Hybrid BlockChain

Kappa Noeler Kyamazinga

Abstract

Can the block chain be scaled to support thousands of transactions per second, like the world's major payment networks?

Ethereum is a public block chain platform that lets anyone build and use decentralized applications. In its current state, Ethereum doesn't scale very well because every transaction needs to be processed by each and every node in the network.

Rather than having each and every node computing each and every smart contract, I propose a private chain with off-blockchain transactions acting as a side chain where any number of transfers can be processed locally.

I explore the possible interoperability between the two; public and private distributed ledgers as a hybrid blockchain. An asset is moved from a private test chain to a public blockchain or from a public to private one. In this use case, the proof of authority block header is checked and accepted as a valid header because the signature of the signers has been verified. Once this is done, a token is issued and this transaction is then tracked on the simulated public chain.

This mechanism allows only for a subset of nodes to verify each transaction and as long as there enough nodes verifying each transaction, the system is still highly secure but the nodes are not too many so that the system can still process as many transactions in parallel.

The advantages of this design include high throughput, low latency and anonymity.

Contents

Chapter 1

Introduction

A block chain is defined as a distributed database/ledger where every network node executes and records the same transactions grouped into blocks. Only one block can be added at a time, and every block contains a nonce that verifies that it follows in sequence from the previous block.

This technology was originally used as the underlying peer-to-peer distributed timestamp server for Bitcoin. It's based on cryptographic proof allowing any two willing parties to transact directly with each other without the need for a trusted third party. A transaction is a chain of digital signatures with each owner making transfers to the next owner by digitally signing a hash of the previous transaction and the public key of the next owner. A hash of a block of transactions to be timestamped is published in the peer to peer network. This proves that the data must have existed at that time to get into the hash. Each timestamp will include the previous timestamp in it's hash, forming a chain that reinforces the blocks before it. Implementing this system as a peer to peer network requires a Proof of Work consensus. Proof of Work is a consensus system that relies on computational proof for the chronological order of transactions. CPU/GPU time and electricity are used when work is done with CPU/GPU cycles to scan for a value that when hashed, like with SHA-256, gives a value that begins with a number of zero bits. The block cannot be changed without having to redo this. Miners race to solve this cryptographic problem and are rewarded with BitCoin to validate transactions and create new blocks.[1]

In Ethereum, each and every node of the network runs the Ethereum Virtual Machine and executes the same Contract Accounts. The virtual machine on every

node maintains consensus as each and every node connected to the network executes the same instructions. This is used to implement a trustless smart contract platform and allows users to create their own decentralised applications. The sender of a transaction pays with Ether for the program activated as well as computation and memory storage. The transaction fees are collected by the nodes that validate the network. These nodes receive, propagate, verify and execute transactions by grouping them into blocks. These miners race to solve a proof of Work which is a memory-hard as well as CPU computational problem.[9]

Proof of stake is an alternate consensus system that is based on proof of ownership of the currency i.e someone holding 5 percent of Bitcoin can mine 5 percent of the blocks.[2]

Proof of Authority is an alternate consensus system where instead of miners racing to find a solution to a difficult problem, authorized signers can at any time and own discretion create new blocks.[1.3]

Transitions between consensus systems or changes on the blockchain can be achieved by hard forks or soft forks. A hard fork is permanent, when non-upgraded nodes can't validate blocks created by upgraded nodes on a chain while a soft fork is temporary, non-upgraded nodes don't follow the new consensus rules.

There are three types of blockchains, Public, a "fully public, uncontrolled network and state machines secured by crypto economics", with examples described earlier. Private blockchains however have "write permissions which are centralized to one organization, read permissions may be public or restricted to an arbitrary extent". The third type, Consortium Block Chains have a "consensus process controlled by a pre-selected set of nodes"[3]

While blockchain technology allows for transparency, some assets need only to be private and localized. A hybrid blockchain would be a mix of both the public and private chains. This could be used by groups of organisation/firms, sharing assets with multiple transactions or within an internal organisation with intra-transactions without need for the public to have access to them.

Chapter 2

Literature Review

2.1 Scaling Ethereum

Scaling addresses the ability of the system to maintain its quality of service as the overall system load increases. Ethereum aims to allow the processing of a high volume of transactions, maybe 10,000+ transactions per second without requiring a full node running Ethereum to become a super computer or to store a terabyte of data of the blockchain.

The two research based proposals for scaling Ethereum include implementing sharding and lightning- like networks, a payment channel network mechanism. Ethereum has a long term goal of changing the consensus from Proof of Work, whose design wastes resources, to Proof of Stake using the Casper protocol. Casper is a security-deposit based economic consensus protocol. Nodes that are validators have to place a security deposit, also known as bonding to produce blocks thus providing the consensus. Sharding and lightning-like networks use the existing Proof of Work consensus in place.

2.1.1 Definitions

2.1.1. Definiton. *State is information about the system's condition. In BitCoin, this data can be represented by the unspent transaction outputs set. In Ethereum, state data is account balances, nonces, contract code and storage and in NameCoin, this data is a key/value pair representing a domain name system entry.*

2.1.2. Definiton. *History is an ordered list of all transactions that have taken place from the genesis block.*

2.1.3. Definiton. *A transaction is a transfer of value. It's broadcast on the peer to peer network to be accepted and recorded in the block-chain. In Ethereum, a digitally signed piece of data that stores a message of a set of bytes and ether value is sent from an externally owned account to another account on the blockchain.*

2.1.4. Definiton. *State transition function takes an existing state, applies a transaction and outputs a new state. This function can be adding and subtracting balances from accounts specified by the transaction, verifying digital signatures and then running contract code.*

2.1.5. Definiton. *Merkle tree is a cryptographic hash tree structure that stores a very large amount of data and verifying each piece of data only takes a short time. In Ethereum, the transaction set of each block, as well as the state, is kept in a Merkle tree, the roots of the trees are committed in a block.*

2.1.6. Definiton. *Receipt is an object stored in a Merkle tree that represents a transaction outcome that is not stored in the state, and is committed in a block so that its existence can later be efficiently proven even to a node that does not have all of the data. Logs in Ethereum are receipts; in sharded models, receipts are used to facilitate asynchronous cross-shard communication.*

2.1.7. Definiton. *Light client is a way of interacting with a blockchain that requires the use of very little computational resources. The client keeps track of only the block headers of the chain and can acquire any needed information about transactions, state or receipts and can verify Merkle proofs of the relevant data as-needed. The amount of resources can be O(1) or Olog(c). O(1) is when the system has approximately the same properties irrespective of how big it gets.*

2.1.8. Definiton. *State root is the root hash of the Merkle Patricia tree storing the state of the system. Please see Figure 2.1*

