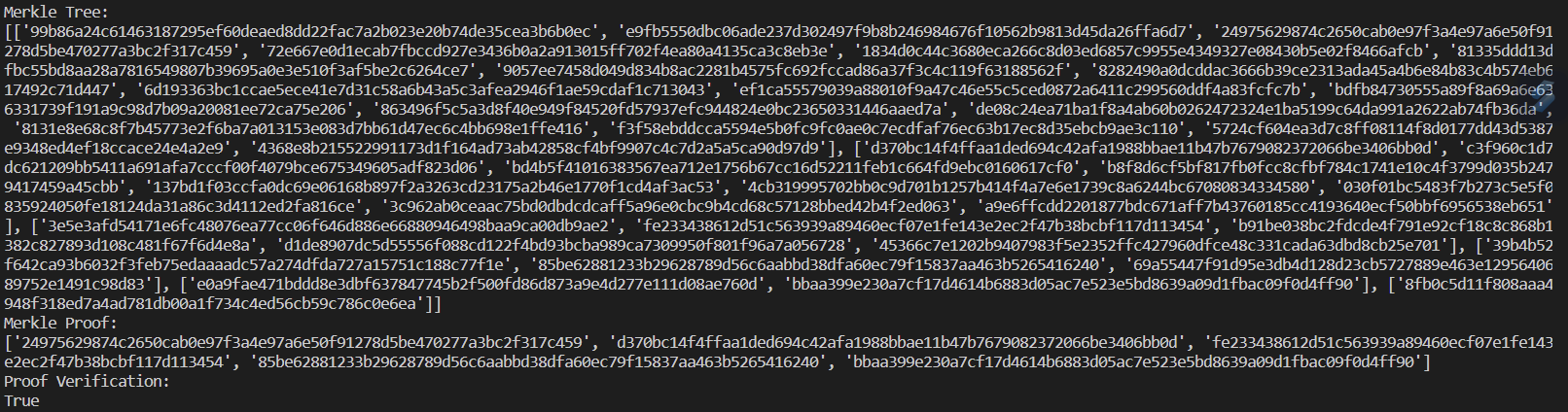
Shows the transaction object attributes describing the transaction taking place and also the digital signature which is a very long number due to the nature of encrypting with the private key through RSA encryption (raising the encoded transaction ID to the power of the private key which in itself is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

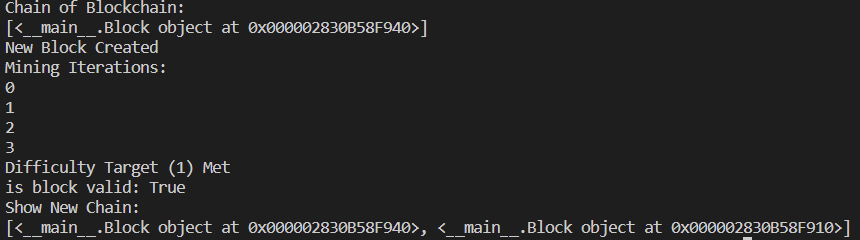
Transaction was then validated further by checking if the user had the sufficient funds, which they didn’t as the wallet was generated and had no currency allocated to it’s balance beforehand

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 & Introduction**

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.

**Data Structures**

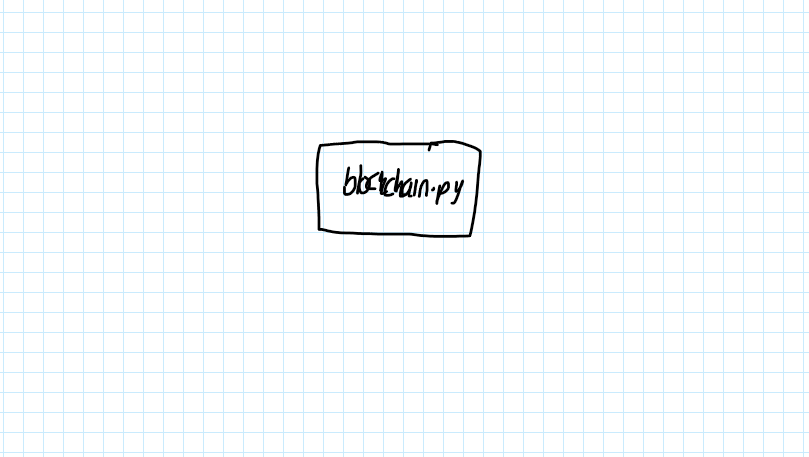
Linked List of Hash Pointers

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.

**File Structure**

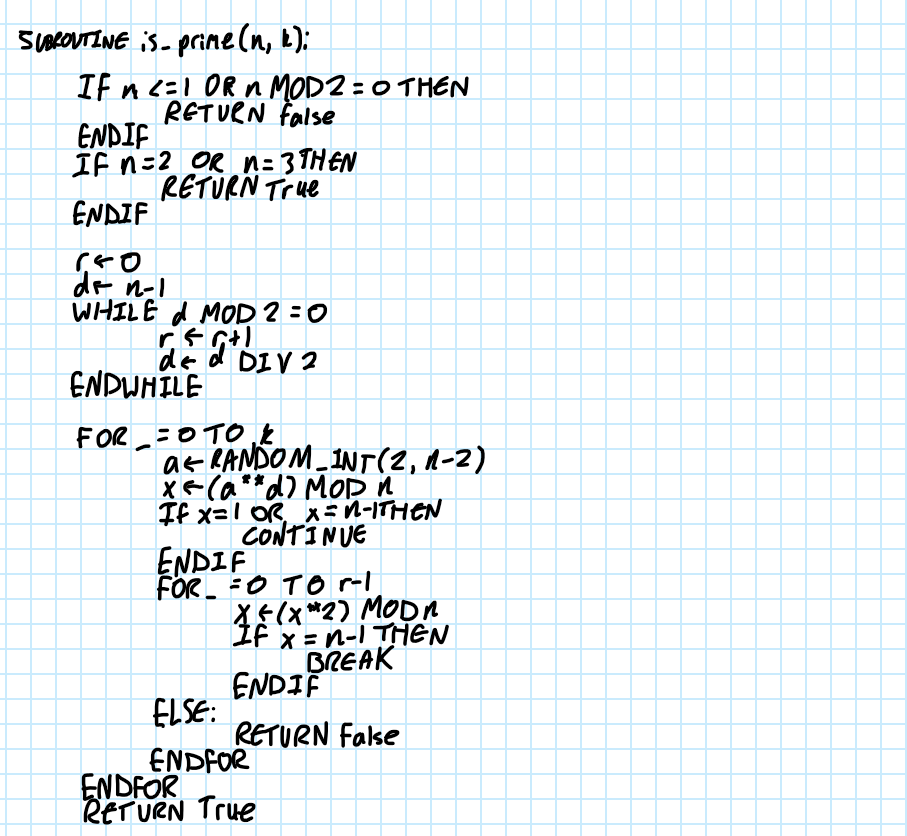


**Key Algorithms**

Miller-Rabin Primality Test (Line 39)

The purpose of the Miller-Rabin primality test is to check if a number is a prime number. This test is probabilistic, not deterministic, meaning it returns a value that is probably prime, however over multiple iterations, the likelihood that it is right is very high. Given some number n that we are testing, you express n as ((2 to the power of r) multiplied by d) + 1, these values are found by brute force. Then choose some random base ‘a’ which is between 2 and n – 2 inclusive. Using modular exponentiation, we then calculate (a to the power of d) modulo n. If the result of the computation is congruent to 1 modulo n or negative 1 modulo n then we consider n as probably prime, if neither of these conditions are met then the result is repeatedly squared, checking if these conditions are met on each iteration. If this also fails then it is unlikely to say that n is prime. This test is repeated for different values of ‘a’ acting as bases for confidence in the test.

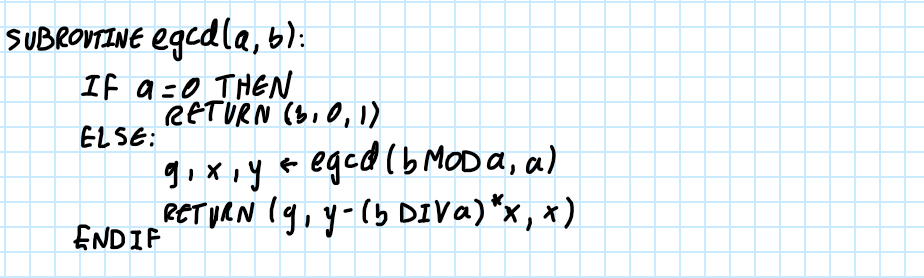
Pseudo-code:



Extended Euclidean Algorithm (Line 79)

The purpose of the extended Euclidean algorithm is to find the greatest common divisor between two integers ‘a’ and ‘b’ and represent the greatest common divisor as a linear combination of these two integers ( ax + by ) where x and y are the coefficients that satisfy the condition. This is implemented by the recursively calling itself where the parameter ‘a’ becomes the last recursive call’s ‘a’ taken modulo ‘b’, until the base case (a = 0) is reached and at this point the algorithm can now use values it has calculated by backtracking through the recursion, finally representing the greatest common divisor as a linear combination of the original ‘a’ and ‘b’.

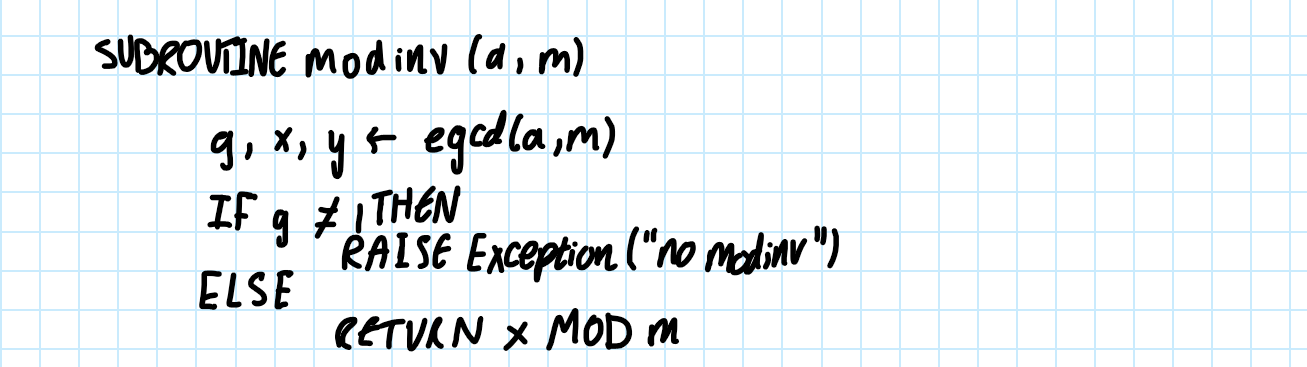
Pseudo-code: s



Multiplicative Modular Inverse (Line 92)

The multiplicative modular inverse of an integer ‘a’ modulo ‘m’ is some value ‘x’ modulo ‘m’ satisfying the condition that ‘ax’ modulo ‘m’ is congruent to 1. For the inverse to exist, ‘a’ and ‘m’ must be coprime, meaning the greatest common divisor between them is 1. The extended Euclidean algorithm is used to compute the greatest common divisor between them and if it is not 1, there is no inverse. If it is 1, the coefficient of ‘a’ in the linear combination of ‘a’ and ‘m’ from the extended Euclidean algorithm would be taken to modulo ‘m’ and this would be the multiplicative modular inverse.

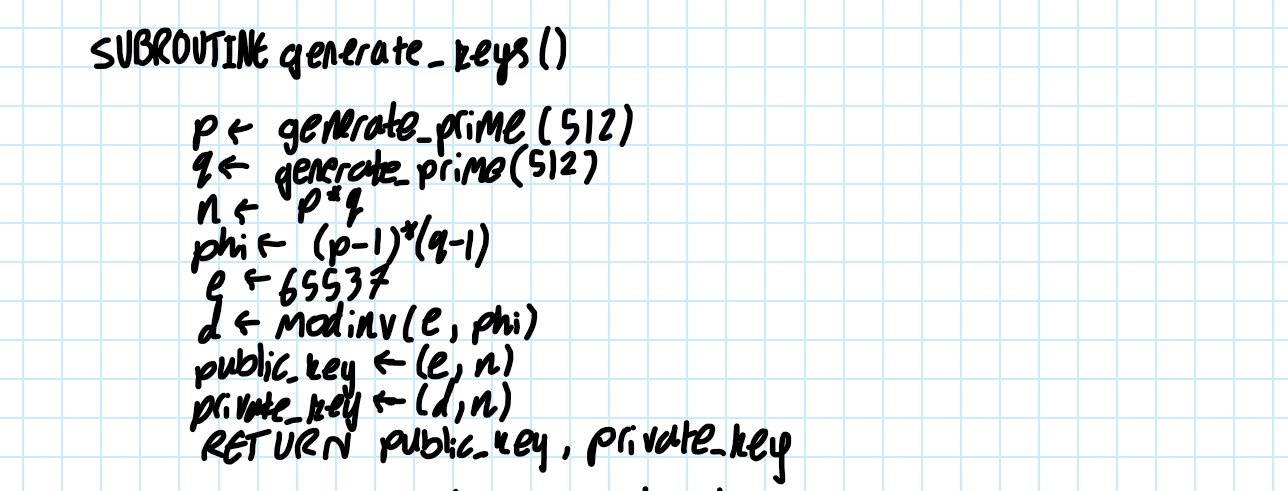
Pseudo-code:



RSA Key Generation (Line 104)

The public and private keys are mathematically linked in such a way that the effects of one can be undone by the effects of the other. In RSA, this is implemented through the properties of primes and modular exponentiation. A value ‘n’ is computed from two large primes which are found through the Miller-Rabin primality test, (they must be large to ensure that it is unlikely you can derive the two primes through brute force from having the product of them otherwise you would be able to derive the private key from the public key because the public key includes the product of the primes). Another value is calculated, the totient of ‘n’. If ‘n’ is the product of the two primes ‘p’ and ‘q’, then the totient would be the product of ‘p’ – 1 and ‘q’ – 1 which we will call Phi(n), this is coprime to the public exponent (greatest common divisor between them is 1). A public exponent is chosen, in this case it is the value 65537 as this is typically the public exponent chosen in RSA encryption. The public key is a tuple containing the public exponent and the product of primes. The private exponent of the private key - which also contains the product of primes - is derived through the fact that raising a value to the power of the one of the exponents, modulo Phi(n), then raising the resulting value to the power of the other exponent, modulo Phi(n), should act as doing nothing, as in one undoes the other, which is possible through multiplicative modular inverses, as raising a value to the power of another number twice is the same as raising the number to the power of the product of the two exponents which we want to have the overall effect of raising it to the power of 1. The algorithm just entails finding the multiplicative inverse of the public exponent with Phi(n) as the modulus, the resulting value is the private exponent, which is computationally infeasible to derive from the public exponent and ‘n’ without knowing ‘p’ and ‘q’ due to the use of Phi(n).

