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.

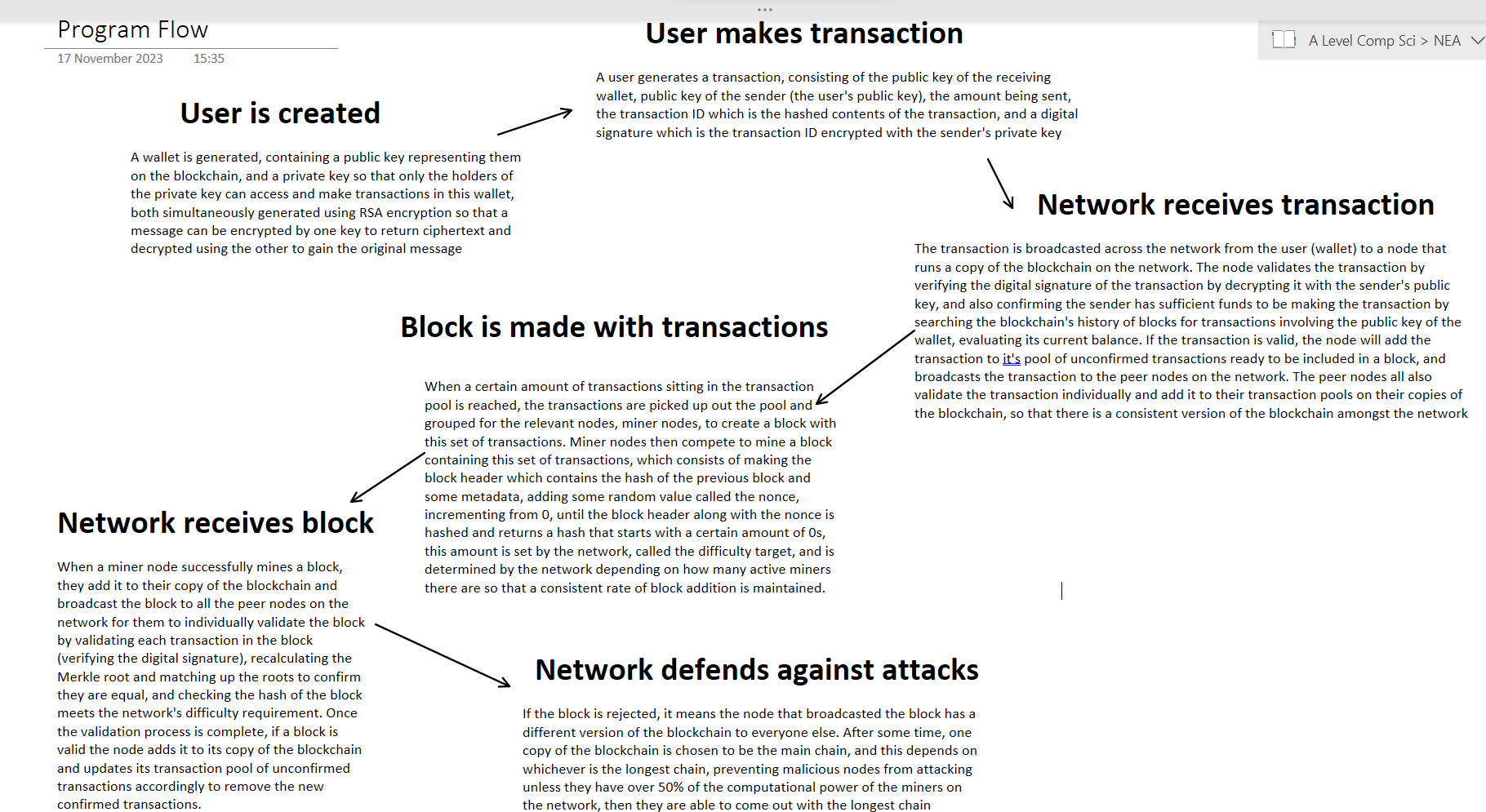
**Objectives**

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

**Modelling**

Flow of the Program

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

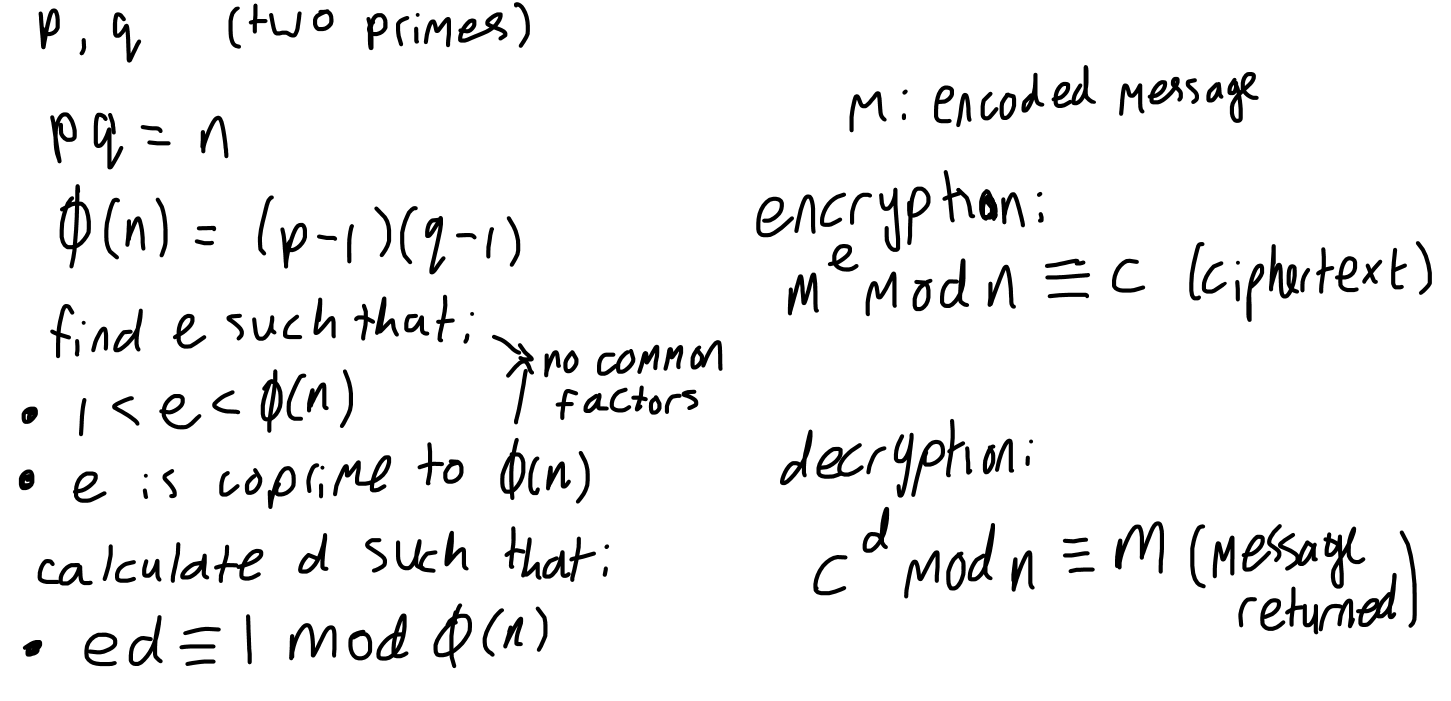


Back to the RSA encryption algorithm, it involves creating a public private key pair that are mathematically linked in such a way that if you encrypt a message with one key, you can decrypt it to gain back the original message from ciphertext with the other key. Typically in RSA encryption, the public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