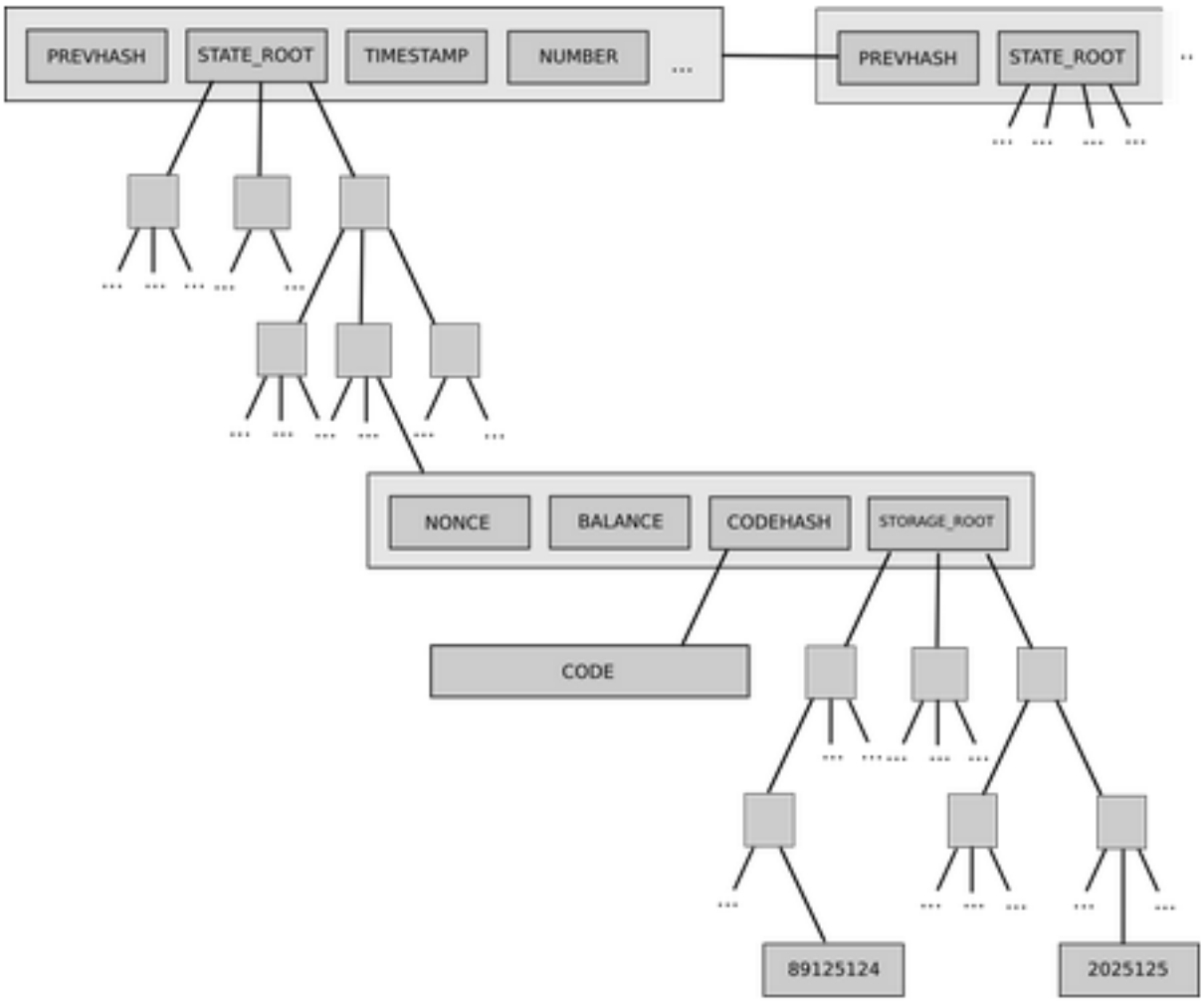

Figure 2.1: EthereumStateTree

[6]

2.1.2 Sharding

The design considerations for scaling with sharding aim to ensure that the workload of state storage, transaction processing, downloading and re-broadcasting is spread out across nodes. ««problem»>

Scalability is being able to process O(n) transactions greater than O(c) computational resources. i.e O(n) > O(c) with each participant having O(c) resources and security against attackers up to O(n). O(n) is a linear scale, every time you double n, the amount of work is doubled so that for every element, you are doing a constant number of operations.

The validation effort required per full node simply grows in a linear scale O(n), but with each node participating on the network computing every smart contract transaction, the combined validation effort of all nodes grows by O(n^2) with decentralization being held constant.O(n^2)is a quadratic scale, Every time n doubles, the operation takes four times as long.

The Bitcoin blockchain is a list of transactions. Ethereum blockchain shows the state of accounts.There are two types of accounts, Externally Owned Accounts (EOAs) which are controlled by a user's private keys and contract accounts controlled by their contract code that performs instructions based on the Externally Owned Accounts.

Sharding is the splitting of the space of possible accounts like contracts into sub spaces. [4] This could be based on the numbering of their addresses.

If the value of the cryptocurrency is O(k log(k)) with k users. A good model to use would be n = O(klog(k)), basing everything off of n and c. O(n log n) operations run in log linear time, O(log n) is performed on each item in your input. c refers to computational resources like computation, bandwidth and storage, n refers to the size of the ecosystem i.e the transaction load, state size and cryptocurrency of the market.

In this solution, nodes might be arranged according to addresses and hold a subset of the state and subsequently of the blockchain thus sharing the load instead of everyone doing the same work. Please see Figure 2.2. Each shard has it's own history and the effects of that shard are limited to it.

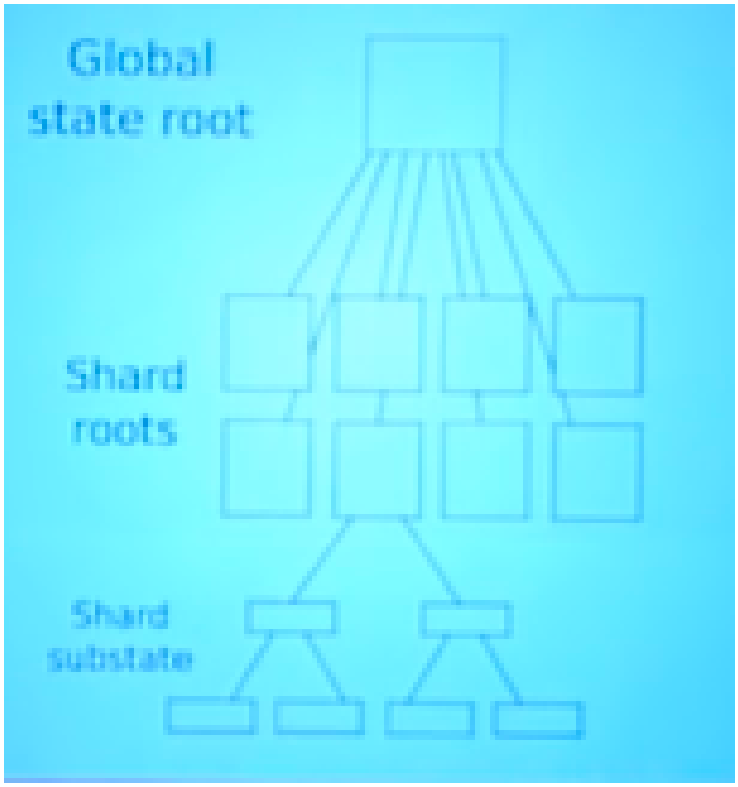

Figure 2.2: Sharding

[1.1]

Cross-shard communication can occur when actions on one shard triggers events on another. The nodes that accept transactions on shard k, either choose which shard or are randomly assigned one creating collations. Collation refers to a set of rules that determine how data is sorted and compared. This collation has a header, short message and the previous state root. A block then contains the collation headers for each shard.

Nodes in the system would include a super full node that processes all transactions in all collations, maintaining the full state for all shards.

A top level node that processes all top level blocks only accepts that a collation is valid, if two thirds of the collators in that shard are in agreement that it's valid. A light node would then only download and verify the block headers of the top level blocks. It doesn't process any collation headers or transactions.

An advantage here is that more transactions per second can be done without changing the trust model, we can count the total work from the genesis block instead of looking at the headers on the top level chain, we look at the shards as well.

Another advantage is that contract executions can be executed within the shards instead of network wide so orders of any magnitude of transactions per second can be received as every node is not validating each and every smart contract. Each shard would store the balance and process the transactions associated with one particular asset, maybe even have cross-shard communication.

The drawback however is an appropriate method of assigning miners to shards so that miners from one shard don't mine from another to produce invalid blocks. There is also need to test the mining power to ensure an honest majority.

Another drawback is there is no immediate channel for cross shard communication and ensuring that a take over attack of a single shard with one attacker controlling majority of the collators is contained.

There should also be a fraud detection mechanism once an invalid collation is done and data goes missing as a result of fraud.