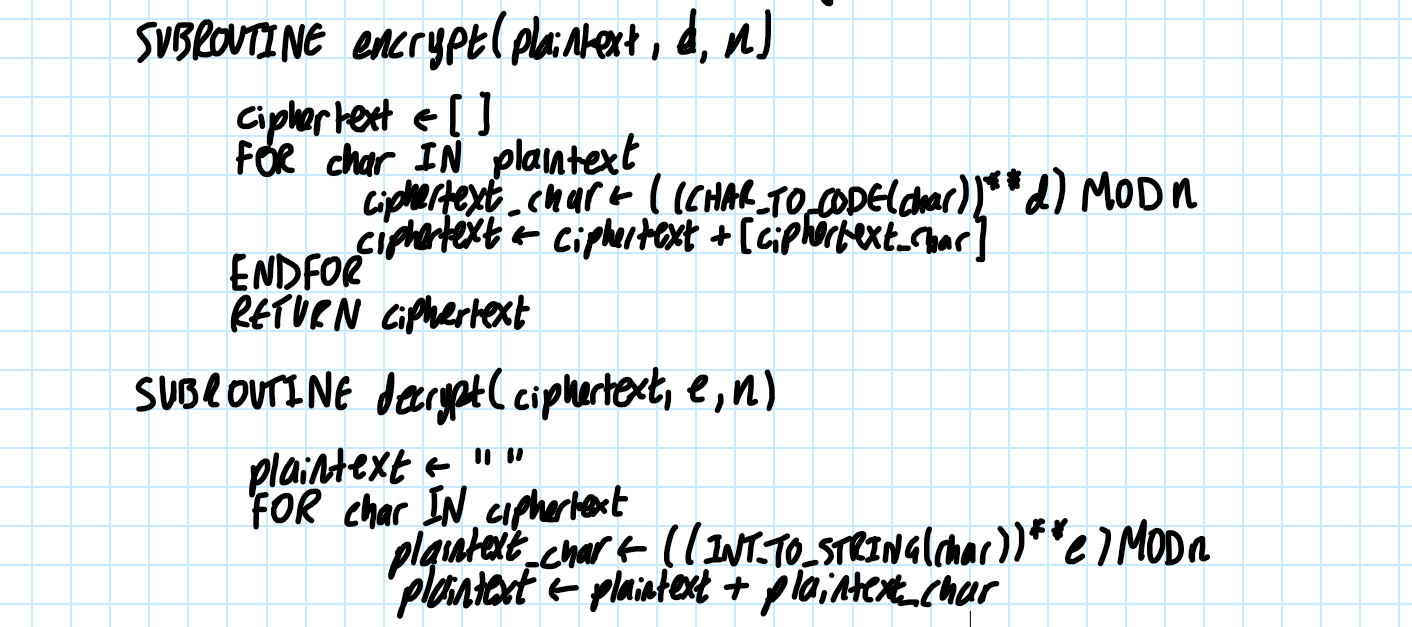
Pseudo-code:



RSA Encryption (Line 121) & Decryption (Line 130)

The purpose of the encryption and decryption algorithms is so that a transaction ID can be encrypted with the private key to act as a digital signature, this digital signature can then be verified by decrypting it with the sender’s public key which would confirm whether the user actually has the private key or not (meaning they are who they say they are and are not impersonating under another user’s public key) because the decryption wouldn’t work if the private key that encrypted the transaction to make the digital signature was not mathematically linked to the public key used to decrypt the digital signature, verifying the authenticity. The encryption and decryption algorithms go as followed: the plaintext or ciphertext is taken then each character would be taken and converted into Unicode so that it is an integer. This integer is then raised to the power of the relevant exponent, which is carried in the relevant key, modulo ‘n’ (which is also carried in both keys). In the case of encryption, the private exponent is used to exponentiate each Unicode value of the transaction ID, modulo ‘n’, which results in a ciphertext and in the case of decryption. Each of these encrypted numbers are raised to the power of the public exponent which undoes the encryption given the two keys were generated together and are mathematically linked.

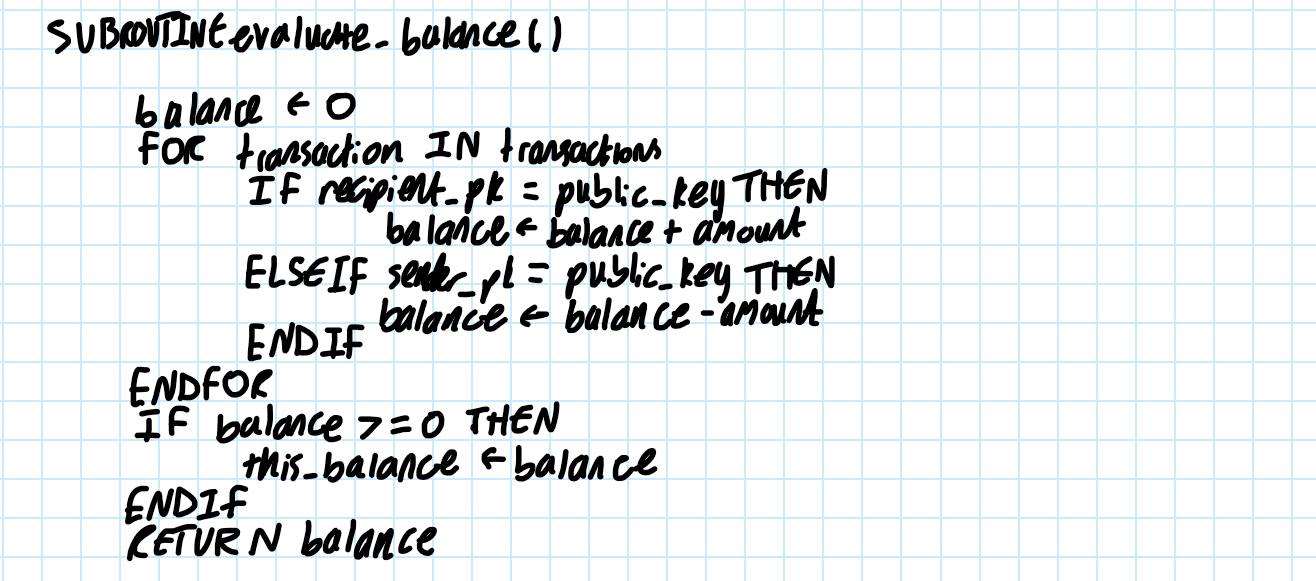
Pseudo-code:



Evaluate Balance (Line 182)

This algorithm takes the transaction history of a wallet, and calculates a current balance by looking at each transaction’s amount and incrementing a count, which starts at 0, by the amount if the transaction’s recipient wallet’s public key matches the public key of the wallet (meaning this transaction had this wallet on the receiving end) or decrementing the count by the amount if the transaction’s sender wallet’s public key matches the public key of the wallet (meaning this transaction had this wallet on the sending end)

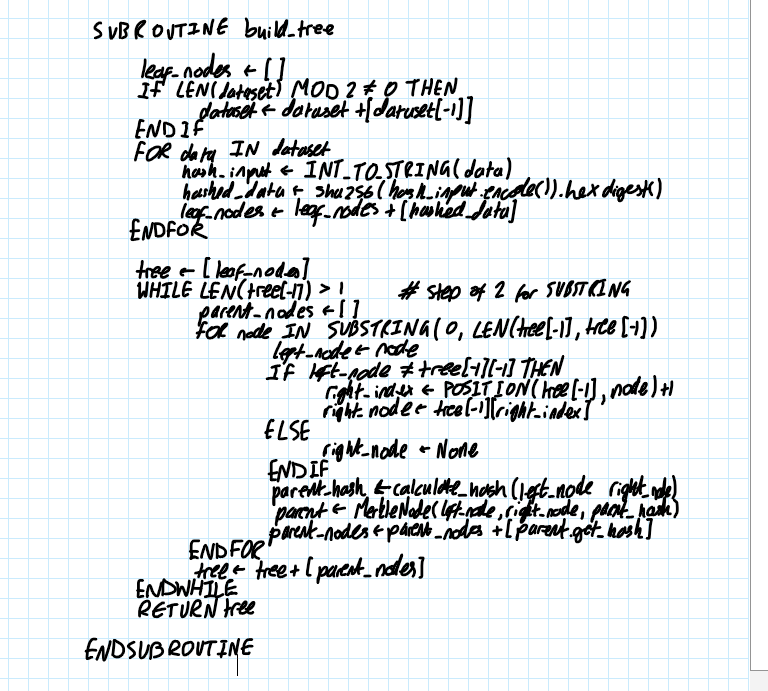
Pseudo-code:



Build Tree (Line 392)

The purpose of this algorithm is to generate the Merkle tree given a set of data, which is a binary tree where the data makes up the leaf nodes and each of the nodes above are made up of the hashing and concatenation of the child nodes beneath it. It works as follows: Given a dataset the first layer (leaf nodes) are generated by using each element in the dataset and hashing (to hexadecimal) with a hashing function (sha-256 from hashlib library). Iteratively it will then look at every other node in the leaf nodes, making up a pair of nodes, the left node and right node if the right node exists as if the left node is at the end of the leaf nodes (odd number of nodes), then there would be no right node (although this is just for safety purposes as when the leaf layer is formed, it is forced to become even length through duplication), the pair of nodes will then generate the parent node after concatenating the string representations of the node and hashing it. This is done until the parent nodes of the current level are formed, and a while loop will keep taking the current level of nodes (which will be the parent nodes formed in the last iteration) and it will generate the next level of nodes with the same process until the root node is reached (only one node in the level).

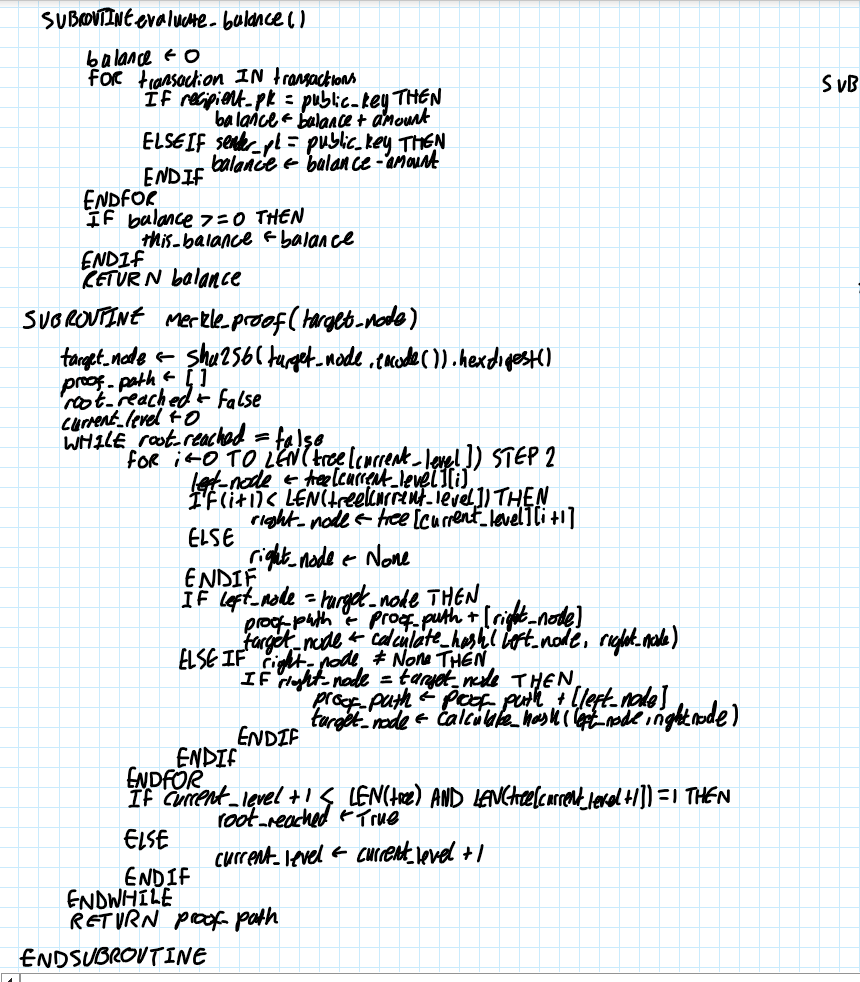
Pseudo-code:



Merkle Proof (Line 431)

This algorithm takes some data, generated a leaf node out of it as if it were in the Merkle tree and then iteratively loops through the elements list of nodes that make up the tree, picking up all the nodes that it would need to directly concatenate and hash with (the other node in a pair of nodes that make up the parent node directly above it) that would lead to eventually reaching the root node. Upon reaching the root node, a list of all the nodes in this path has now been formed which is then returned.

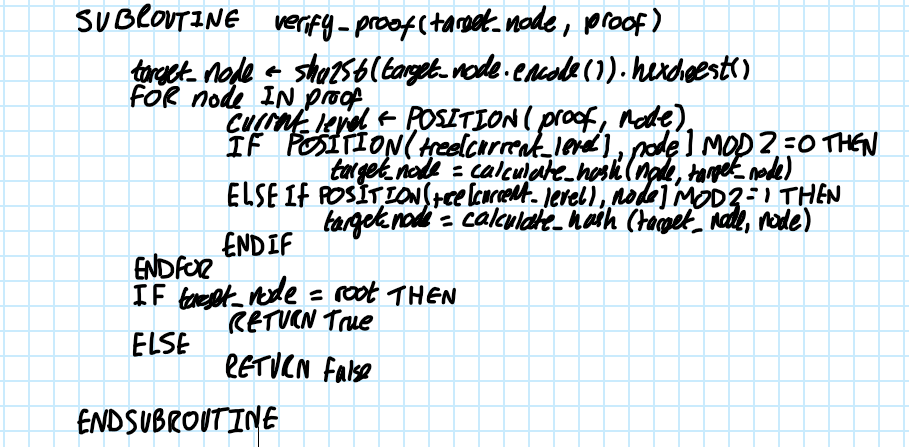
Pseudo-code:



Verify Proof (Line 459)

This algorithm takes some target node and its proof path and concatenates and hashes the target node with the first node in the path, taking the result, and concatenating and hashing this node with the next node in the path, and this is repeated until a final hash has been calculated. If the final hash is equal to the root node’s hash then it means the target node was in the tree to begin with, if the final hash is not equal to the root node’s hash then it means this target node is not in the tree, used for confirming whether some data was in the tree of transactions in a block.