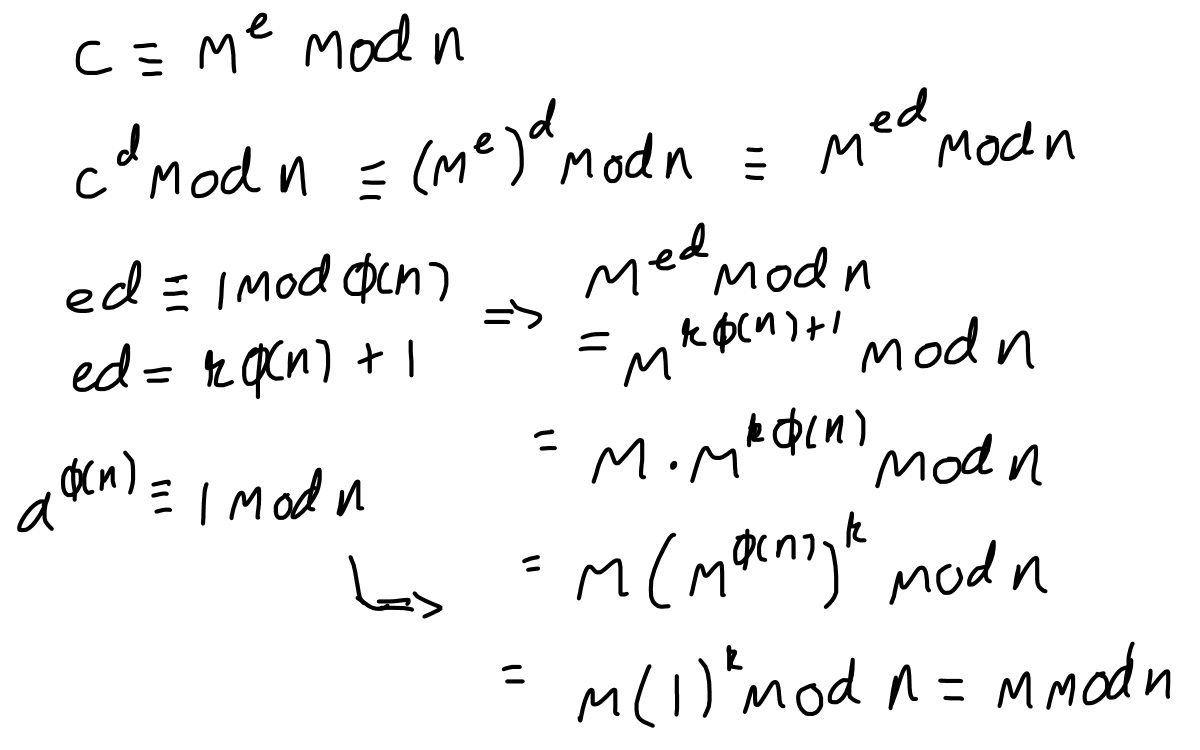
Now going into how the RSA encryption algorithm can generate these two mathematically linked keys, it uses a lot of modular arithmetic and prime numbers. The main underlying concept behind it is the fact that currently, we have a lot of methods to find prime numbers, but not efficient ways to break up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