»>Still incomplete :) In noSQL databases, shards are single instances of a database, often run on a commodity server which houses a subset of the system's total data set. Scaling these databases can be done vertically, by adding resources to each individual component within the system for example switching to a bigger machine or horizontally, by adding individual components to the system e.g load balancing or data partitioning.

Performing algorithmic look ups uses a rule to retrieve the shard location of the data based on some portion of the address itself assigned to this sub space or on a whole on the collation headers in a block as a result of sharding.

In key- based partitioning, you use the data itself to do the partitioning; a one-way hashing algorithm to map the data of the collation headers.

A more dynamic improvement to the sharding proposal would be to improve shard partitioning. Domain Partitioning as a method of partitioning the database has a master index lookup table with an index shard and a domain shard. A shard is a sub space based on the numeric addresses of the contracts. The index shard has the dataset's main partitioned indexes and the existing domain shards attached to these indexes.

A scheme using these shard URL addresses to reference the collation headers can allow you to use multiple shard-storage types. Some of the data types in a given state include balances, nonces, code or storage. Using this scheme enables support for multiple sharding schemes. Each entity is associated with a name and an address. Then, whenever the system needs to find the collation header for a given entity, the name is composed into a URL using the associated address.

One advantage would be caching as you get additional meta data associated with every query in the application, the main draw back would be additional overhead in the top level nodes or index shards parsing all the data.

2.2 Payment Channels

State channels are any kind of state-altering operation that normally would be performed on a block chain but is done off the block chain. A payment channel is a state channel for payments with an escrow arrangement between two parties.

A Unidirectional Payment Channel is a payment in only one direction and works a lot like a blind auction. Once payment is complete, the channel is closed.

If Alice wants to make a lot of payments to Bob, without paying gas fees for every transaction. She sets up a contract and deposits some ether. For each payment, she sends Bob a signed message, saying "I agree to give 10 ether to Bob." At any time, Bob can post one of Alice's message to the contract, which will check the signature and send Bob the money.

Bob can only do this once. After he does it, the contract remembers it's done, and refunds the remaining money to Alice. So Alice can send Bob a series of messages, each with a higher payment. If she's already sent Bob a message that pays 10 ether, she can pay him again by sending a message that pays 11 ether.

Alice can add an expiration date, after which she can retrieve any money she deposited that's not already paid out. Until then, her funds are locked. Before the deadline, Bob can work offline, he just has to check the balance and to post the message with the highest value before it expires.

Bidirectional Payment Channels allows payments on a channel in a two way direction. Alice wants to make some payments to Bob and Bob wants to make some payments to Alice.

These two parties create an entry on the block chain by sending funds to a multi-signature address. Any spending of these funds requires both of them to sign off of a commitment transaction. Once Alice creates a commitment transaction, she does not broadcast it to the block chain, she sends it directly to Bob and Bob does the same. This commitment transaction then refunds their ledger entry to their individual allocations and sends some funds to another multi signature address opening up a channel. They can update their individual allocations on this channel by creating several transactions, spending from the current ledger entry output. The most recent version of this entry is considered valid and the channel can be closed out at any time by either party when they broadcast the most recent version to the block chain. Please see Figure 2.3

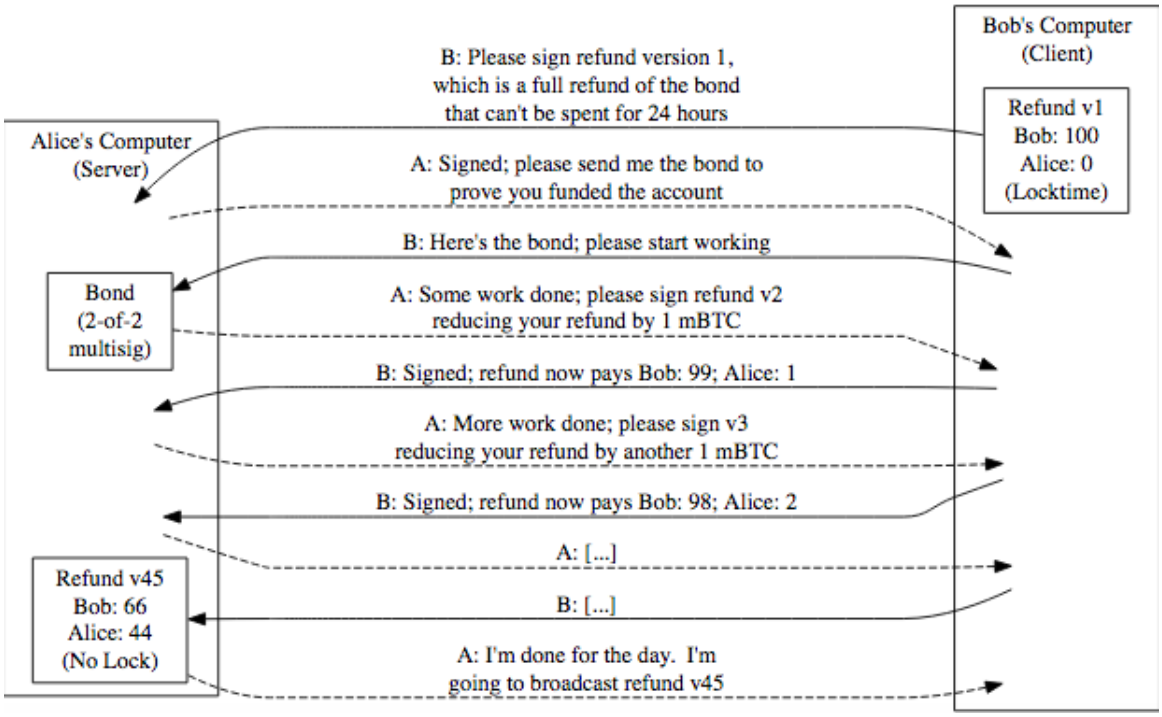

Figure 2.3: MicroPaymentChannel

[5]

Micropayment or Microtransaction channels create a relationship between two parties to continually update balances. It allows these parties to be able to update the channel's state off-chain and settle to the block chain at any time, and take all the funds in the channel if the counter party attempts to write an old channel state to the block chain.

The implementation of a smart contract in Ethereum for a payment channel accepts and verifies a signed transaction command and a nonce. Please see Appendix A for the source code. Using an incrementing nonce prevents replay attacks in transactions. A replay attack is a valid data transmission that is maliciously or fraudulently repeated or delayed. In block chains, a replay attack is taking a transaction on one block chain, and repeating it on another block chain.

In the payment channel, participants send ether to the smart contract after creating and exchanging signed spending transaction commands. Alice signs a transaction command sending funds immediately to Bob and locking funds for herself to be released after some amount of time, and vice versa. The smart contract contains the logic that any funds time locked for the counterparty can be claimed by producing a

transaction command signed by the counterparty with a higher nonce than the one which the counterparty broadcast.

The channels also allow a transaction state to be broadcast at a later time. The contracts are enforced by creating a responsibility for one party to broadcast transactions before or after certain dates.

2.2.1 Lightning Network

2.2.1. Definiton. *A Hashed TimeLock Contract is a class of payments that uses hash locks and time locks where the receipt of a payment either receives the payment before a deadline by generating cryptographic proof of payment or loses the ability to claim the payment after the deadline, returning it to the payer.*

2.2.2. Definiton. *A Hashlock is a type of impediment that restricts the spending of an output until a specified piece of data is publicly revealed.*

2.2.3. Definiton. *A Timelock is a type of impediment that restricts the spending of some cryptocurrency until a specified future time or block height.*

The Lighting Network protocol applies to Bitcoin and Bitcoin-like blockchains such as Litecoin, where it's currently implemented.