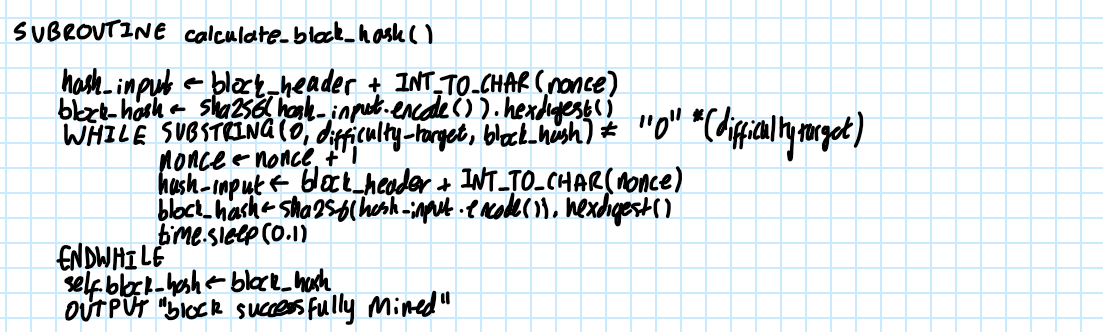
Pseudo-code:



Calculate Block Hash (Line 524)

This algorithm uses the SHA-256 hashing algorithm from the hashlib library to compute a hash representative of the block that includes information of the block including the hash of the previous block in the chain. The algorithm uses a while loop to keep reassigning the nonce value attribute of the block, providing a different hash value of the block each time, checking if the hash meets the difficulty target (the minimum amount of leading 0s the hash of the block must start with). Once a hash meeting this condition is found, it is used as the hash of the block given the corresponding nonce value that contributed to the result, and the block mining process is complete.

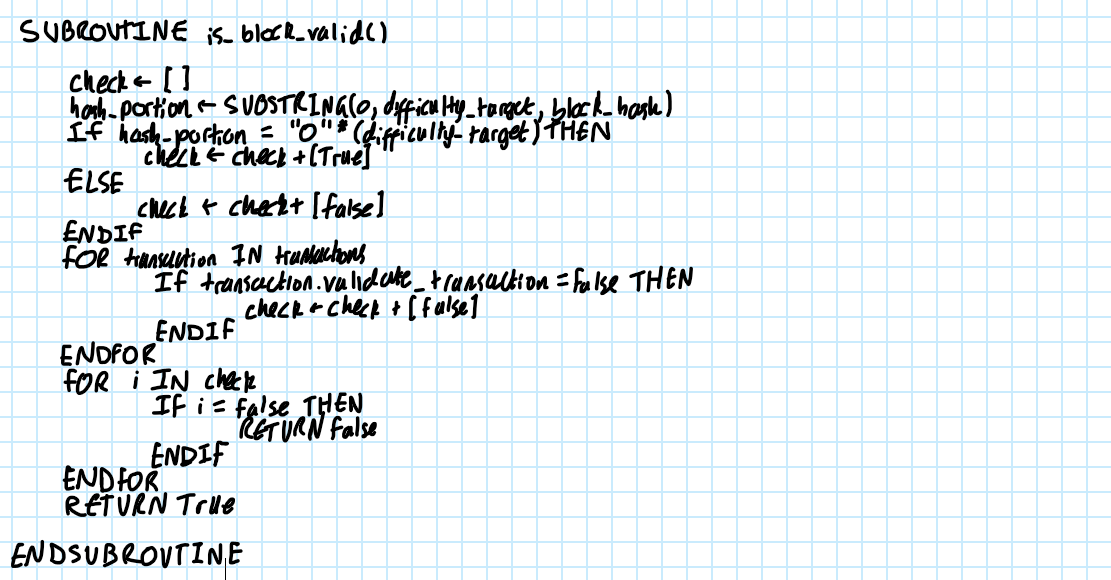
Pseudo-code:



Is Block Valid (Line 541)

This algorithm check that the hash of a given block meets the difficulty target (starts with the right amount of 0s set by the network to be allowed to be added to the chain as a block), and that each transaction in the block’s set of transactions is valid by checking the wallet making the transaction has the wallet’s private key too and isn’t just impersonating with another user’s public key (through RSA decryption methods previously built) and by checking that each transaction does not cause any users to have negative balance, returning True if both conditions are met, False otherwise.

Pseudo-code:



Genesis Block (Line 620)

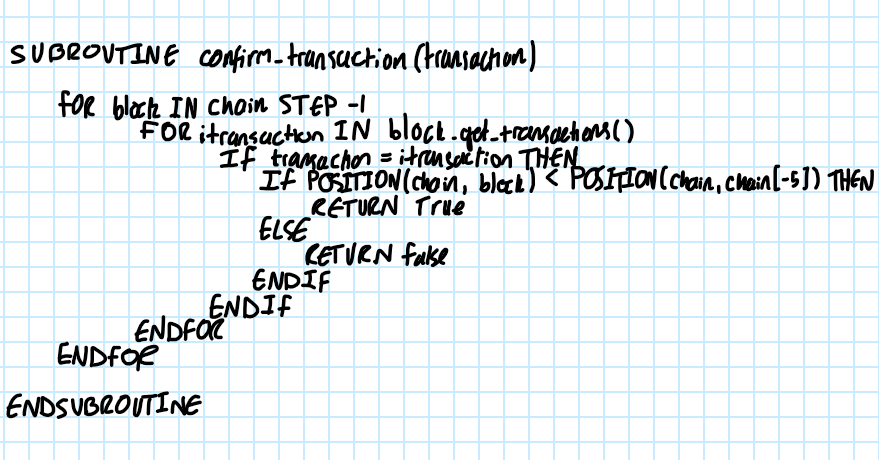
This algorithm creates the genesis block, the first block in the chain that must be hardcoded into the chain as it will not have an actual previous hash in the block header unlike the other blocks, and the transaction will be the issuance of the blockchain, meaning the starting currency (the only time currency will be generated other than block rewards). It works by setting the node’s wallet’s balance to however much the issuance is, and it records this as a transaction in the block so that it is in the transaction history of the node’s wallet (for future balance evaluating purposes).

* No pseudo-code as it just uses methods from different classes of the program.

Confirm Transaction (Line 667)

This algorithm takes a transaction and checks the blockchain history to see how many blocks far back the transaction is. A transaction can be considered confirmed when it is 6 blocks back or deeper, this has no use in the program but is a feature for the end-user for confidence in a block’s validity (the further back a block is, the more resistant it is to mutation as all blocks after a block must be remined and this must be done at a faster rate than all other miners combined: 51% attack)

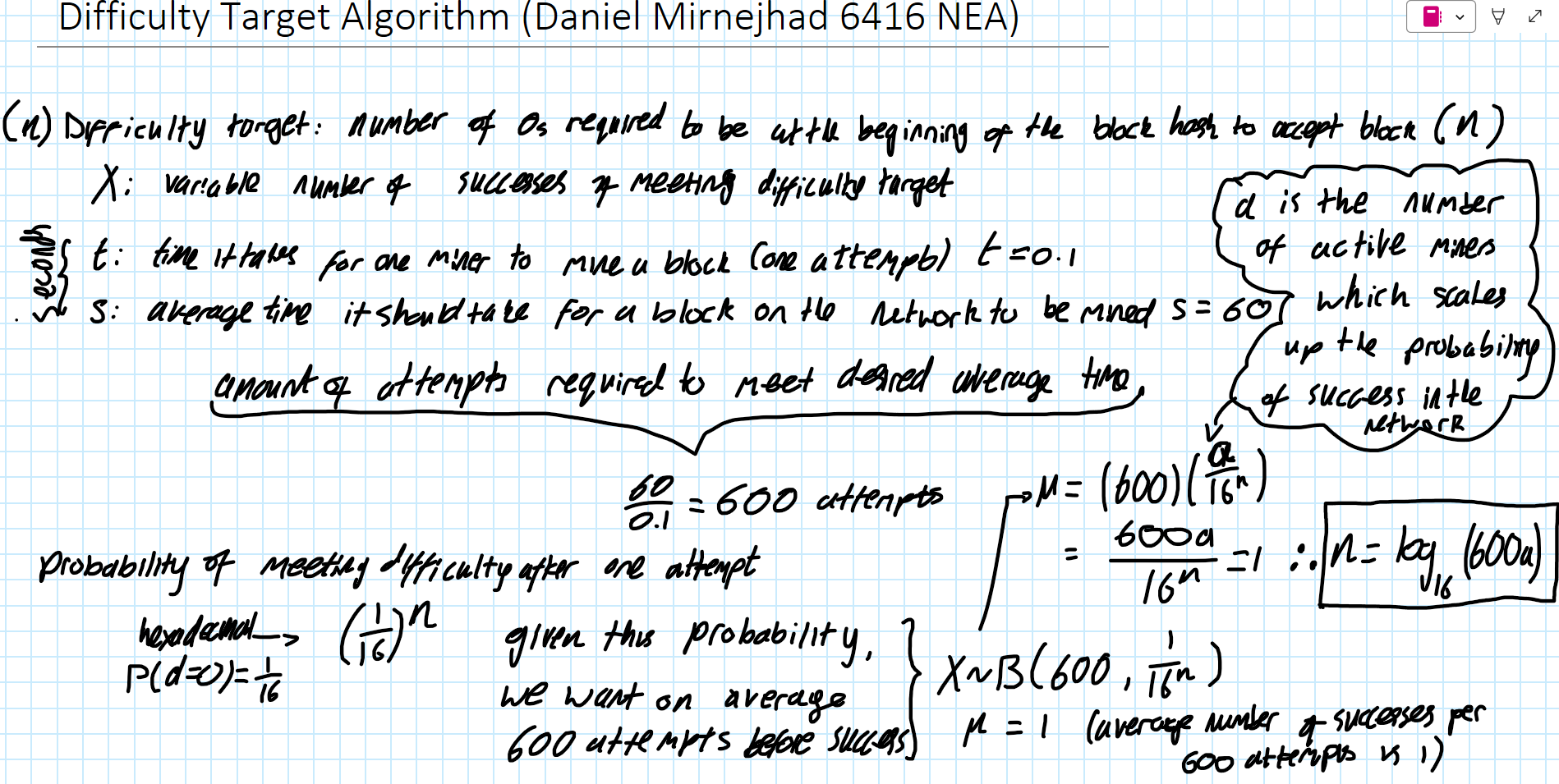
Pseudo-code:



Handle Transaction

This algorithm validates a received transaction from the client and if the transaction is valid, it adds it to the transaction histories of the wallets on both the sending and receiving ends so that it can be considered for future balance evaluations. (keep in mind that it gets added to the deserialised wallets, which are copies of the wallet objects that are broadcasted over the network, so this is later dealt with by updating the real wallets to match these). Then the transaction gets added to this copy of the blockchain’s transaction pool. If the transaction pool is now full, then the algorithm checks to see if this copy of the blockchain is being maintained by a miner node or just a regular node. If it is a miner node then the creation of a block using the transaction pool’s contents is initiated, then the transaction pool is emptied. If the node is just a regular node, then the transaction pool is just emptied.