Mathematics Behind RSA Encryption

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



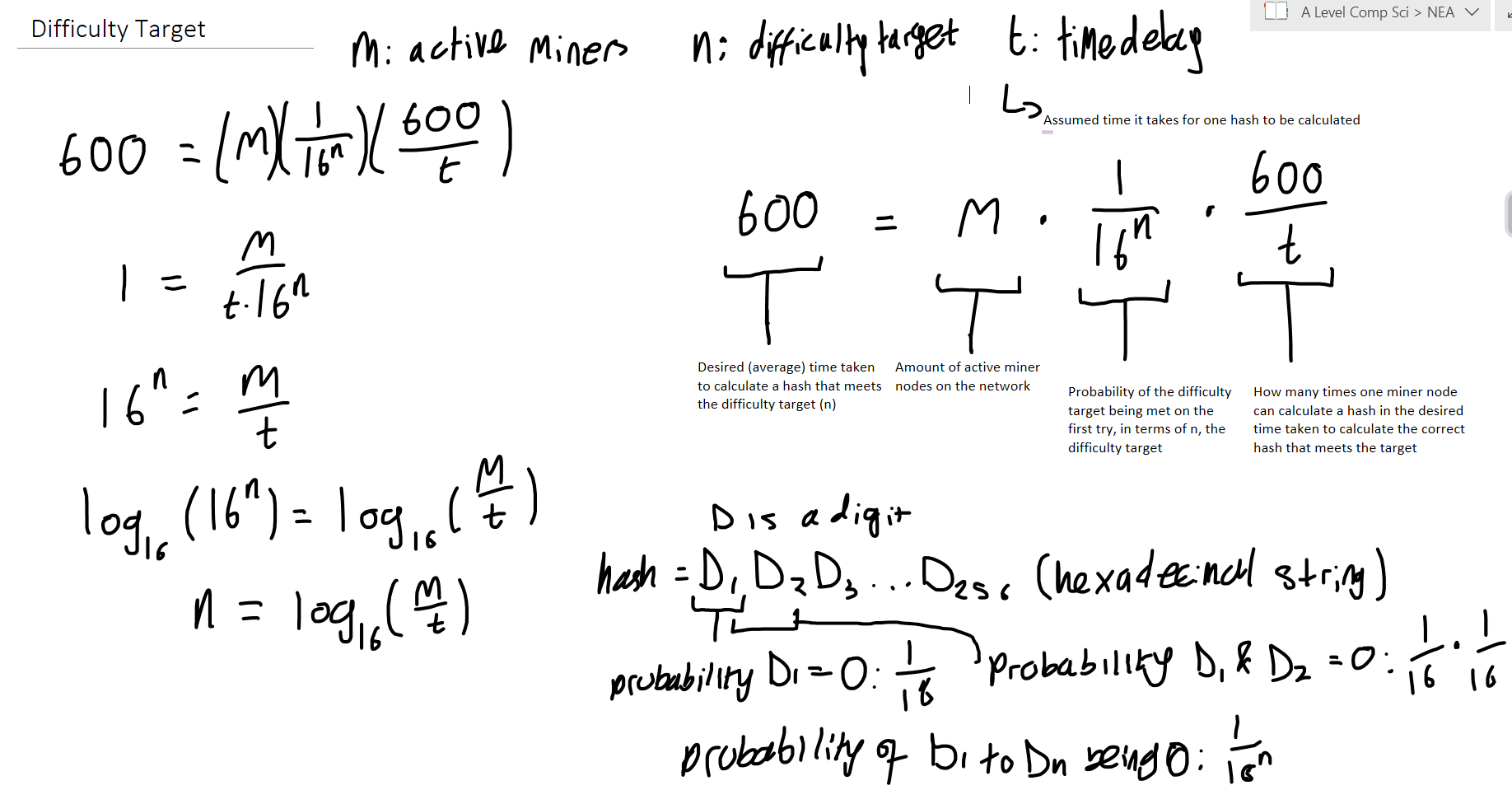
The reason this works is because of Fermat’s Little Theorem, which states that if p is a prime number, and a is an integer not divisible by p, then ap-1 1 mod p, which can be extended to Euler’s Totient Theorem, to work for composite numbers (like n being a composite of p and q) by considering the totient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



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

Difficulty Target Calculation

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



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

Blockchain Database ERD – Not Finished

Data Saving and Loading – Not Finished

Prototype

The prototype has a few key differences to my main program. The prototype does not store the blockchain’s data on a database, meaning it is not saved. The prototype does not have a network system, only the functionality of a blockchain on one client, meaning nothing is broadcasted between different nodes, and essentially only one node can run at a time. There is no wallet address and the public key is directly used to represent users on the blockchain. Terminal responses to navigating actions on the blockchain instead of an external interface. There are no serialisation and deserialization methods compacting and reconstructing objects for broadcasting efficiency since there’s no broadcasting across a network in this prototype. The private key is stored in the class in this prototype whereas in the final product it will be securely stored or not stored at all anywhere. Validating blocks only consists of verifying all the digital signatures of the transactions and confirming that the difficulty target has been met, which is constant because there’s no network class in this prototype so no class to adjust the difficulty target based on active miners to keep a consistent rate of blocks coming in, the difficulty target is set to a low number for debugging purposes (makes mining times a lot shorter). The prototype does not include much exception handling, so the program will terminate because of errors, whereas in the final version, errors wont cause the program to terminate as it might be apart of the program like defending against malicious attacks should not lead to the program closing.

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.