Payment channels use hashlocks and timelocks. This allows the creation of complex contracts using Bitcoin multisignature transactions and scripts by creating time frames where certain states can be broadcast and later invalidated. It is also possible to create a network of transaction paths. By chaining together multiple micropayment channels, paths can be routed using a Border Gateway Protocol-like system, and the sender may designate a particular path to the recipient. The output scripts are checked by a hash, which is generated by the recipient. By disclosing the input to that hash, the recipient's counter party will be able to pull funds along the route

An initial channel funding transaction is created whereby one or both channel counter parties blocks fund the inputs of this transaction. Both parties create the inputs and outputs for this transaction but do not sign the transaction.

It is possible to find a path across the network similar to routing packets on the internet by creating a network of these payment channels. The nodes along the path are not trusted so the payment is enforced with atomicity, where either the

entire payment succeeds or fails via decremented time-locks. This scales transaction capacity by a constant factor but cannot scale state storage.

2.2.2 Raiden Network

The Raiden Network uses a network of peer to peer payment channels for asset or value transfers and deposits with Hashed Time Lock Contracts in Ethereum. Users can send the payment through a network of chain channels until it reaches the recipient.

The design switches from a model where all transactions go to the block chain to one where users can privately exchange messages which sign the transfer of value.

The related transactions take place almost instantly on these off-chain micro payment channels and only the final settlement or dispute resolution is processed by the block chain.

It uses two-way micro payment channels so that users can update the transaction amount in both directions but also supports uni-directional payments.

The Raiden node is built on top of Ethereum as a smart contract and a Raiden light node runs alongside an Ethereum node for delegation of transfers of deposits and for communication with other Raiden nodes for transfers.

2.3 BTC Relay

BTC Relay allows for interoperability between Ethereum and Bitcoin. It is an Ethereum contract that implements Bitcoin Simplified Payment Verification (SPV).It allows Ethereum contracts to securely verify Bitcoin transactions without any intermediaries so users can pay with Bitcoin to use Ethereum Decentralised Applications also known as DAPP's. This Ethereum contract stores Bitcoin block headers and uses these headers to build a mini-version of the Bitcoin blockchain, a method already used by Bitcoin Simplified Payment Verification (SPV) light wallets.

BTC Relay keeps track of the Bitcoin headers and proof of work chain in the contract's storage. The headers are stored in a skip list data structure that supports quickly finding ancestors of a given chain.

Simplified Payment Verification security relies on at least one honest user submitting new Bitcoin block headers in a timely fashion.

BTC Relay relies on an incentive based mechanism where the submitter can choose a price, and consumers that call to verifyTx must pay it. Relayers are those who submit block headers to BTC Relay. When any transaction is verified in the block, or the header is retrieved, relayers are rewarded a fee. This allows BTC Relay to be autonomous and up-to-date with the Bitcoin blockchain. When an application processes a Bitcoin payment, it uses a header to verify that the payment is legitimate. The cycle of relayers submitting headers, then applications processing Bitcoin payments and rewarding a relayer with a micro-fee, allow the system to be self-sustaining.

The main functionality includes verification of a Bitcoin transaction, optionally relay the Bitcoin transaction to any Ethereum contract, storage of Bitcoin block headers and the inspection of the latest Bitcoin block header stored in the contract and BTC Relay contract address. »>Application Binary Interface

Please see Appendix A. [7]

Chapter 3

Research Methodology

3.1 Side Chains

3.1.1 Pegged side chains

Pegged side chains have the following desired properties;- - They should be independent In case of failure, transfers should be terminated and a bug on a side chain should not affect another side chain - Assets which are moved between block chains should only be moved back by the current owner. - No dishonest party should be able to prevent this asset from being moved

Assets are transferred with proofs of possession. A transaction on the first block chain locks the assets, a transaction on the second blockchain has inputs that show the lock was done correctly and are tagged with an asset type like the genesis hash of the parent blockchain. Transferring the coins between side chains is done using with block headers acting as a simplified payment verification proof.

3.1.2 Atomic Swaps

The problem of atomic cross-chain trading is one where (at least) two parties, Alice and Bob, own coins in separate cryptocurrencies, and want to exchange them without having to trust a third party (centralized exchange).

A non-atomic trivial solution would have Alice send her Bitcoins to Bob, and then have Bob send another cryptocurrency to Alice - but Bob has the option of going back on his end of the bargain and simply not following through with the

protocol, ending up with both Bitcoins and the altcoin

The solution is using revealing secrets of contract with Contracts and n LockTime Contracts.

To implement this, this protocol can be used;-

Party 'A' generates some random data, x (the secret). Party 'A' generates Tx1 (the payment) containing an output with the chain-trade script in it. It allows coin release either by signing with the two keys (key 'A' and key 'B') or with (secret 'x', key 'B'). This transaction is not broadcast. The chain releases a script that contains hashes, not the actual secrets themselves. Party 'A' generates Tx2 (the contract), which spends Tx1 and has an output going back to key 'A'. It has a lock time in the future and the input has a sequence number of zero, so it can be replaced. 'A' signs Tx2 and sends it to 'B', who also signs it and sends it back. 'A' broadcasts Tx1 and Tx2. Party 'B' can now see the coins but cannot spend them because it does not have an output going to him, and the tx is not finalized anyway. 'B' performs the same scheme in reverse on the alternative chain. The lock time for 'B' should be much larger than the lock time for 'A'. Both sides of the trade are now pending but incomplete. Since 'A' knows the secret, 'A' can claim his coins immediately. However, 'A', in the process of claiming his coin, reveals the secret 'x' to 'B', who then uses it to finish the other side of the trade with ('x', key 'B').

nLockTime This is a parameter of a transaction, that, if any input indicates by having nSequence not equal to UINT-MAX, mandates a minimal time, specified in either unix time or block height, before which the transaction cannot be accepted into a block. If all inputs in a transaction have nSequence equal to UINT-MAX, then nLockTime is ignored

3.2 Two way peg

A simplified payment verification proof has a) a list of block headers demonstrating proof of work b) a cryptographic proof that an output was created in one of the blocks in the list

By requiring a block header to commit to the block chain's unspent output set, anyone in possession of a Simplified Payment Verification proof can determine the state of the chain without any need to replay every block.

3.2.1 Symmetric Two way peg

To transfer parent chain coins into side chain coins, the parent chain coins are sent to a special output on the parent chain that can only be unlocked by a Simplified Payment Verification proof of possession on the sidechain. To synchronize the two chains, we define two waiting periods.

- The confirmation period of a transfer between side chains is a duration for which a coin must be locked on the parent chain before it can be transferred to the side chain. This allows work to be created. User then creates a transaction on the side chain referencing the output on the parent chain providing a Simplified Payment Verification. Confirmation period is a per-sidechain security parameter which trades cross-chain transfer speed for security

-User must then wait for the contest period, duration in which neither coin may be spent on the side chain. Contest prevents double-spending and transfers the previously-locked coins during a re-organisation. If any new proof is published containing a chain with more aggregate work which does not include the block where the lock output was created, the conversion is invalidated. This Contest period is usually long and can be replaced by atomic swaps.

- While locked on the parent chain, the coin can be freely transferred within the side chain without any interaction with the parent chain but can only be transferred back to the same chain that it came from and also retains the identity of the parent chain.