Adjust Difficulty

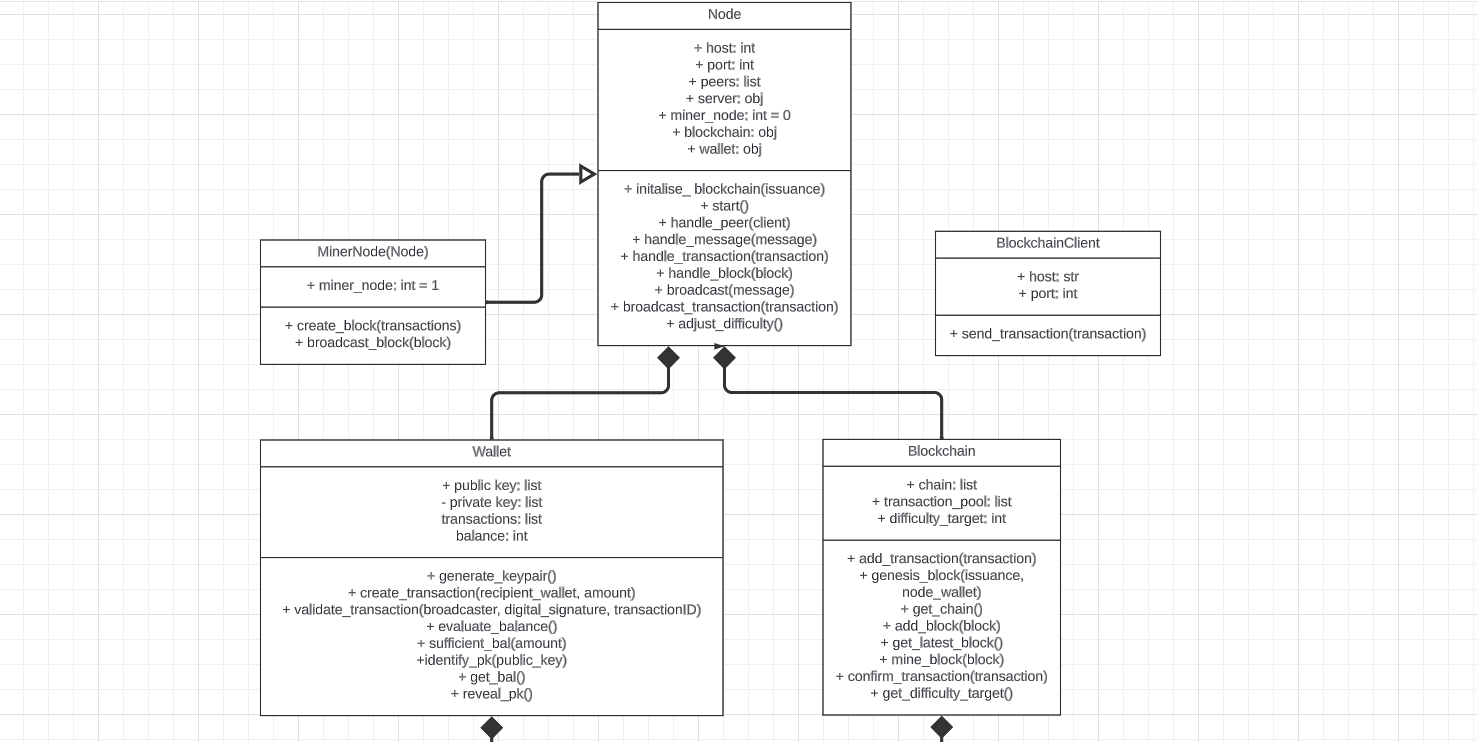


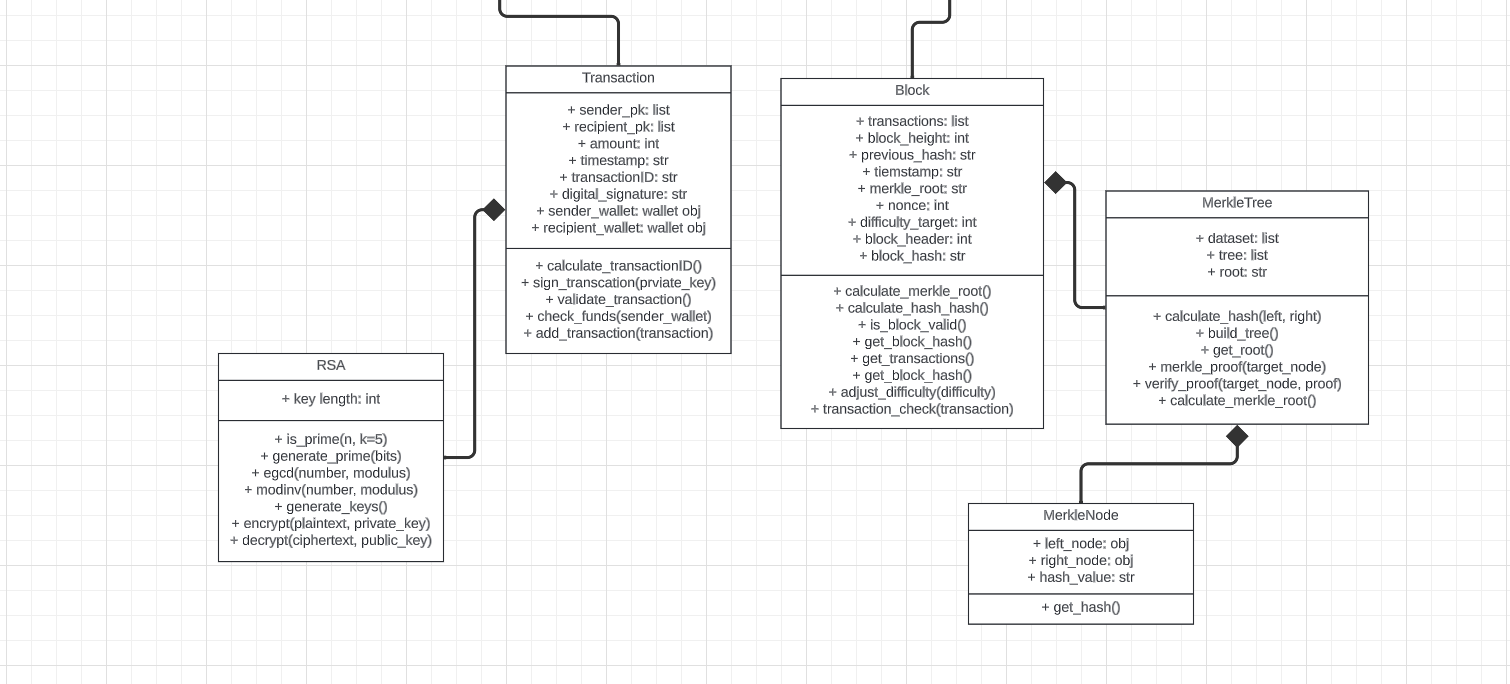
The algorithm just follows an equation that takes in the number of active miners and assumes time to go through one iteration of the mining process is 0.1 seconds (forced using time.sleep(0.1) in block mining) and also assumes the desired time we want it to take to mine a block to be 1 minute on average. The above image explains how I derived the equation using the binomial distribution.

Create Block

The purpose of this algorithm is to create a block with the valid transactions of a transaction pool, tackling the issue of – even though transactions may be valid individually, a wallet can be involved in multiple transactions in the same block, so they must be valid considering other relevant transactions too. This is done by iterating through each transaction, applying its potential effects on all relevant wallets, and then evaluating balances. If any balances turn out negative then the effects of the relevant transaction must be undone which is done by always keeping track of the state of the wallets in the previous iteration. After this process, there will be a list containing the transactions that are okay to go ahead as they don’t cause negative balances, preventing users from overspending, any invalid transactions will not go ahead in the block (as the effects on the wallets are undone after an invalid transaction), which can be checked using Merkle proofs, as seen in testing. On each iteration of successfully validated transactions, the valid transaction is used to update the actual wallets (as the function has been dealing with the deserialised copies of the relevant wallets due to the broadcasting of copies of the wallet objects and not the actual wallet objects) and the block is created using this list of valid transactions after all transactions have been checked using the methods in the block and blockchain classes.

**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. This method also makes sure that all no transactions go through that cause negative balance without altering the state of a wallet to check. |
| 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**

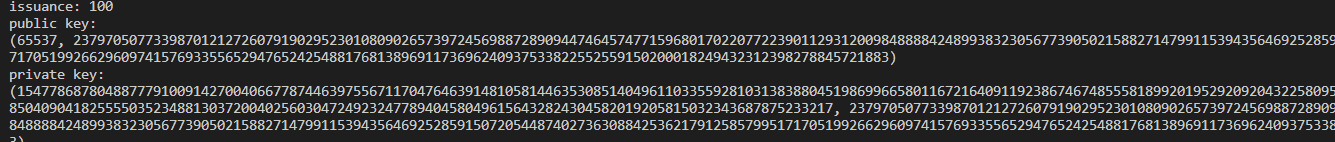


The main menu consists of two options, creating an account, or logging into an account. Logging into an account doesn’t involve retrieving the private key of an existing user, as then the blockchain would be centralised, which is not what the end-user wants, the end-user wants blockchain for the purpose of it being decentralised and its benefits of privacy.



If log into account is selected, the user must enter the public and private key they wish to make transaction with, invalid entries (non-existent public keys) are denied and the user is sent back to the main menu. The benefits of privacy from RSA are shown here, a user can log in with the wrong private key (as private keys are not checked for privacy reasons) but transactions will always be invalid due to RSA’s asymmetric key system (decrypting the digital signature with a fraud user’s public key will not end up returning the transaction ID so it will be invalid) – allowing the privacy that the end user wants from a blockchain system.





If create node is selected, an issuance (starting currency – the only way to generate currency apart from block rewards) is chosen for the blockchain (assumes the node is the origin node of the blockchain in this design). A public and private key are generated for the user using RSA encryption.



Resulting menu after creating a node or logging into a node (a menu with more privileges than users for managing the copy of the blockchain that the node has, and for simplicity and privacy sake as the end user wants the currency to be used by people, not everyone needs access to all information on the chain).



This is the menu for regular users, and I have shown the example choice 3 here which as returned the user’s history of transactions (in this example the user had no past transactions). They have less privileges for the sake of privacy.

# Technical Solution

Guide to Complex Functions

Evaluate Balance (Line 182)

This algorithm takes the transaction history of a wallet, and calculates a current balance by looking at each transaction’s amount and incrementing a count, which starts at 0, by the amount if the transaction’s recipient wallet’s public key matches the public key of the wallet (meaning this transaction had this wallet on the receiving end) or decrementing the count by the amount if the transaction’s sender wallet’s public key matches the public key of the wallet (meaning this transaction had this wallet on the sending end)

Build Tree (Line 392)

The purpose of this algorithm is to generate the Merkle tree given a set of data, which is a binary tree where the data makes up the leaf nodes and each of the nodes above are made up of the hashing and concatenation of the child nodes beneath it. It works as follows: Given a dataset the first layer (leaf nodes) are generated by using each element in the dataset and hashing (to hexadecimal) with a hashing function (sha-256 from hashlib library). Iteratively it will then look at every other node in the leaf nodes, making up a pair of nodes, the left node and right node if the right node exists as if the left node is at the end of the leaf nodes (odd number of nodes), then there would be no right node (although this is just for safety purposes as when the leaf layer is formed, it is forced to become even length through duplication), the pair of nodes will then generate the parent node after concatenating the string representations of the node and hashing it. This is done until the parent nodes of the current level are formed, and a while loop will keep taking the current level of nodes (which will be the parent nodes formed in the last iteration) and it will generate the next level of nodes with the same process until the root node is reached (only one node in the level).

Merkle Proof (Line 431)

This algorithm takes some data, generated a leaf node out of it as if it were in the Merkle tree and then iteratively loops through the elements list of nodes that make up the tree, picking up all the nodes that it would need to directly concatenate and hash with (the other node in a pair of nodes that make up the parent node directly above it) that would lead to eventually reaching the root node. Upon reaching the root node, a list of all the nodes in this path has now been formed which is then returned.

Verify Proof (Line 459)

This algorithm takes some target node and its proof path and concatenates and hashes the target node with the first node in the path, taking the result, and concatenating and hashing this node with the next node in the path, and this is repeated until a final hash has been calculated. If the final hash is equal to the root node’s hash then it means the target node was in the tree to begin with, if the final hash is not equal to the root node’s hash then it means this target node is not in the tree, used for confirming whether some data was in the tree of transactions in a block.

Calculate Block Hash (Line 524)

This algorithm uses the SHA-256 hashing algorithm from the hashlib library to compute a hash representative of the block that includes information of the block including the hash of the previous block in the chain. The algorithm uses a while loop to keep reassigning the nonce value attribute of the block, providing a different hash value of the block each time, checking if the hash meets the difficulty target (the minimum amount of leading 0s the hash of the block must start with). Once a hash meeting this condition is found, it is used as the hash of the block given the corresponding nonce value that contributed to the result, and the block mining process is complete.

Is Block Valid (Line 541)

This algorithm check that the hash of a given block meets the difficulty target (starts with the right amount of 0s set by the network to be allowed to be added to the chain as a block), and that each transaction in the block’s set of transactions is valid by checking the wallet making the transaction has the wallet’s private key too and isn’t just impersonating with another user’s public key (through RSA decryption methods previously built) and by checking that each transaction does not cause any users to have negative balance, returning True if both conditions are met, False otherwise.

Handle Transaction (Line 792)

This algorithm validates a received transaction from the client and if the transaction is valid, it adds it to the transaction histories of the wallets on both the sending and receiving ends so that it can be considered for future balance evaluations. (keep in mind that it gets added to the deserialised wallets, which are copies of the wallet objects that are broadcasted over the network, so this is later dealt with by updating the real wallets to match these). Then the transaction gets added to this copy of the blockchain’s transaction pool. If the transaction pool is now full, then the algorithm checks to see if this copy of the blockchain is being maintained by a miner node or just a regular node. If it is a miner node then the creation of a block using the transaction pool’s contents is initiated, then the transaction pool is emptied. If the node is just a regular node, then the transaction pool is just emptied.

Create Block (Line 863)

The purpose of this algorithm is to create a block with the valid transactions of a transaction pool, tackling the issue of – even though transactions may be valid individually, a wallet can be involved in multiple transactions in the same block, so they must be valid considering other relevant transactions too. This is done by iterating through each transaction, applying its potential effects on all relevant wallets, and then evaluating balances. If any balances turn out negative then the effects of the relevant transaction must be undone which is done by always keeping track of the state of the wallets in the previous iteration. After this process, there will be a list containing the transactions that are okay to go ahead as they don’t cause negative balances, preventing users from overspending, any invalid transactions will not go ahead in the block (as the effects on the wallets are undone after an invalid transaction), which can be checked using Merkle proofs, as seen in testing. On each iteration of successfully validated transactions, the valid transaction is used to update the actual wallets (as the function has been dealing with the deserialised copies of the relevant wallets due to the broadcasting of copies of the wallet objects and not the actual wallet objects) and the block is created using this list of valid transactions after all transactions have been checked using the methods in the block and blockchain classes.

User Access

Users get access to 5 actions they can perform, creating a transaction, which involves the program taking in an input of a public key to make the transaction with, then converting the resulting string into a tuple (which is the form it usually takes) using a library, then using methods created in the previous classes to send the transaction, but the program checks this using an attribute in the interface class which keeps track of the existing wallets, looping through looking to see if the public key inputted matches any of them, however the first wallet always references the node that holds the copy of the blockchain so it handles that case differently, as it needs to access the public key through the wallet attribute of the node object, all other actions that the user can perform is just calling methods that are made in other classes.

**The Code:**

1    *'''this github repository and the following code is the work of Daniel Mirnejhad (candidate number 6416, centre number 20570) for the AQA A-level Computer Science Non-Examination Assessment'''*

2

3    debug\_mode = 0

4    desired\_mining\_time = 60

5    import hashlib  *# hashing function (sha-256)*

6    from datetime import datetime  *# timestamps in blocks*

7    import random  *# example dataset generation*

8    import time  *# testing mining times*

9    import socket  *# blockchain network*

10   import threading  *# efficient handling of multiple connections between nodes at once*

11   import pickle  *# for message serialisation and deserialisation*

12   import enum  *# for differentiating between transactions and blocks over broadcasting in the network*

13   import math  *# used for logarithms to calculate difficulty target*

14   import ast  *# used to take string copy of public key into tuple*

15

16   test = False  *# used for testing individual classes (not the final test)*

17

18

19   class ExampleDataset:

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

21

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

23          self.dataset = []

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

25          self.data\_gen()

26

27      def get\_dataset(self):

28          return self.dataset

29

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

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

32              string = f'Data{i}'

33              self.dataset.append(string)

34

35

36   class RSA:

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

38

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

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

41

42      def is\_prime(self, n, k=5):

43          *"""Miller-Rabin primality test, test if a number is prime (test is repeated as it is not perfectly accurate).*

44    *name: is\_prime*

45    *parameters: n (number to test to see if it is a prime number), k (amount of times to repeat test for more accuracy)*

46    *returns: Boolean (True if n is prime, False if n is not prime)*

47    *"""*

48          if n <= 1 or n % 2 == 0:

49              return False

50          if n == 2 or n == 3:

51              return True

52

53          r, d = 0, n - 1  *# n = 2^r \* d + 1 (to test for primality)*

54          while d % 2 == 0:

55              r += 1

56              d //= 2

57

58          for \_ in range(

59                  k):  *# loop for trying different values of a, repeated iterations of test - for higher accuracy of generating a prime*

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

61              x = pow(a, d, n)  *# a to the power of d, modulo n*

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

63                  continue

64              for \_ in range(

65                      r - 1):  *# \_ indicates variable looped through is not actually used in the loop, for readability*

66                  x = pow(x, 2, n)  *# x squared, modulo n*

67                  if x == n - 1:

68                      break

69              else:

70                  return False

71          return True

72

73      def generate\_prime(self, bits):

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

75    *name: generate\_prime*

76    *parameters: bits (size of generated prime number)*

77    *returns: num (prime number that is found using is\_prime method)*

78    *"""*

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

80              num = random.getrandbits(bits)

81              if self.is\_prime(num):

82                  return num

83

84      def egcd(self, a, b):

85          *"""Extended Euclidean Algorithm to find the greatest common divisor between two numbers for finding modular inverses and finds the coefficients of the two numbers*

86    *such that the linear combination of the two numbers result in the greatest common divisor between them.*

87    *name: egcd*

88    *parameters: a, b (two values for which we find the greatest common divisor of)*

89    *returns: g, x, y (g is the greatest common divisor, x and y are the coefficients of the two numbers such that the linear combination of the two numbers result in the GCD*

90    *"""*

91          if a == 0:

92              return (b, 0, 1)

93          else:

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

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

96

97      def modinv(self, a, m):

98          *"""calculates the modular multiplicative inverse of an integer a modulo m, which is an integer x such that a multiplied by x, modulo m, is congruent to 1.*

99    *name: modinv*

100   *parameters: a, m (a is multiplied by some value x and then is taken modulo m such that it is congruent to 1)*

101   *returns: x modulo m (the modular inverse of a modulo m)*

102   *"""*

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

104          if g != 1:

105              raise Exception('modular inverse does not exist')

106          else:

107              return x % m

108

109      def generate\_keys(self):

110          *''' generates mathematically linked public-private key pair using RSA encryption - generating two large primes and using the fact that given a number that is the product*

111   *of two large primes it is computationally infeasible to derive the two primes from it and using the properties of modular exponentiation*