- A reverse transaction is similar, send the coins on the sidechain to an Simplified Payment Verification-locked output, produces a sufficient Simplified Payment Verification proof. Please see Figure 3.1

[8]

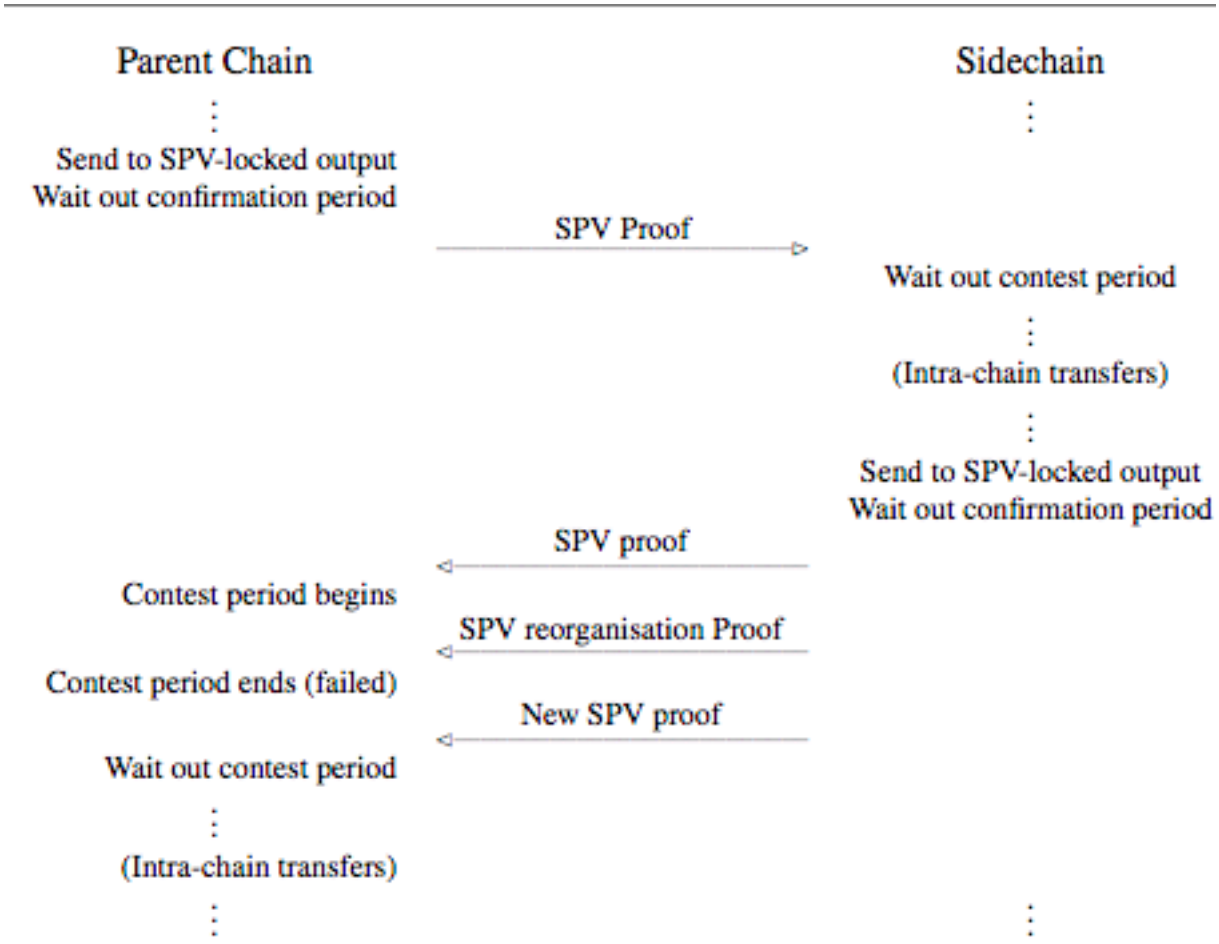

Figure 3.1: sidechains

Since pegged sidechains carry assets from many clients, assumptions of the security of these chains cannot be made so the assets are not interchangeable except on trading so side chains treat assets from separate parent chains as different asset types. Parent and Side Chains do Simplified Payment Verification validation on each other. Such proofs would need to be made compact enough to fit in a Bitcoin transaction or a smart contract. However, this is just a soft-forking change, without effect on transactions which do not use the new features

.

3.2.2 Asymmetric Two way peg

Users of the side chain are full validators of the parent chain and transfers from parent chain to side chain don't require Simplified Payment Verification proofs since all validators are aware of the state of the parent chain but the parent chain is unaware of the sidechain so Simplified Payment Verifications are required to transfer back. There is increased security but side chain has to track the parent chain and reorganizations on parent chain mean re-organisations on sidechains

Disadvantages On the network level, you have multiple independent unsynched blockchains supporting transfers between each other. They must support transcription scripts which can be invalidated by a later re-organisation proof and software that detects misbehaviour.

Enabling the blockchain to handle advanced features isn't enough, user interfaces for wallets must be reconsidered to support multiple chains

Fraudulent transfers where reorganizations of some depth might allow an attacker to completely transfer coins between side chains before causing a reorganization longer than the contest period on the sending chain. This can prevented by increasing contest period for transfers

3.3 Proof of Authority

In the Clique proof of authority protocol proposal, authorized signers can at any time create new blocks instead of mining in the proof of work consesus algorithm.

A Proof of Authority scheme is based on the idea that blocks may only be minted by trusted signers. As such, every block or header that a client sees can be matched against the list of trusted signers.

The list of authorized signers can change in time so the protocol of maintaining the list of authorized signers must be fully contained in the block headers.

Some of the immediate solutions are to store it in an Ethereum contract but fast, light and warp syncs don't have access to the state during syncing. There is also a solution to change the structure of the block headers to drop the idea of Proof Of Work, and introduce new fields to cater for voting mechanisms but changing a core data structure in multiple implementations has many drawbacks for development, maintenance and security so the protocol of maintaining the list of authorized signers must fit fully into the current data models.

We can't use the Ethereum Virtual Machine for voting, rather have to resort to headers and we can't change header fields so there is need to resort to the currently available ones. The header fields are re-purposed for signing and voting.

One of the fields that are currently used solely as non critical metadata is the 32 byte extra-data section in block headers. The protocol would extend this field to 65 bytes with the purpose of a secp256k1 miner signature. This would allow anyone obtaining a block to verify it against a list of authorized signers. This serves a dual purpose of making the miner section in block headers obsolete (since the address can be derived from the signature).

Changing the length of a header field is a non invasive operation as all code (such as RLP encoding, hashing), so clients wouldn't need any custom logic. This is enough to validate a chain, but not enough to update a dynamic list of signers.

The newly obsoleted miner field and the proof of authority obsoleted nonce field are re-purposed to create a voting protocol. During regular blocks, both of these fields would be set to zero. If a signer wishes to enact a change to the list of authorized signers, it would : -Set the miner to the signer it wishes to vote about -Set the nonce to 0 or 0xff...f to vote in favor of adding or kicking out

Any clients syncing the chain can add up the votes during block processing, and maintain a dynamically changing list of authorized signers with popular vote. The initial set of signers can be given as genesis chain parameters.

To avoid having an infinite window to add up votes in, and allow periodic flushing stale proposals, we can reuse the concept of an epoch from ethash, where every epoch

transition flushes all pending votes. These epoch transitions can also act as stateless checkpoints containing the list of current authorized signers within the header extra-data. This permits clients to sync up based only on a checkpoint hash. It also allows the genesis header to fully define the chain, containing the list of initial signers.

Attack vectors

It may happen that a malicious user gets added to the list of signers, or that a signer key/machine is compromised. In such a scenario the protocol needs to be able to defend itself against re-organizations and spamming. The proposed solution is that given a list of N authorized signers, any signer may only mint 1 block out of every K. This ensures that damage is limited, and the remainder of the miners can vote out the malicious user.

Another attack vector is if a signer (or group of signers) attempts to censor out blocks that vote on removing them from the authorization list. To work around this, we restrict the allowed minting frequency of signers to 1 out of N/2. This ensures that malicious signers need to control at least 51 percent of signing accounts, at which case the system is fully compromised.

A final spam attack vector is that of malicious signers injecting new vote proposals inside every block they mint. Since nodes need to add up votes to create the actual list of authorized signers, they need to track all votes through time. Without placing a limit on the vote window, this could grow slowly, yet unbounded. The solution is to place a moving window of W blocks after which votes are considered stale. A sane window might be 1-2 epochs. We'll call this an epoch.

With concurrent blocks, if the number of authorized signers are N, and we allow each signer to mint 1 block out of K, then at any point in time N-K miners are allowed to mint. To avoid these racing for blocks, every signer would add a small random "offset" to the time it releases a new block. This ensures that small forks are rare, but occasionally still happen (as on the main net). If a signer is caught abusing it's authority and causing chaos, it can be voted out

Syncing a blockchain is usually done in two ways, getting the genesis block and getting all the transactions one by one which is very costly in terms of computation. The other is to only download the chain of block headers and verify their validity, after which point an arbitrary recent state may be downloaded from the network and checked against recent headers. Please see Figure 3.2

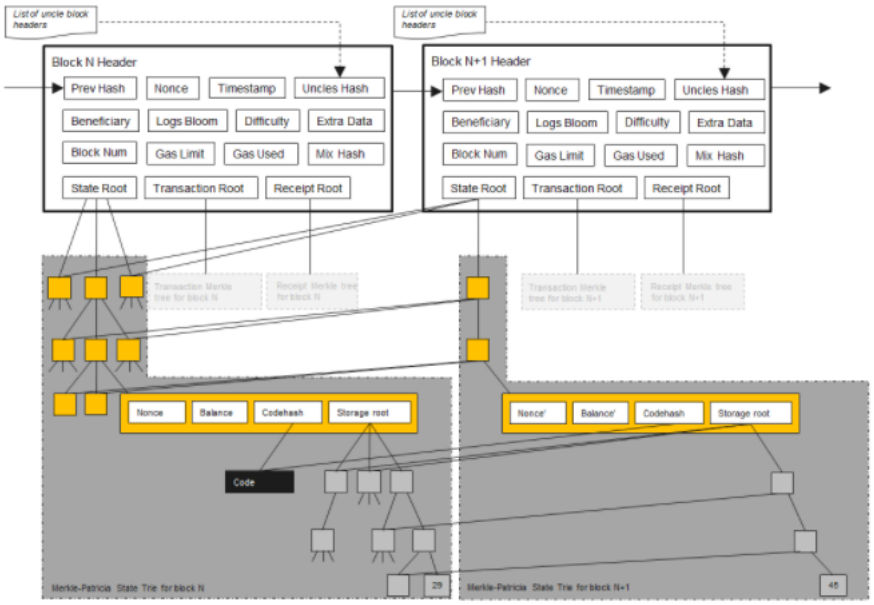

Figure 3.2: architecture

3.3.1 Definitions Ethereum Block Header

3.3.1. Definiton. *The block in Ethereum is the collection of relevant pieces of information (known as the block header), together with information corresponding to the comprised transactions, and a set of other blockheaders, that are known to have a parent equal to the present block's parent's parent (such blocks are known as ommers).*

The block header contains several pieces of information:

3.3.2. Definiton. *parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety.*

3.3.3. Definiton. *ommersHash: The Keccak 256-bit hash of the ommers list portion of this block*

3.3.4. Definiton. *beneficiary: The 160-bit address to which all fees collected from the successful mining of this block are to be transferred*

3.3.5. Definiton. *stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied*

.

3.3.6. Definiton. *transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block*

3.3.7. Definiton. *receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.*

3.3.8. Definiton. *logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list .*

3.3.9. Definiton. *difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp*

3.3.10. Definiton. *number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero .*

3.3.11. Definiton. *gasLimit: A scalar value equal to the current limit of gas expenditure per block .*

3.3.12. Definiton. *gasUsed: A scalar value equal to the total gas used in transactions in this block*

3.3.13. Definiton. *timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception. .*

3.3.14. Definiton. *extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer. .*

3.3.15. Definiton. *mixHash: A 256-bit hash which proves combined with the nonce that a sufficient amount of computation has been carried out on this block*

3.3.16. Definiton. *nonce: A 64-bit hash which proves combined with the mix-hash that a sufficient amount of computation has been carried out on this block*

BlockHeaderDefinitions[1.4]

Chapter 4

Findings / Results / Data Analysis

4.1 Scalable BlockChain

The following competing models are used to evaluate the safety of blockchain designs

Honest majority (or honest supermajority):

We assume that there is some set of validators and up to half (or a third or a quarter) of those validators are controlled by an attacker, and the remaining validators honestly follow the protocol.

Uncoordinated majority:

We assume that all validators are rational in a game-theoretic sense (except the attacker, who is motivated to make the network fail in some way), but no more than some fraction (often between a quarter and a half) are capable of coordinating their actions.

Coordinated choice:

We assume that all validators are controlled by the same actor, or are fully capable of coordinating on the economically optimal choice between themselves. We can talk about the cost to the coalition (or profit to the coalition) of achieving some undesirable outcome.

Bribing attacker model:

We take the uncoordinated majority model, but instead of making the attacker be one of the participants, the attacker sits outside the protocol, and has the ability to bribe any participants to change their behavior. Attackers are modeled as having a budget, which is the maximum that they are willing to pay, and we can talk about

their cost, the amount that they end up paying to disrupt the protocol equilibrium.

4.2 Hybrid BlockChain

Design for a Hybrid BlockChain - A genesis block for a proof of authority test chain with network id 1988 simulating a private block chain. - A genesis block for a proof of work test chain with network id 1987 simulating a public block chain. - Mist wallet with geth - A smart contract that checks and accepts a proof of authority block header where the signature or the signer is verified on the private chain - A smart contract that checks and verifies the transaction on the proof of work test chain. - An online Remix solidity IDE to write solidity contracts - An online Web3 deploy to remove line breaks

As part of the implementation, I create a genesis block for the private test chain using the Ethereum protocol. I use Mist 0.9.0 Browser that has geth included as part of the package. All these tests are carried out on MacOSX.

[1.2] Source code

```
{

"config": {
"chainId": 1987,
"homesteadBlock": 0,
"eip155Block": 0,
"eip158Block": 0
},
    "nonce": "0x0000000000000042",     "timestamp": "0x0",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000
    "extraData": "0x00",     "gasLimit": "0x8000000",     "difficulty": "0x400",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000
    "coinbase": "0x3333333333333333333333333333333333333333",
    "alloc":
    {
     "0xa6c24f36c5463a1850aab17547ea83c99a56e0a6": { "balance": "200000000000000
     }
```

Figure 4.1: GenesisBlock

}

The genesis block is initialized using

/Users/User1/go-ethereum/build/bin/geth –identity "Princess" –rpc –rpccorsdomain "http://localhost:8000" –rpccorsdomain "*" –rpcport "8545" –ipcpath /Users/User1/Library/Ethereum/geth.ipc –datadir /.ethereumprivate –port "30303" –nodiscover –rpcapi "db,eth,net,web3" –networkid 1987 init /Users/User1/Library/Ethereum/CustomGenesis.json

To interact with geth through the console;

/Users/User1/go-ethereum/build/bin/geth –identity "Princess" –rpc –rpccorsdomain "http://localhost:8000" –rpccorsdomain "*" –rpcport "8545" –ipcpath /Users/User1/Library/Ethereum/geth.ipc –datadir /.ethereumprivate –port "30303" –nodiscover –rpcapi "db,eth,net,web3" –networkid 1987 console