112   *name: generate\_keys*

113   *parameters: None*

114   *returns: public-private key pair (both including the product of primes and an exponent which undo eachothers effects when exponentiating the product of primes by both exponents modulo phi )*

115   *'''*

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

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

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

119          phi = (p - 1) \* (q - 1)  *# Compute euler totient function value of p and q (phi)*

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

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

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

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

124          return public\_key, private\_key

125

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

127          *'''encryption for signing transactions by exponentiating by private exponent which can be undone by public exponent to confirm the holder of the public key has the private key*

128   *name: encrypt*

129   *parameters: plaintext (some message to encrypt using RSA), d (the private exponent to encrypt message which can be undone (decryption) by public exponent), n (mod value of process)*

130   *returns: cipher\_text (the encrypted message)*

131   *'''*

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

133          return cipher\_text

134

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

136          *'''decryption for verifying digital signatures by properties of modular exponentiation (undoing the effects of exponentiation from private exponent using exponentiation of public)*

137   *name: decrypt*

138   *parameters: cipher\_text (some encrypted message to decrypt using RSA), e (the public exponent to decrypt ciphertext by exponentiation), n (mod value of process)*

139   *returns: plain\_text (decrypted ciphertext)*

140   *'''*

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

142          return plain\_text

143

144

145  class Wallet:

146

147      def \_\_init\_\_(self):

148          self.public\_key = None

149          self.\_private\_key = None

150          self.transactions = []

151          self.\_balance = 0

152

153      def generate\_keypair(self):

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

155   *name: generate\_keypair*

156   *parameters: None*

157   *returns: None*

158   *'''*

159          self.public\_key, self.\_private\_key = RSA().generate\_keys()

160

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

162          *''' create a transaction object representing a transaction between self (sending from this wallet) and a different wallet*

163   *name: create\_transaction*

164   *parameters: recipient\_wallet (the wallet that will be on the receiving end of the transaction), amount (amount sent)*

165   *returns: transaction (a transaction object representing this transaction between this wallet and the receiver given some amount)*

166   *'''*

167          if amount < 0:

168              print('invalid transaction')

169              return

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

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

172                                    recipient\_wallet)  *# automatically signs transaction*

173          *# print(transaction)*

174          return transaction

175

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

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

178   *name: validate\_transaction*

179   *parameters: broadcaster (wallet of the user that has created the transaction), digital\_signature (digital\_signature of the transaction made by the broadcaster encrypting*

180   *with their private key) transactionID (transactionID of the transaction representing the transaction's contents and date which is used to compare against decrypted digital\_signature for verification)*

181   *returns: Boolean (True if transactionID matches decrypted digital\_signature, False if not)*

182   *'''*

183          broadcaster\_pk = broadcaster.public\_key

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

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

186          if plain\_text == transactionID:

187              return True

188          else:

189              return False

190

191      def evaluate\_balance(self):

192          *'''check record of transactions involving the wallet (which are stored on the wallet as a part of their transaction history) and evaluate a final balance*

193   *name: evaluate\_balance*

194   *parameters: None*

195   *returns: balance (current balance given transaction history which accounts for income and outcome)*

196   *'''*

197          balance = 0

198          for transaction in self.transactions:

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

200                  balance += transaction.amount

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

202                  balance -= transaction.amount

203          if balance >= 0:

204              self.\_balance = balance

205          return balance

206

207      def sufficient\_bal(self, amount):

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

209   *name: sufficient\_bal*

210   *parameters: amount (unconfirmed transaction's amount that is specified to be leaving the user's balance)*

211   *returns: Boolean (True if user has enough to allow transaction to go through, False if it would cause negative balance)*

212   *'''*

213          global debug\_mode

214          balance = 0

215          for transaction in self.transactions:

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

217                  balance += transaction.amount

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

219                  balance -= transaction.amount

220          if debug\_mode == 1:

221              print(f'current calculated balance: {balance} vs 0')

222          if balance >= 0:

223              return True

224          elif balance < amount:

225              return False

226

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

228          *''' add transaction to wallet's history of transactions*

229   *name: add\_transaction*

230   *parameters: transaction\_obj (the transaction object that is going to be added to the history of transactions)*

231   *returns: None*

232   *'''*

233          self.transactions.append(transaction\_obj)

234

235      def identify\_pk(self, pk):

236          *'''check if a given public key is the same as this wallet's public key*

237   *name: identify\_pk*

238   *parameters: pk (public key to check against)*

239   *returns: self (public key of this wallet if they are equal, else None)*

240   *'''*

241          if self.public\_key == pk:

242              return self

243

244      def get\_bal(self):

245          *'''returns current balance of this wallet as it is a protected attribute and shouldn't be accessed directly*

246   *name: get\_bal*

247   *parameters: None*

248   *returns: self.\_balance (current balance of this wallet)*

249   *'''*

250          return self.\_balance

251

252      def reveal\_pk(self):

253          *'''returns the public key of this wallet*

254   *name: reveal\_pk*

255   *parameters: None*

256   *returns: self.public\_key (public key of this wallet)*

257   *'''*

258          return self.public\_key

259

260      def update\_wallet(self, deserialised\_wallet):

261          *'''takes a deserialised wallet and updates the current wallet to match the deserialised wallet as when a wallet is transferred over the network it is serialised and deserialised*

262   *for efficiency which leads to a copy of the wallet object being sent, so any updates on the transmitted wallet such as balance updates are not actually affecting the real wallet*

263   *name: update\_wallet*

264   *parameters: deserialised\_wallet (the wallet that had changes made to it after being broadcasted across the network)*

265   *returns: None*

266   *'''*

267          self.transactions = deserialised\_wallet.transactions

268          self.\_balance = deserialised\_wallet.\_balance

269

270

271  class Transaction():

272

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

274          self.sender\_pk = sender\_pk

275          self.recipient\_pk = recipient\_pk

276          self.amount = amount

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

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

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

280          self.sender\_wallet = sender\_wallet

281          self.recipient\_wallet = recipient\_wallet

282

283      def calculate\_transactionID(self):

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

285   *name: calculate\_transactionID*

286   *parameters: None*

287   *returns: transactionID (calculated by hashing (sha-256 library function) the contents of the transaction such as the involved user's public keys, the amount and the time of creation)*

288   *'''*

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

290          transactionID = hashlib.sha256(

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

292          return transactionID

293

294      def sign\_transaction(self, private\_key):

295          *'''encryption for signing transactions, same process as encryption in RSA class but for ease of use, it is in this class too, as some parameters are no longer needed*

296   *name: sign\_transaction*

297   *parameters: private\_key (used for the encryption process of exponentiation with the private exponent, modulo n, n is also in the private key)*

298   *returns: digital\_signature (the resulting ciphertext of the encryption process which can be decrypted using the public key mathematically linked to the used private key)*

299   *'''*

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

301                               self.transactionID]  *# pow function is exponentiation, third input is modular exponentiation*

302          return digital\_signature

303

304      def validate\_transaction(self):

305          *'''decryption for verifying digital signatures, ssame process as decryption in RSA class but for ease of use, it is in this class too, as some parameters are no longer needed)*

306   *name: validate\_transaction*

307   *parameters: None*

308   *returns: Boolean (True if public key used to decrypt is mathematically linked to the private key used to encrypt to make the digital signature, else False)*

309   *'''*

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

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

312                                self.digital\_signature])  *# pow function is exponentiation, third input is modular exponentiation*

313          if decryption == self.transactionID:

314              return True

315          else:

316              return False

317

318      def check\_funds(self, sender):

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

320   *name: check\_funds*

321   *parameters: sender (wallet of the user making the transaction to call sufficient\_bal method on the wallet to check transaction history to see if user has enough to make transaction)*

322   *returns: check (the Boolean value returned from the sufficient\_bal method that takes the argument self.amount - amount specified in transaction)*

323   *'''*

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

325          return check

326

327      def update\_records(self):

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

329   *name: update\_records*

330   *parameters: None*

331   *returns: None*

332   *'''*

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

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

335

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

337          receiver\_obj.add\_transaction(self)

338

339      def \_\_repr\_\_(self):

340          return (

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

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

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

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

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

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

347          )

348

349

350  '''Wallet Generation & Transactions Testing'''

351

352  if test == True:

353      *# user / wallet generation*

354      Me = Wallet()

355      Me.generate\_keypair()

356      You = Wallet()

357      You.generate\_keypair()

358

359      *# transaction between users*

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

361      print('Example Transaction between 2 Users:')

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

363

364      *# validating transaction*

365      print('Verifying Digital Signature:')

366      print(NewTransaction.validate\_transaction())

367      print('Checking Balances for Sufficient Funds:')

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

369

370

371  class MerkleNode:

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

373

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

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

376          self.left\_node = left\_node

377          self.right\_node = right\_node

378          self.hash = hash\_value

379

380      def get\_hash(self):

381          *'''returns the hash representation of the concatenation of the left and right child nodes*

382   *name: get\_hash*

383   *parameters: None*

384   *returns: self.hash (hash of the concatenation of the left and right child node that make up this parent node)*

385   *'''*

386          return self.hash

387

388

389  class MerkleTree:

390

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

392          self.dataset = dataset

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

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

395

396      def calculate\_hash(self, left,

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

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

399   *name: calculate\_hash*

400   *parameters: left, right (left and right child nodes that will be used as the hash inputs for the parent node)*

401   *returns: hashed (hexadecimal hash for the parent node)*

402   *'''*

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

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

405          return hashed

406

407      def build\_tree(self):

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

409   *name: build\_tree*

410   *parameters: None*

411   *returns: tree (the merkle tree, where the leaf nodes consist of the data and the nodes above are all made from the hashes of the two child nodes beneath it)*

412   *'''*

413          leaf\_nodes = []

414          if len(self.dataset) % 2 != 0:  *# add hashed dataset values into leaf level in string form*

415              self.dataset.append(self.dataset[-1])

416          for data in self.dataset:

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

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

419              leaf\_nodes.append(hashed\_data)

420

421          tree = [leaf\_nodes]  *# generate parent nodes from child nodes in previous level*

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

423              parent\_nodes = []

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

425                  left\_node = node

426                  if left\_node != tree[-1][

427                      -1]:  *# if left node isnt the last node then there is a right node (for checking purposes)*

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

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

430                  else:

431                      right\_node = None

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

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

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

435              tree.append(parent\_nodes)

436          return (tree)

437

438      def get\_root(self):

439          *'''returns the merkle root which is the root node of the merkle tree*

440   *name: get\_root*

441   *parameters: None*

442   *returns: root (root node object of the tree, of class MerkleNode)*

443   *'''*

444          root = self.tree[-1][0]

445          return root

446

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

448          *'''generates the sibling nodes that are in the path the target node takes to the root, providing a path of nodes you would need to concatenate and hash some leaf node with*

449   *to reach the root node, this is used to verify a transaction is actually in the tree without looking at the information of the other data*

450   *name: merkle\_proof*

451   *parameters: target\_node (the data that the proof path will be generated for, which will be converted into what it would be if it was a leaf node in the tree)*

452   *returns: proof\_path (the path of nodes this leaf node would concatenate and hash with to reach the root node if it were in the merkle tree in the first place)*

453   *'''*

454          target\_node = hashlib.sha256(

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

456          proof\_path = []

457          root\_reached = False

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

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

460              for i in range(0, len(self.tree[current\_level]),

461                             2):  *# pick up sibling nodes during traversal and add to proof path*

462                  left\_node = self.tree[current\_level][i]

463                  right\_node = self.tree[current\_level][i + 1] if i + 1 < len(self.tree[current\_level]) else None

464                  if left\_node == target\_node:  *# check if target node is either of the nodes just defined in the pair*

465                      proof\_path.append(right\_node)

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

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

468                  elif right\_node != None:

469                      if right\_node == target\_node:

470                          proof\_path.append(left\_node)

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

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

473              if current\_level + 1 < len(self.tree) and len(self.tree[

474                                                                current\_level + 1]) == 1:  *# check if the next level is within bounds and has a single root node*

475                  root\_reached = True  *# don't search the next level (not needed for the proof path)*

476              else:

477                  current\_level += 1  *# search the next level*

478          return proof\_path

479

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

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

482   *name: verify\_proof*

483   *parameters: target\_node (the data to check if it is in the tree), proof (the proof path the node representing the data will need to follow to eventually make the root node if it is in the tree)*

484   *returns: Boolean (True if target\_node is in the merkle tree, False if it is not in the merkle tree)*

485   *'''*

486          target\_node = hashlib.sha256(

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

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

489              current\_level = proof.index(

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

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

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

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

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

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

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

497              return True

498          else:

499              return False

500

501

502  '''Merkle Tree Testing'''

503

504  if test == True:

505      dataset1 = ExampleDataset(4).get\_dataset()  *# generate example dataset*

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

507      print('Merkle Tree:')

508      print(tree1.tree)

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

510      print('Merkle Proof:')

511      print(proof)

512      print('Proof Verification:')

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

514

515

516  class Block:

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

518

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

520          self.transactions = transactions

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

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

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

524          else:

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

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

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

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

529          self.difficulty\_target = blockchain.get\_difficulty\_target()  *# before block creation, node calculates difficulty target*

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

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

532                              timestamp = {self.timestamp},

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

534                              transactions = {self.transactions},

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

536          self.block\_hash = None

537

538      def calculate\_merkle\_root(self):

539          *'''calculate merkle root from transaction list as the leaf nodes of the Merkle tree*

540   *name: calculate\_merkle\_root*

541   *parameters: None*

542   *returns: this\_merkle\_tree.get\_root() (the merkle root of the generated tree that represents the transactions of the block)*

543   *'''*

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

545          return this\_merkle\_tree.get\_root()  *# generate merkle tree and return merkle root*

546

547      def calculate\_block\_hash(self):

548          *'''take block header and hash it, if hash meets difficulty target (some amount of leading 0s in the hash), return, if not, increment nonce and repeat - this is the hash puzzle that must be solved to make a block*

549   *name: calculate\_block\_hash*

550   *parameters: None*

551   *returns: None*

552   *'''*

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

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

555          while block\_hash[0: (

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

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

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

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

560              *# time.sleep(0.1)*

561          self.block\_hash = block\_hash

562          print('block successfully mined')

563

564      def is\_block\_valid(self):

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

566   *name: is\_block\_valid*

567   *parameters: None*

568   *returns: Boolean (True if hash meets difficulty target by starting with enough 0s and if all transactions in block are valid, False if either condition is not met)*

569   *'''*

570          check = []

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

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

573              check.append(True)

574          else:

575              check.append(False)

576

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

578              pass

579              for transaction in self.transactions:

580                  check.append(transaction.validate\_transaction())

581

582          validate\_transactions()  *# transaction double spending prevented (no duplicate transactions)*

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

584          return all(check)

585

586      def get\_block\_hash(self):

587          *'''return block's hash as to not have to directly access the attribute*

588   *name: get\_block\_hash*

589   *parameters: None*

590   *returns: self.block\_hash (hash of the block's contents)*

591   *'''*

592          return self.block\_hash

593

594      def get\_transactions(self):