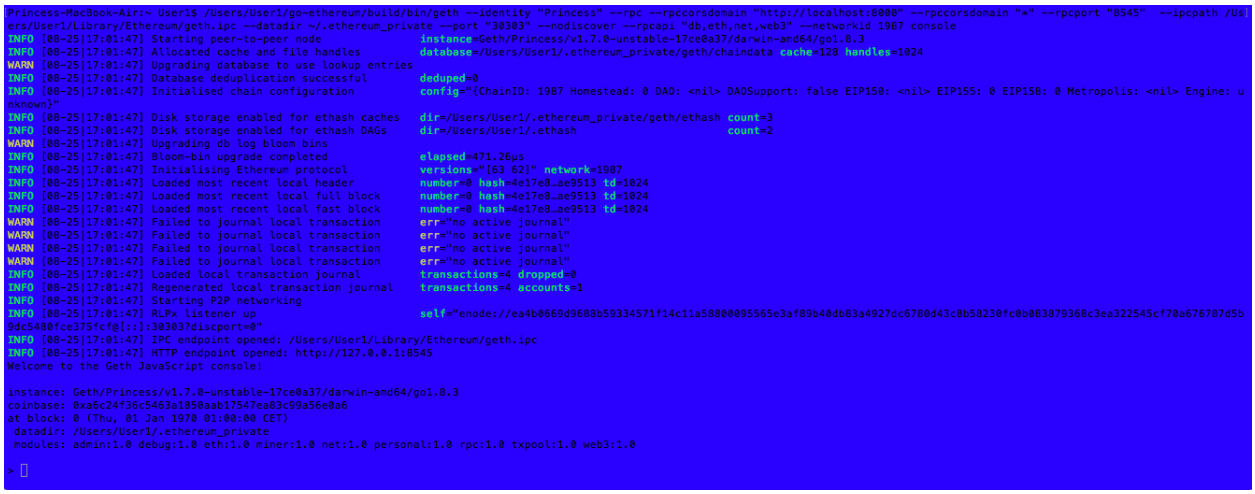

Figure 4.2: GethConsole

To initialize a new account on the testnest, that will be used as your etherbase (the address that receives mining rewards).

personal.newAccount("password")

To add a new wallet to the Custom Genesis file, adjust the alloc settings in the genesis.json file and reinitate genesis file

Commands run on remote peer node

Set it as the etherbase

miner.setEtherbase(personal.listAccounts[0])

Return account addresses you possess.

eth.accounts

Set primary account

primary = eth.accounts[0]

Check balance

balance = web3.fromWei(eth.getBalance(primary), "ether")

Select

Figure 4.3: Accounts

sudo geth –datadir path/to/custom/data/folder –networkid 1987 –bootnodes enode://98aa7ae99c72280ff329eb02c6ed2c6aa49a70c14d1ca41410854cdd46470e28c8ce99b0efb862c5

mine

–minerthreads=1

Running another private test chain with proof of authority consensus. Using puppeth, tools installed with geth, i generate and initiate a proof of work json file as below

Source code

```
{
  "config": {
    "chainId": 1988,
    "homesteadBlock": 1,
    "eip150Block": 2,
    "eip150Hash": "0x0000000000000000000000000000000000000000000000000000000000000000
    "eip155Block": 3,
    "eip158Block": 3,
    "clique": {
      "period": 15,
      "epoch": 30000
    }
  },
  "nonce": "0x0",
  "timestamp": "0x59b116d0",
  "extraData": "0x64616d6e626c6f636b636861696e0000000000000000000000000000000000000
  "gasLimit": "0x47b760",
  "difficulty": "0x1",
  "number": "0",
  "gasUsed": "0x0",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000
  "mixHash": "0x0000000000000000000000000000000000000000000000000000000000000000"
  "coinbase": "0x0000000000000000000000000000000000000000",
  "alloc": {
    "0000000000000000000000000000000000000000": {
      "balance": "0x1"
    },
    "0000000000000000000000000000000000000001": {
      "balance": "0x1"
```

```
},
"0000000000000000000000000000000000000002": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000003": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000004": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000005": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000006": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000007": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000008": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000009": {
  "balance": "0x1"
},
"000000000000000000000000000000000000000a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000000b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000000c": {
  "balance": "0x1"
},
```

```
"000000000000000000000000000000000000000d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000000e": {
  "balance": "0x1"
},
"000000000000000000000000000000000000000f": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000010": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000011": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000012": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000013": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000014": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000015": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000016": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000017": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000018": {
```

```
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000019": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001a": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001b": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001c": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001d": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001e": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000001f": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000020": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000021": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000022": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000023": {
    "balance": "0x1"
```

```
},
"0000000000000000000000000000000000000024": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000025": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000026": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000027": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000028": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000029": {
  "balance": "0x1"
},
"000000000000000000000000000000000000002a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000002b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000002c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000002d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000002e": {
  "balance": "0x1"
},
```

```
"000000000000000000000000000000000000002f": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000030": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000031": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000032": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000033": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000034": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000035": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000036": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000037": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000038": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000039": {
  "balance": "0x1"
},
"000000000000000000000000000000000000003a": {
```

```
    "balance": "0x1"
  },
  "000000000000000000000000000000000000003b": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000003c": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000003d": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000003e": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000003f": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000040": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000041": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000042": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000043": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000044": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000045": {
    "balance": "0x1"
```

```
},
"0000000000000000000000000000000000000046": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000047": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000048": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000049": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004e": {
  "balance": "0x1"
},
"000000000000000000000000000000000000004f": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000050": {
  "balance": "0x1"
},
```

```
"0000000000000000000000000000000000000051": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000052": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000053": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000054": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000055": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000056": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000057": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000058": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000059": {
  "balance": "0x1"
},
"000000000000000000000000000000000000005a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000005b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000005c": {
```

```
    "balance": "0x1"
  },
  "000000000000000000000000000000000000005d": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000005e": {
    "balance": "0x1"
  },
  "000000000000000000000000000000000000005f": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000060": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000061": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000062": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000063": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000064": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000065": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000066": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000067": {
    "balance": "0x1"
```

```
},
"0000000000000000000000000000000000000068": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000069": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006e": {
  "balance": "0x1"
},
"000000000000000000000000000000000000006f": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000070": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000071": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000072": {
  "balance": "0x1"
},
```

```
"0000000000000000000000000000000000000073": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000074": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000075": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000076": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000077": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000078": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000079": {
  "balance": "0x1"
},
"000000000000000000000000000000000000007a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000007b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000007c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000007d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000007e": {
```

```
    "balance": "0x1"
  },
  "000000000000000000000000000000000000007f": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000080": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000081": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000082": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000083": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000084": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000085": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000086": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000087": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000088": {
    "balance": "0x1"
  },
  "0000000000000000000000000000000000000089": {
    "balance": "0x1"
```

```
},
"000000000000000000000000000000000000008a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000008b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000008c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000008d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000008e": {
  "balance": "0x1"
},
"000000000000000000000000000000000000008f": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000090": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000091": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000092": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000093": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000094": {
  "balance": "0x1"
},
```

```
"0000000000000000000000000000000000000095": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000096": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000097": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000098": {
  "balance": "0x1"
},
"0000000000000000000000000000000000000099": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009a": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009b": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009c": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009d": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009e": {
  "balance": "0x1"
},
"000000000000000000000000000000000000009f": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000a0": {
```

```
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a1": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a2": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a3": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a4": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a5": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a6": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a7": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a8": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000a9": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000aa": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ab": {
    "balance": "0x1"
```

```
},
"00000000000000000000000000000000000000ac": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000ad": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000ae": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000af": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b0": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b1": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b2": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b3": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b4": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b5": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b6": {
  "balance": "0x1"
},
```

```
"00000000000000000000000000000000000000b7": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b8": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000b9": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000ba": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000bb": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000bc": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000bd": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000be": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000bf": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000c0": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000c1": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000c2": {
```

```
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c3": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c4": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c5": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c6": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c7": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c8": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000c9": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ca": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000cb": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000cc": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000cd": {
    "balance": "0x1"
```

```
},
"00000000000000000000000000000000000000ce": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000cf": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d0": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d1": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d2": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d3": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d4": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d5": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d6": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d7": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000d8": {
  "balance": "0x1"
},
```

```
"00000000000000000000000000000000000000d9": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000da": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000db": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000dc": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000dd": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000de": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000df": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000e0": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000e1": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000e2": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000e3": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000e4": {
```

```
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000e5": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000e6": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000e7": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000e8": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000e9": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ea": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000eb": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ec": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ed": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ee": {
    "balance": "0x1"
  },
  "00000000000000000000000000000000000000ef": {
    "balance": "0x1"
```

```
},
"00000000000000000000000000000000000000f0": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f1": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f2": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f3": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f4": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f5": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f6": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f7": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f8": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000f9": {
  "balance": "0x1"
},
"00000000000000000000000000000000000000fa": {
  "balance": "0x1"
},
```

```
    "00000000000000000000000000000000000000fb": {
      "balance": "0x1"
    },
    "00000000000000000000000000000000000000fc": {
      "balance": "0x1"
    },
    "00000000000000000000000000000000000000fd": {
      "balance": "0x1"
    },
    "00000000000000000000000000000000000000fe": {
      "balance": "0x1"
    },
    "00000000000000000000000000000000000000ff": {
      "balance": "0x1"
    },
    "a6c24f36c5463a1850aab17547ea83c99a56e0a6": {
      "balance": "0x2000000000000000000000000000000000000000000000000000000000000000
    }
  }
}
```