595          *'''return block's set of transactions*

596   *name: get\_transactions*

597   *parameters: None*

598   *returns: self.transactions (block's set of transactions)*

599   *'''*

600          return self.transactions

601

602      def get\_block\_header(self):

603          *'''return block's header which contains most of the information that determines a block's hash, it is used as part of the hash input*

604   *name: get\_block\_header*

605   *parameters: None*

606   *returns: self.block\_header (the information that determines a block's hash'''*

607          return self.block\_header

608

609      def adjust\_difficulty(self, difficulty):

610          *'''given a level of difficulty (amount of 0s the computed hash of the block needs to start with at least), set the difficulty target attribute*

611   *name: adjust\_difficulty*

612   *parameters: difficulty (amount of 0s the computed hash of the block needs to start with at least)*

613   *returns: None'''*

614          self.difficulty\_target = difficulty

615

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

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

618   *name: transaction\_check*

619   *parameters: transaction (the transaction to check if included in a block after the block's creation)*

620   *returns: None*

621   *'''*

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

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

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

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

626

627

628  class Blockchain():

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

630

631      def \_\_init\_\_(self):

632          self.chain = []

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

634          self.difficulty\_target = None  *# adjusted by network node directly on each block creation*

635

636      def add\_transaction(self, transaction):

637          *'''adds a transaction to the transaction pool of unconfirmed transactions waiting to be used for a block*

638   *name: add\_transaction*

639   *parameters: transaction (transaction to add to transaction pool)*

640   *returns: None*

641   *'''*

642          self.transaction\_pool.append(transaction)

643

644      def genesis\_block(self, issuance,

645                        node\_wallet):  *# issuance is the first amount of currency the program starts with (starting in the node's wallet)*

646          *'''first block on blockchain is hardcoded in as it has no previous hash and the transaction represents issuance ("printing" currency)*

647   *name: genesis\_block*

648   *parameters: issuance (amount to be hardcoded into the blockchain in the beginning, only other way to introduce currency is through block rewards)*

649   *node\_wallet (wallet of the node that is initialising the network, as the issuance goes to them and they can act as a participant on the network as all nodes can)*

650   *returns: genesis block (first block in the chain that introduces the issuance and has no actual previous hash unlike all other blocks so it is hardcoded into chain)*

651   *'''*

652          node\_wallet.\_balance = issuance  *# directly change balance of wallet of node to issuance to hardcode genesis block into chain*

653          genesis\_transaction = node\_wallet.create\_transaction(node\_wallet,

654                                                               issuance)  *# record issuance as the first transaction on the blockchain*

655          node\_wallet.add\_transaction(genesis\_transaction)

656          genesis\_block = Block([genesis\_transaction], self)

657          return genesis\_block

658

659      def get\_chain(self):

660          *'''returns the chain (list of blocks) in the blockchain*

661   *name: get\_chain*

662   *parameters: None*

663   *returns: self.chain (the array of blocks in the blockchain)*

664   *'''*

665          return self.chain

666

667      def add\_block(self, block):

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

669   *name: add\_block*

670   *parameters: block (block to add to chain, usually is called after creating and validating a block)*

671   *returns: None*

672   *'''*

673          self.chain.append(block)

674

675      def get\_latest\_block(self):

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

677   *name: get\_latest\_block*

678   *parameters: None*

679   *returns: self.chain[-1] (the last block added to the chain, used for the purpose of retrieving the hash value of the block that is coming before a new block)*

680   *'''*

681          return self.chain[-1]

682

683      def mine\_block(self, block):

684          *'''initiate mining process, solving hash puzzle (called by miner node)*

685   *name: mine\_block*

686   *parameters: block (unconfirmed block that holds the data of a block but has not been mined yet)*

687   *returns: block (after being mined, meaning a nonce value such that the hash of the block meets the difficulty target is found and added to the block's block header)*

688   *'''*

689          block.calculate\_block\_hash()

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

691          return block

692

693      def confirm\_transaction(self, transaction):

694          *'''given some transaction, will return whether the transaction is considered as 'confirmed' (is 6 blocks deep in the chain) as defence against malicious attacks*

695   *name: confirm\_transaction*

696   *parameters: transaction (transaction to search through blocks, from latest block, backwards through the chain)*

697   *returns: Boolean (True if transaction is in a block that is 6 blocks or more deeper in the chain, False if block is in 5 newest blocks added)*

698   *'''*

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

700              if transaction in block.get\_transactions():  *# searches for block containing transaction*

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

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

703                  if transaction\_depth >= 6:

704                      return True

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

706

707      def get\_difficulty\_target(self):

708          *'''returns current difficulty target of this copy of the blockchain*

709   *name: get\_difficulty\_target*

710   *parameters: None*

711   *returns: self.difficulty\_target (current difficulty target of this copy of the blockchain)*

712   *'''*

713          return self.difficulty\_target

714

715

716  '''Block & Blockchain Testing'''

717

718  if test == True:

719      *# blockchain and genesis block creation*

720      Blockchain1 = Blockchain()

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

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

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

724      print('Chain of Blockchain:')

725      print(Blockchain1.get\_chain())

726

727      *# block mining and creation*

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

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

730      print('New Block Created')

731      print('Mining Iterations:')

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

733      print('Difficulty Target (1) Met')

734

735      *# adding block to the blockchain*

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

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

738      print('Show New Chain:')

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

740

741

742  class MessageType(enum.Enum):

743      *'''defines incoming message types over the network, transaction or block, needed to handle serialisation differently'''*

744      TRANSACTION = 1

745      BLOCK = 2

746

747

748  class Node:

749      def \_\_init\_\_(self, host, port, client=None):

750          self.host = host

751          self.port = port

752          self.peer\_ports = []  *# List of connected peers (ports on localhost)*

753          self.server = None

754          self.miner\_node = 0  *# 0: is not a miner node, 1: is a miner node, used to check number of active miner nodes on network*

755          self.blockchain = Blockchain()  *# local copy of the network's blockchain*

756          self.wallet = Wallet()  *# nodes can act as participants on the network, making and receiving transactions*

757          self.listening = 1

758          self.client = client

759

760      def initialise\_blockchain(self, issuance):

761          *'''hardcode genesis block into chain and generate the issuance (starting currency in circulation of the blockchain) using the genesis\_block method of the blockchain class*

762   *name: initialise\_blockchain*

763   *parameters: issuance (amount of starting currency the node's wallet will initialise the blockchain network with)*

764   *returns: None*

765   *'''*

766          self.wallet.generate\_keypair()  *# generate public and private key for wallet of node*

767          genesis\_block = self.blockchain.genesis\_block(issuance, self.wallet)  *# create genesis block*

768          self.blockchain.add\_block(genesis\_block)  *# add genesis block to blockchain*

769

770      def start(self):

771          *'''initialises a server that listens for incoming messages: transactions from users and peer nodes, blocks from peer nodes*

772   *name: start*

773   *parameters: None*

774   *returns: None*

775   *'''*

776          global debug\_mode

777          try:

778              if self.listening == 1:

779                  self.server = socket.socket(socket.AF\_INET,

780                                              socket.SOCK\_STREAM)  *# socket used to listen for incoming messages from peer nodes*

781                  self.server.bind((self.host, self.port))

782                  self.server.listen(1)

783              if debug\_mode == 1:

784                  print(f"Node listening on {self.host}:{self.port}")

785

786              while self.listening == 1:  *# always running until connection is broken*

787                  client, address = self.server.accept()  *# block until a connection is established returning a new socket with the peer node*

788                  if debug\_mode == 1:

789                      print(f"Connection from {address}")

790

791                  peer\_thread = threading.Thread(target=self.handle\_peer, args=(

792                  client,))  *# threads allow for efficient handling of multiple connections*

793                  peer\_thread.start()  *# thread calls handle peer to manage incoming messages on establish connections*

794          except socket.error:

795              pass

796

797      def handle\_peer(self, client):

798          *'''manages communication with a connected peer node*

799   *name: handle\_peer*

800   *parameters: client (the client that the node will be receiving data from)*

801   *returns: None*

802   *'''*

803          while True:

804              try:

805                  data = client.recv(65536)

806                  if not data:

807                      break

808                  message = pickle.loads(data)  *# deserialisation*

809                  self.handle\_message(message)

810              except Exception as e:

811                  print(f"Error handling peer: {e}")

812                  break

813

814      def handle\_message(self, message):

815          *'''directs received messages to transaction handling or block handling*

816   *name: handle\_message*

817   *parameters: message (the received message from client)*

818   *returns: None*

819   *'''*

820          global debug\_mode

821          if message['type'] == MessageType.TRANSACTION:

822              self.handle\_transaction(message['data'])

823              if debug\_mode == 1:

824                  print('transcation received, node is handling...')

825          elif message['type'] == MessageType.BLOCK:

826              self.handle\_block(message['data'])

827

828      def handle\_transaction(self, transaction):

829          *'''processes (validates) and adds transaction to the node's copy of the transaction pool, initiates the creation of a block if the transaction pool is full after addition*

830   *name: handle\_transaction*

831   *parameters: transaction (the transaction received from client that is to be validated and, if valid, added to transaction pool to be added to a block)*

832   *returns: None*

833   *'''*

834          global debug\_mode

835          validation = transaction.validate\_transaction()  *# transaction is validated*

836          if validation == True:  *# check if transaction is valid*

837

838              self.blockchain.add\_transaction(

839                  transaction)  *# transaction is added to list of unconfirmed transactions on local copy of blockchain*

840              transaction.sender\_wallet.transactions.append(

841                  transaction)  *# add transaction to sender's history of transactions*

842              transaction.recipient\_wallet.transactions.append(

843                  transaction)  *# add transaction to recipient's history of transactions*

844              *# self.broadcast\_transaction(transaction) # broadcast the valid transaction to peer nodes for them to validate and add to their copies*

845

846              if len(self.blockchain.transaction\_pool) == 4:  *# if transaction pool limit reached, empty transaction pool*

847                  if self.miner\_node == 1:  *# if node is a miner node, create a block with transactions and empty transaction pool*

848                      self.create\_block(self.blockchain.transaction\_pool)

849                      self.blockchain.transaction\_pool = []

850                      if debug\_mode == 1:

851                          print(f'transaction pool is full, commencing block mining...')

852                  elif self.miner\_node == 0:  *# if node is not a miner node, just empty transaction pool*

853                      self.blockchain.transaction\_pool = []

854

855          elif validation == False:

856              print('invalid transaction')

857          print('transaction pool updated updated')

858

859      def handle\_block(self, block):

860          *''' processes (validates) and adds block to the node's copy of the blockchain*

861   *name: handle\_block*

862   *parameters: block (the block to be validated and if valid, added to the node's copy of the blockchain, block is typically broadcasted from a differnt node)*

863   *returns: None*

864   *'''*

865          validation = block.is\_block\_valid()  *# validate the block*

866          if validation == True:

867              self.blockchain.add\_block(block)  *# add block if its valid*

868          elif validation == False:

869              pass

870          print(f"Received block: {self.miner\_node} ")

871

872      def adjust\_difficulty(self, desired\_time=60):

873          *'''difficulty target algorithm to adjust the difficulty target each time a block is ready to start being mined again*

874   *name: adjust\_difficulty*

875   *parameters: None*

876   *returns: difficulty target (minimum amount of 0s all block hash's must start with to be)*

877   *'''*

878          active\_miners = 1

879          for node in self.peer\_ports:  *# check list of nodes to see how many active miners there are*

880              if node.miner\_node == 1:

881                  active\_miners += 1

882              elif node.miner\_node == 0:

883                  pass

884          t = 0.1  *# time delay between block mining iterations of incrementing nonce value*

885          attempts = desired\_time / t

886          difficulty\_target = math.log(attempts \* active\_miners,

887                                       16)  *# equation for difficulty target given active miners and time delay such that mining takes 1 minute*

888          self.blockchain.difficulty\_target = math.ceil(

889              difficulty\_target)  *# round difficulty target up (decimal target not possible)*

890          return math.ceil(difficulty\_target)

891

892

893  class MinerNode(Node):

894      *'''instances of this class can use the methods a typical node can use but they can also broadcast blocks to the network'''*

895

896      def \_\_init\_\_(self, host, port):

897          super().\_\_init\_\_(host, port)

898

899      def initialise\_miner(self):

900          *'''confirm a node is a miner node in the attributes of the node object, miner nodes will be able to mine a block when the transaction pool is full*

901   *name: initialise\_miner*

902   *parameters: None*

903   *returns: None*

904   *'''*

905          self.miner\_node = 1

906

907      def create\_block(self, transactions):

908          *'''validate transactions and make sure that transactions do not lead to negative balances (taking into account users being involved in multiple transactions in one set of transactions) and make a block with the valid transactions*

909   *name: create\_block*

910   *parameters: transactions (list of transactions to validate and use for a block)*

911   *returns: None*

912   *'''*

913          global debug\_mode

914          validated\_transactions = []

915          seen\_wallets = []

916          previous\_wallet = None  *# for undoing*

917          for transaction in transactions:

918              for i in seen\_wallets:  *# add transaction from transaction's wallets to updated wallet*

919                  if transaction.sender\_wallet.public\_key == i.public\_key:  *# update wallet to last iteration's version*

920                      previous\_wallet = i  *# keep track of previous wallet incase transaction ends up being invalid and wallet needs to get reverted*

921                      i.transactions.append(transaction)

922                      previous\_wallet\_pos = seen\_wallets.index(previous\_wallet)

923                      transaction.sender\_wallet = i

924                  elif transaction.recipient\_wallet.public\_key == i.public\_key:

925                      previous\_wallet = i

926                      i.transactions.append(transaction)

927                      transaction.recipient\_wallet = i

928                      previous\_wallet\_pos = seen\_wallets.index(previous\_wallet)

929              check = []  *# validate transactions (no causes in negative balance) after adding new transactions to users' wallets*

930              check.append(transaction.check\_funds(transaction.sender\_wallet))

931              check.append(transaction.validate\_transaction())  *# check if digital signature checks out*

932              if all(check) == False:  *# revert wallets back to initial state before transactions were added to evaluate balances, because the transaction is not going through*

933                  if previous\_wallet != None:

934                      previous\_wallet.transactions.pop()

935                      seen\_wallets[previous\_wallet\_pos] = previous\_wallet  *# revert last wallet mutated to original state*

936                      if debug\_mode == 1:

937                          print('invalid transaction')

938                  else:

939                      pass

940              elif all(check) == True:  *# transaction is going through*

941                  validated\_transactions.append(transaction)

942                  if debug\_mode == 1:

943                      print('transaction validated')

944                  transaction.sender\_wallet.evaluate\_balance()

945                  transaction.recipient\_wallet.evaluate\_balance()

946                  if self.client != None:  *# counter different instances in memory from serialisation problem (synchronise wallet instances)*

947                      for wallet in self.client.users:  *# actual wallets are all updated to have the same state as these deserialised wallets that had the transaction added*

948                          if wallet.public\_key == transaction.sender\_wallet.public\_key:

949                              if debug\_mode == 1:

950                                  print(f'updating wallet')

951                              seen\_flag = 0

952                              for j in seen\_wallets:

953                                  if wallet.public\_key == j.public\_key:

954                                      seen\_flag = 1

955                                      pos = seen\_wallets.index(j)

956                                      wallet.update\_wallet(transaction.sender\_wallet)

957                              if seen\_flag == 0:

958                                  seen\_wallets.append(transaction.sender\_wallet)

959                              else:

960                                  seen\_wallets[pos] = transaction.sender\_wallet

961                          elif wallet.public\_key == transaction.recipient\_wallet.public\_key:

962                              if debug\_mode == 1:

963                                  print(f'updating wallet')

964                              seen\_flag = 0

965                              for j in seen\_wallets:

966                                  if wallet.public\_key == j.public\_key:

967                                      seen\_flag = 1

968                                      pos = seen\_wallets.index(j)

969                                      wallet.update\_wallet(transaction.recipient\_wallet)

970                              if seen\_flag == 0:

971                                  seen\_wallets.append(transaction.recipient\_wallet)

972                              else:

973                                  seen\_wallets[pos] = transaction.recipient\_wallet

974              else:

975                  print(f'invalid transaction')

976          for k in self.client.users:

977              for l in seen\_wallets:

978                  if k.public\_key == l.public\_key:

979                      k.update\_wallet(l)

980          if len(validated\_transactions) != 0:

981              global desired\_mining\_time

982              difficulty\_target = self.adjust\_difficulty(

983                  desired\_mining\_time)  *# recalculates difficulty target before block creation*

984              if debug\_mode == 1:

985                  print(f'updated difficulty target: {difficulty\_target}')

986              block = Block(validated\_transactions, self.blockchain)  *# block is created*

987              block.calculate\_block\_hash()  *# block is mined*

988              block.is\_block\_valid()  *# block is validated*

989              self.blockchain.add\_block(block)  *# block is added to chain*

990

991          block\_reward = 5  *# reward for mining block, added into transaction pool*

992

993

994  class BlockchainClient:

995      *'''Users who do not run nodes interact (making transactions, broadcasting it to the nodes) with the blockchain through a blockchain client'''*

996

997      def \_\_init\_\_(self, host, port, node):

998          self.host = host

999          self.port = port

1000        self.node = node

1001        self.users = []

1002

1003    def send\_transaction(self, transaction):

1004        *'''send a transaction to the blockchain network'''*

1005        if transaction.sender\_wallet not in self.users:

1006            self.users.append(transaction.sender\_wallet)

1007            print('user added')

1008        if transaction.recipient\_wallet not in self.users:

1009            self.users.append(transaction.recipient\_wallet)

1010            print('user added')

1011        try:

1012            with socket.socket(socket.AF\_INET,

1013                               socket.SOCK\_STREAM) as client\_socket:  *# connect client to peer nodes on network*

1014                client\_socket.connect((self.host, self.port))

1015

1016                message = {'type': MessageType.TRANSACTION, 'data': transaction}  *# serialise transaction*

1017                serialised\_message = pickle.dumps(message)

1018

1019                client\_socket.send(serialised\_message)  *# send transaction to network*

1020

1021                print(f"Transaction sent from client")

1022        except Exception as e:

1023            print(f"Error sending transaction: {e}")

1024

1025

1026 '''Full Program Testing'''

1027

1028

1029 def start\_node\_in\_thread(node):

1030    node.start()

1031

1032

1033 testfin = False

1034 if testfin == True:

1035    user\_1 = Wallet()

1036    user\_1.generate\_keypair()

1037    user\_2 = Wallet()

1038    user\_2.generate\_keypair()

1039    user\_3 = Wallet()

1040    user\_3.generate\_keypair()

1041    print(f'user profiles created')

1042

1043    regular\_node = MinerNode('localhost', 5000)

1044    blockchain\_client = BlockchainClient('localhost', 5000,

1045                                         regular\_node)  *# users broadcast transactions to network through client*

1046    regular\_node.client = blockchain\_client

1047

1048    regular\_node.initialise\_blockchain(100)  *# create genesis blocks and introduce issuance into chain circulation*

1049    regular\_node.initialise\_miner()

1050

1051    node\_thread = threading.Thread(target=start\_node\_in\_thread, args=(regular\_node,))

1052    node\_thread.start()  *# connect to network and listen for incoming messages*

1053

1054    t1 = regular\_node.wallet.create\_transaction(user\_1, 11)

1055    t2 = regular\_node.wallet.create\_transaction(user\_2, 14)

1056    t3 = regular\_node.wallet.create\_transaction(user\_1, 6)

1057    t4 = regular\_node.wallet.create\_transaction(user\_3, 43)

1058    t5 = user\_2.create\_transaction(user\_3, 10)

1059    t6 = user\_3.create\_transaction(regular\_node.wallet, 1)

1060    t7 = user\_1.create\_transaction(regular\_node.wallet, 4)

1061    t8 = regular\_node.wallet.create\_transaction(user\_1, 300)

1062    token = [t4, t2, t1, t3, t8, t6, t7, t5]

1063    for ti in token:

1064        blockchain\_client.send\_transaction(ti)

1065        time.sleep(2)

1066    print(regular\_node.wallet.get\_bal())  *# 31*

1067    print(user\_1.get\_bal())  *# 13*

1068    print(user\_2.get\_bal())  *# 4*

1069    print(user\_3.get\_bal())  *# 52*

1070

1071    for bloc in regular\_node.blockchain.get\_chain()[1:]:

1072        print(f'amount of transactions in this block: {len(bloc.transactions)}')

1073        bloc.transaction\_check(t8)

1074

1075 *# prevent users from sending money to themself*

1076

1077 *# 1: users, client and nodes are generated*

1078 *# 2: users or nodes make transactions*

1079 *# 3: blockchain client sends transaction to node*

1080 *# 4: node validates transaction and broadcasts to other nodes for other nodes to validate*

1081 *# 5: transaction pool reaches limit*

1082 *# 6: transaction pool is emptied and used as transactions for a block*

1083 *# 7: merkle tree is generated*

1084 *# 8: block is mined by node, block reward is given to miner in next block, added to local copy of chain and broadcasted to all other nodes on network to validate and add*

1085 *# 9: transactions are successfully completed and cycle can restart*

1086

1087

1088 *# transaction pool is filled*

1089 *# miner nodes create blocks*

1090 *# miner nodes compete to mine block*

1091 *# winning miner broadcasts block to peer nodes*

1092 *# block reward*

1093

1094 '''User-End'''

1095

1096

1097 class interface:

1098

1099    def \_\_init\_\_(self):

1100        self.wallets = []

1101        self.main\_menu()

1102

1103    def main\_menu(self):

1104        *'''provides interface to create or log into an account (node or user that communicates through blockchain interface)*

1105  *parameters: None*

1106  *returns: None*

1107  *'''*

1108        menu1 = input('''create account (1) \nlog into account (2)''')

1109        if menu1 == '1':

1110            self.create\_account()

1111        elif menu1 == '2':

1112            self.login()

1113        else:

1114            print('invalid response')

1115            self.main\_menu()

1116        *# create account*

1117        *# log into existing account (show defence against impersonation through digital signatures by making transactions and getting them denied)*

1118

1119    def create\_account(self):

1120        *'''creates an account, regular user that communicates through a client or a node that holds a copy of the blockchain, generating a wallet for the end-user*

1121  *name: create\_account*

1122  *parameters: None*

1123  *returns: None*

1124  *'''*

1125        account\_type = input('create node (1) \ncreate user (2) \n')

1126        if account\_type == '1':

1127            node = MinerNode('localhost', 5000)

1128            node.initialise\_miner()

1129            if self.wallets != []:

1130                node.blockchain = self.wallets[0].blockchain

1131                node.client = self.wallets[0].client

1132                print(f'public key: \n{node.wallet.public\_key} \nprivate key: \n{node.wallet.\_private\_key}')

1133                self.node\_access(node)

1134            issuance = input('issuance: ')

1135            try:

1136                if int(issuance) > 0:  *# validating issuance*

1137                    node.initialise\_blockchain(int(issuance))

1138                    print(f'public key: \n{node.wallet.public\_key} \nprivate key: \n{node.wallet.\_private\_key}')

1139                    print(f'initialised blockchain with an issuance of {issuance} \n')

1140                else:

1141                    print('invalid issuance')

1142                    self.main\_menu()

1143            except Exception as e:

1144                print(f'invalid issuance: {e}')

1145                self.main\_menu()

1146            node.client = BlockchainClient('localhost', 5000, node)

1147            self.wallets.append(node)

1148            node\_thread = threading.Thread(target=start\_node\_in\_thread, args=(node,))

1149            node\_thread.start()  *# connects node to network to listen for incoming messages from client*

1150            self.node\_access(node)

1151        elif account\_type == '2':

1152            try:

1153                wallet = Wallet()

1154                wallet.generate\_keypair()

1155                self.wallets.append(wallet)

1156                self.wallets[0].client.users.append(wallet)

1157                print(f'public key: \n{wallet.public\_key} \nprivate key: \n{wallet.\_private\_key}')

1158                self.user\_access(wallet)

1159            except AttributeError:  *# no blockchain client for users to exist yet*

1160                print(f'no blockchain client for users')

1161                self.wallets.pop()

1162                self.main\_menu()

1163            except Exception as e:  *# debugging*

1164                print(f'{e}')

1165        else:

1166            print('invalid response')

1167            self.main\_menu()

1168

1169    def login(self):

1170        public\_key = input(f'public key: ')

1171        private\_key = input(f'private key: ')

1172        for wallet in self.wallets:

1173            if isinstance(wallet, Node) or isinstance(wallet, MinerNode):

1174                if str(wallet.wallet.public\_key) == str(public\_key):

1175                    wallet.wallet.\_private\_key = ast.literal\_eval(private\_key)

1176                    print('logging onto account')

1177                    self.node\_access(wallet)

1178            elif str(wallet.public\_key) == str(public\_key):

1179                print('logging onto account')

1180                wallet.\_private\_key = ast.literal\_eval(private\_key)

1181                self.user\_access(wallet)

1182        print('invalid response')

1183        self.main\_menu()

1184

1185    def node\_access(self, node):

1186        *'''provides the functionality for nodes, different to users as they get more in depth access on viewing the blockchain and controlling aspects of it like difficulty target*

1187  *name: node\_access*

1188  *parameters: node (the specific node that is maintaining the copy of the blockchain)*

1189  *returns: None*

1190  *'''*

1191        global debug\_mode

1192        node\_menu = input(

1193            '''create transaction (1) \nview balance (2) \nview history (3) \nfind transaction(4) \nview chain (5) \nview transaction pool (6) \nadjust  difficulty (7) \ntoggle debug mode (8) \nlogout (9) \n''')

1194        if node\_menu == '1':

1195            recipient = input('specify public key of recipient wallet: ')

1196            amount = input('choose desired amount to send: ')

1197            recipient\_pk = ast.literal\_eval(recipient)  *# convert string into tuple*

1198            wal = None

1199            for wallet in self.wallets:

1200                if isinstance(wallet, Node) or isinstance(wallet,

1201                                                          MinerNode):  *# find the wallet of the desired public key to send to (if recipient is node)*

1202                    if recipient\_pk == ast.literal\_eval(str(wallet.wallet.public\_key)):

1203                        wal = wallet.wallet

1204                        break

1205                elif recipient\_pk == ast.literal\_eval(

1206                        str(wallet.public\_key)):  *# find the wallet of the desired pubic key to send to (if recipient is just a user)*

1207                    wal = wallet

1208                    break

1209            if wal == None:

1210                print('user does not exist')

1211                self.node\_access(node)

1212            else:

1213                transaction = node.wallet.create\_transaction(wal, int(amount))  *# make transation*

1214                node.client.send\_transaction(

1215                    transaction)  *# broadcast transaction from blockchain client to node for transaction handling*

1216                self.node\_access(node)

1217        elif node\_menu == '2':

1218            print(node.wallet.get\_bal())

1219            self.node\_access(node)

1220        elif node\_menu == '3':

1221            print(node.wallet.transactions)

1222            self.node\_access(node)

1223        elif node\_menu == '4':

1224            transaction = input('transaction to look for (number in history): ')

1225            for block in self.wallets[0].blockchain.get\_chain()[1:]:

1226                block.transaction\_check(node.wallet.transactions[

1227                                            transaction - 1])  *# uses merkle proof for ech block past the genesis block to check for a transaction*

1228            self.node\_access(node)

1229        elif node\_menu == '5':

1230            print(node.blockchain.get\_chain())

1231            self.node\_access(node)

1232        elif node\_menu == '6':

1233            if debug\_mode == 1:

1234                print(node.blockchain.transaction\_pool)

1235            else:

1236                print([f'transaction' for i in node.blockchain.transaction\_pool])

1237            self.node\_access(node)

1238        elif node\_menu == '7':

1239            global desired\_mining\_time

1240            target = input('desired average time to mine a block: ')

1241            try:

1242                target = int(target)

1243                if target > 0:

1244                    desired\_mining\_time = target

1245            except Exception as e:

1246                print('target must be a postive number')

1247                self.node\_access(node)

1248            print('invalid response')

1249            self.node\_access(node)

1250        elif node\_menu == '8':

1251            if debug\_mode == 0:

1252                debug\_mode = 1  *# allows node to see debugging statements like prints in the mining process*

1253            elif debug\_mode == 1:

1254                debug\_mode = 0

1255            self.node\_access(node)

1256        elif node\_menu == '9':

1257            print(f'logging out of node, returning to main menu')

1258            self.main\_menu()

1259        else:

1260            print(f'invalid response')

1261            self.node\_access(node)

1262

1263    def user\_access(self, user):

1264        *'''provides the functionality regular users interacting through a blockchain interface*

1265  *name: user\_access*

1266  *parameters: user (which user is currently logged in)*

1267  *returns: None*

1268  *'''*

1269        node\_menu = input(

1270            '''create transaction (1) \nview balance (2) \nview history (3) \nfind transaction(4)  \nlogout (5) \n''')

1271        if node\_menu == '1':

1272            print(self.wallets)

1273            recipient = input('specify public key of recipient wallet: ')

1274            amount = input('choose desired amount to send: ')

1275            recipient\_pk = ast.literal\_eval(recipient)  *# convert string into tuple*

1276            wal = None

1277            for wallet in self.wallets:

1278                if isinstance(wallet, Node) or isinstance(wallet,

1279                                                          MinerNode):  *# find the wallet of the desired public key to send to (if recipient is node)*

1280                    if recipient\_pk == ast.literal\_eval(str(wallet.wallet.public\_key)):

1281                        wal = wallet.wallet

1282                        break

1283                elif recipient\_pk == ast.literal\_eval(

1284                        str(wallet.public\_key)):  *# find the wallet of the desired pubic key to send to (if recipient is just a user)*

1285                    wal = wallet

1286                    break

1287            if wal == None:

1288                print('user does not exist')

1289                self.user\_access(user)

1290            transaction = user.create\_transaction(wal, int(amount))

1291            self.wallets[0].client.send\_transaction(transaction)

1292            time.sleep(1)

1293            self.user\_access(user)

1294        elif node\_menu == '2':

1295            print(user.get\_bal())

1296            self.user\_access(user)

1297        elif node\_menu == '3':

1298            print(user.transactions)

1299            self.user\_access(user)

1300        elif node\_menu == '4':

1301            transaction = input('transaction to look for (number in history): ')

1302            for block in self.users[0].blockchain.get\_chain()[1:]:

1303                block.transaction\_check(user.transactions[transaction - 1])

1304            self.user\_access(user)

1305        elif node\_menu == '5':

1306            self.main\_menu()

1307        else:

1308            print(f'invalid response')

1309            self.user\_access(user)

1310

1311

1312 interface1 = interface()

# Testing

**Test Strategy**

I plan to test all the aspects of the program such as the implemented RSA encryption, how it provides privacy (no retrieving private key at any point) while preventing users from making fraudulent transactions, the process of preventing this using transaction IDs (which will be revealed on debug mode), the system’s capability to prevent overspending and how it can take account for users being involved in multiple transactions in a block without the system evaluating any transactions as valid (since invalid transactions could count as valid individually but invalid taking the context of the rest of the contents of the block), the use of Merkle trees and Merkle proofs to confirm if a transaction is in the chain or in a specific block without the user having access to all the transactions in a block for privacy. The testing will show the process of mining on debug mode and how the time it takes (on average) for the network to mine a block can be controlled using the difficulty target algorithm, and all of these will be explored through the creation of multiple transactions between users by logging in and out of their accounts (without ever using the retrieval of their private key, for privacy, which is possible through RSA). Testing all these will show the features provided by the completion of almost all my objectives stated previously.

1. Testing the creation of a node, and the creation of a user, and how two nodes on a network behave (second one takes the first node’s copy of the blockchain & uses its client).
2. Testing the privileges of a node before anything has happened, compared to user privileges.
3. Testing possible invalid inputs so far -> creating user before node, logging into account with non-existent public key, invalid responses for menus, negative or 0 issuance.
4. Testing creation of transactions, the transaction pool, and how it doesn’t affect the chain until a block is mined when the pool is full (genesis only in chain, failed Merkle proof search).
5. Creation of more users to test transactions between nodes and users, and users and users.
6. The system’s handling of the same wallet referenced multiple times in one block.
7. Testing and exploring how the blockchain client is used for users who don’t run a node.
8. Debug mode: looking at transactions, the process of mining and hashing, adjusting the block mining time, show how other users cannot use debug mode even if it is on for a node.
9. Response to invalid transaction (overspend) how it is excluded, doesn’t affect balances, and you cannot search for it using Merkle trees and Merkle proofs.
10. The results of different features after a block has been added.
11. Adding another block showing the same response to overspends given wallets are referenced multiple times, the results of features after this 3rd block.
12. Minor edge case testing like actions after creating blocks, more invalid responses such as negative amount transactions, sending to non-existent users, making user before client.
13. Response to fraudulent transactions, show RSA process through debug mode, another node.

**Test Video**

<https://youtu.be/GSi-EdultZo>



**Test Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Description** | **Test Data** | **Expected Outcome** | **Actual Outcome** | **Video Timestamp** |
| 1 | The creation of a node, the creation of users and how they interact through a client, first testing invalid issuance for node creation and creating user first error |  | invalid issuance: back to main menu, creating user first: back to main menu, creation of a node returning a public key, private key, and the node actions menu, successful log out to main menu and creation of a user, resulting in their public key, private key, and the user menu | as expected | 00:00 |
| 2 | Testing the default privileges of both nodes and users |  | node   * balance of 0 * no history of transactions * chain just has genesis block * transaction pool is empty * toggle debug mode on & off * logout to menu   user   * balance of 0 * no history of transactions * logging out | as expected | 01:28 |
| 3 | Logging in and out of accounts, how the system responds to invalid inputs so far, such as invalid responses, logging in with a non-existent public key |  | after invalid responses, end-user is brought back to the same menu they were in before, with an invalid input response in the terminal describing the specific invalid issue | as expected | 04:22 |
| 4 | creation of transactions and the effects of creating transactions on the transaction pool of the blockchain and the lack of effect on the history of a user until block is created, genesis block in chain, |  | transaction pool should have as many ‘representative’ transactions as made, this action on debug mode should print the full transaction objects, the history of the node making the transaction should still be empty | as expected | 08:15 |
| 5 | the system handling transactions referencing the same user multiple times in one block (no invalid transaction), proving it works through checking all users’ balances |  | check all users balances, the maths on the transactions check out.  block  Node -> user 1 (11)  Node -> user 2 (14)  Node -> user 1 (6)  Node -> user 3 (43)  Expected Balances  Node: 26, User 1: 17, User 2: 14, User 3: 43 | as expected | 10:40 |
| 6 | exploring debug mode, differences in node features after debug mode (such as seeing print statements of transaction validation during block creation, wallet synchronisation statements during block creation, changes in difficulty target, transaction pool full statements) and how switching to different users will not give them debug mode as it is a node only feature |  | series of print statements | almost all as expected, functionality of difficulty target working and works as normal and intended but print statement saying it has occurred does not print (not necessary) | done throughout the video |
| 7 | debug mode to see the process of mining, using the adjust difficulty feature for nodes before mining the second block to see how it can affect how long it takes to mine a block by using the algorithm |  | make desired average time to mine greater to increase difficulty target, see visual effects of longer mining process, print statement of contents of the block upon mining completion | works as intended, expected time changed to two minutes from 1 second, block mining took about that much longer, but no print statement (still works and is functional). | 11:57 – shown that it worked after transactions made, after the next test – test 8, seen at 21:03 |
| 8 | creating transactions (edge cases) user to node, user to user, node to user, and an invalid transaction |  | Block:  user 2 -> user 3 (10)  user 3 -> node (1)  user 1 -> node (4)  node -> user 1 (300)  last transaction should be invalid.  expected balances:  node: 31  user 1: 13  user 2: 4  user 3: 52 | as expected | 14:16 |
| 9 | the invalid transaction will not be invalid on its own but invalid given the context of the rest of the transactions in that block (overspend), |  | signs that the overspend transaction is ignored such as the balances not taking the transaction into account | the math is consistent meaning the overspend transaction at the end was ignored. (node balance is 31) | 21:16 |
| 10 | results of different features for node and other users after the second block is added to the chain |  | view chain (3 block objects now)  view different transaction histories.  node: 7 transactions  user 1: 3 transactions  user 2: 2 transactions  user 3: 3 transactions | as expected | checked simultaneously with test 8 and 9 |
| 11 | system response to fraudulent transactions (logging in with a public key and incorrect private key) – unable to make transactions |  | log into a user with wrong private key, try make transaction (fails). | as expected | 29:58 |

# Evaluation

Evaluation of Each Objective

RSA Class – Class Functions

RSA Class- Key Pair Generation

RSA Class - RSA Encryption & Decryption

Transaction Class - Transaction Signing & Verifying Functions

* These 4 objectives have been met.
* Test Evidence: public and private keys are generated through RSA encryption and used to represent users on the blockchain network, the evidence is that users cannot make fraudulent transaction in other users’ names even with access to their public key because upon making a transaction a digital signature is made using encryption from the private key the user has logged in with and to validate a transaction after it is made the digital signature is decrypted with the public key to return what was originally encrypted (the transaction ID which is in the transaction object for checking against), which will not work if it was not encrypted with the corresponding private key in the first place.

Transaction Class and Wallet Class – Transaction History and Balance Checking

* This objective has been met.
* The program successfully can return the history and balance on multiple occasions as seen.

Transaction Class – Transaction Structure (initialisation)

Transaction Class and Wallet Class – Sufficient Funds Validation

Node Class & Blockchain Class – Overspend Balance Deriving Function

* These 3 objectives have been met.
* The program can make a block where the transactions consist of users that have been referenced multiple times and even though the individual transactions may be valid (sufficient funds) a transaction given the context of the transactions before it in the set of transactions that will be used for the block may be invalid, the program successfully ignores this transaction in the last example of block creation during testing, as the transaction does not appear in the block (tested through Merkle proofs) and does not appear in either the sender or recipient’s transaction history.

Transaction Class - Transaction Serialisation & Deserialization Functions

Block Serialisation & Deserialization Functions

* These 2 objectives have been met.
* During the process explained in the previous objective, the program constantly updates the wallets (synchronising them as serialised data which is deserialised returns a copy of the original object that was serialised, so mutating values of this copy like balance doesn’t change the original wallet, requiring synchronisation upon these changes which is shown in debug mode)

Network Communication Class – User to Node (to network to nodes later)

Transaction Class - Transaction Broadcasting Function

* These 2 objectives have been met.
* The blockchain client broadcasts the transactions received by users communicating through the client to the node which is seen by the change in the transaction pool of the blockchain as transactions are made, allowing users to participate in the network without a node.

Blockchain Data Structure – Chain Mutation Functions

* This objective has been met.
* The chain is seen to be mutated repeatedly as new blocks are added.

Blockchain Data Structure – Hard Coded Genesis Block

* This objective has been met.
* The genesis block can be seen hardcoded into the chain when the node actions are tested out for the first time, the chain just has one block in it which is before any block creation.

Blockchain Data Structure – Transaction Pool (Mem-pool)

* This objective has been met.
* The transaction pool is constantly monitored throughout testing and is crucial to the system as it acts as an intermediary or buffer of transactions before block creation.

Node Class – Validate Transaction

* This objective has been met.
* The validation of transactions is crucial to the performance of my system and is seen to prevent users from making fraudulent transactions or overspending transactions.

Node Class – Transaction Pool Functions

Merkle Tree Data Structure, Tree Generation

Merkle Root Generation

Merkle Proof Generation

Block Structure – Merkle Root Calculation Function

* These 4 objectives have been met.
* The Merkle tree and root is consistently seen to be created as a Merkle proof is created to check if a given transaction is in the chain or not through the node/user menu actions, allowing the users to check if a transaction has gone through yet or not.

Block Structure – Block Header

Block Structure – Block Header Hashing Function

Block Structure – Mining Function

Child Class (Inherited from Node Class) Miner Node Class Functions

Network Communication – Block Template Generation

* These 5 objectives have been met.
* Together these features allow for the making of a block and the data it is supposed to hold while maintaining the blockchain’s data structure of being essentially a linked list of hash pointers, and then the mining of a block which can only happen given the block is in the correct format and exists. The test shows the mining of blocks as transactions are created and make up the set of transactions used for a block.

Network Communication – Difficulty Target Adjusting Algorithm

* This objective has been met.
* The difficulty target is adjusted to increase the time it took to mine the second block, this allows for the adjusting of the time it takes to mine a block so that it stays consistent even with fluctuations of amount of active miners.

Node Class – Block Validation (Transaction Verifications & Valid Hash Check)

* This objective has been met.
* A block is validated after creation for safety, individually checking each transaction in the block and verifying it, and seeing that the hash meets the difficulty target of starting with a certain number of 0s, which we see get confirmed in debug mode after block creations.

Mining Block Reward + Fee Allocations in Transaction History

* This objective has not been met.
* This objective describes the reward of currency miners get for mining a block and adding it to the chain, which will take form as transactions in the block itself, which get added to the transaction history of the wallet of the miner node, the impact of this objective not being met is that whatever starting currency is issued into the blockchain to begin with is all there will be in circulation.

Blockchain Database (Saving Blockchain History)

* This objective has not been met.
* The implications of this are the fact that a node will have to always be running at one time since the program is kept in main memory (all data is lost when the machine loses power).

Node Client Interface – Switching Between Users

Node Client Interface – Node Privileges (Viewing)

User Interface – All Functionality

* These 3 objectives have been met.
* This is consistently seen through the program, the interface is required for end-users to perform any actions or log in and out of different accounts, and this is done on many occasions.

**User Feedback**

The end-user’s (my brother Sam) response to how well the system meets their needs/the requirements. ‘the system allows for private and public keys not to be maintained by one governing body as users’ keys are generated through RSA meaning no one can make fraudulent transactions due to how asymmetric encryption prevents this, confirming everyone’s identities. This contributes well to what I want, a decentralised system for currency exchange, however there is still minor centralisation as currently only one node can hold a copy of the blockchain while all other users interact with the node through an interface with this system, mining is implemented well and blocks can be made such that they are mined every 10 or so minutes or longer or shorter which is nice for the adaptivity depending on how many active miners there are, this can be used well to prevent malicious attacks like 51% attacks, which my system needs especially early on because 51% attacks are more prevalent when there are less total ‘good’ miners.’

I have no issues with this response as it is fair, they would like the system to be improved in the sense that it is not completely decentralised, only decentralised enough so that it meets their requirements and needs. They seem to be happy with the features I have implemented into the system such as the difficulty target adjusting which helps fight against some malicious attacks, which is good and it makes sense as these are crucial from a developer’s point of view – someone wanting to start a cryptocurrency.

**Future Improvements**

Three ways my program could be improved to meet the end-user’s ideal program would be by adding protection to the private keys that blockchain clients use, a representation for public and private keys that are not as lengthy as the raw public and private keys for end-users’ ease of use, and multi-node broadcasting, as in my implementation only one node has a copy of the blockchain even though multiple nodes can exist at once, there is no broadcasting between blocks.

For the first point of adding protection to the private keys that the blockchain client hold due to users communicating to some node through the client (if the users are not running their own node), this would be beneficial because there is a risk to the users due to not storing them securely. To get around this and implement security for the private keys, the private keys can be encrypted using a cipher and decrypted for use by the blockchain client, or the python script can be kept away from the users and users only get access to the areas that are running. Outside of ways to implement security into the code manually, an external key management service could be used, like AWS Key Management Service.

For the second point of representing public and private keys as different strings while still representing the values of the public and private keys, this would be beneficial as it would provide ease of use for the end-users, in my testing I had to copy and paste these very long keys even just for the public key, a unique username representing the public key would be easier. This could be done through a dictionary, where the program checks to make sure the username the user wants to hold is unique to the existing ones, and if it is, the program will add it to a dictionary where the key is the username and the corresponding value is the actual public key of the user, this allows the public key to be accessed easily while allowing the user to log in with a simpler username. For private keys it might pose a risk to use this method as the private keys would all be stored together, so a better way may be to encrypt the private key with a similar method used on the first point, adding an extra layer of protection to the resulting private key from RSA encryption, it can just be encrypted to become shorter in length.

For the third point of using multi-node broadcasting so that multiple nodes can run their own copies of the blockchain (which is a decent portion of the point of blockchain), this would be beneficial as it would allow users to run their own nodes instead of making transactions through the client which will make the system fully decentralised, which is what the end-user wants. This could be implemented by adding functions which broadcast received transactions from client interfaces and self-mined blocks to all the other nodes on the network, allowing them to validate the transaction or block and add it to their copy of the blockchain (transaction pool or chain respectively), maintaining equality between all the different nodes without one central node maintaining it, making the system fully decentralised.