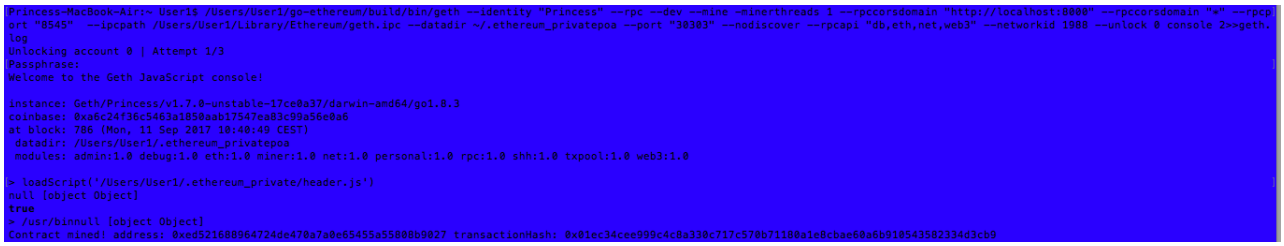

Figure 4.4: ProofofAuthority

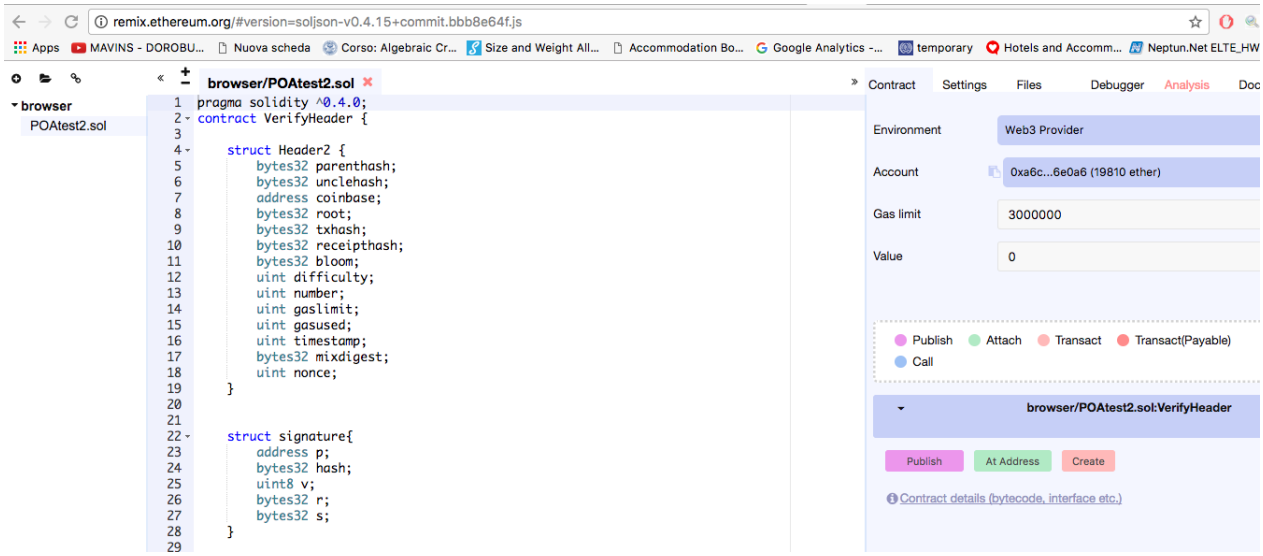

Figure 4.5: RemixIDE

To interact with geth through the console;

/Users/User1/go-ethereum/build/bin/geth –identity "Princess" –rpc –dev –mine -minerthreads 1 –rpccorsdomain "http://localhost:8000" –rpccorsdomain "*" –rpcport "8545" –ipcpath /Users/User1/Library/Ethereum/geth.ipc –datadir /.ethereumprivatepoa –port "30303" –nodiscover –rpcapi "db,eth,net,web3" –networkid 1988 –unlock 0 console 2»geth.log

4.2.1 SmartContracts

I use the online Remix IDE to write the solidity contracts;

To deploy the contract on geth; I use the Web3 deploy autogenerated script from the contract details on the IDE, remove line breaks and then deploy using the loadScript javascript file command as seen in Figure 4.4 above.

4.3 VerifyHeader.sol

The implementation of a smart contract in Ethereum for verifying the header accepts two structs containing variables for the blockheader of different types and inputs for the ecrecover function that expects four inputs. The ecrecover function is used to verify the signature of the proven signer, in this case, my local ethereum account, by recovering the public key from elliptic curve signature. The functions also verify the header byte length and has creates a token. Please see Appendix A for the source code.

4.4 Tracker.sol

The implementation of a smart contract that accepts the header and then goes on to create a token. The transaction of this token is then recorded on the previous private chain and then tracked on this public blockchain. Please see Appendix A for the source code.

4.5 Results

To write a Test Contracts, I use truffle to create simple unit tests. I installed truffle using the npm package

```
Princess-MacBook-Air:~ User1: npm install -g truffle
/usr/local/bin/truffle -> /usr/local/lib/nodemodules/truffle/build/cli.bundled.js
+ truffle@3.4.9
added 91 packages in 16.269s


Princess-MacBook-Air:~ User1$ mkdir truffleproject
Princess-MacBook-Air:~ User1$ cd truffleproject/
Princess-MacBook-Air:truffleproject User1$ truffle init
Downloading project...
Project initialized.
```

```
  Documentation: http://truffleframework.com/docs

Commands:

  Compile: truffle compile
  Migrate: truffle migrate
  Test:    truffle test



Princess-MacBook-Air:truffleproject User1$ ls
contracts migrations test truffle.js
Princess-MacBook-Air:truffleproject User1$ truffle compile /Users/User1/Desktop/s

Compiling ./contracts/ConvertLib.sol...
Compiling ./contracts/MetaCoin.sol...
Compiling ./contracts/Migrations.sol...
Writing artifacts to ./build/contracts

Princess-MacBook-Air:truffleproject User1$ ls
build contracts migrations test truffle.js
Princess-MacBook-Air:truffleproject User1$ truffle migrate
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xc456898848b57ce2ac4b99aaedb6d3c89f0ef875810dce7a59c608538cbacc85
  Migrations: 0xb6f78eede9d738172d43f47be70347c9e1879be1
Saving successful migration to network...
  ... 0x68df088e01dbd6804f160fabbbf8023d01c05232c8106df8395b17084023497d
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ... 0x45b27ecba6afe2ed013cebfc836b1772f43d6dcf1ae173b01143197b37c07191
  ConvertLib: 0x657c18d9b2524ee5c63d25673b12c403c898e9a7
```

```
  Linking ConvertLib to MetaCoin
  Deploying MetaCoin...
  ... 0xd18207c3dda3b9384a5301fc67ef155286e251dd5b4027d04e3d0da0a6fb923a
  MetaCoin: 0x74d5fb8353ff3594f07160061fdef7dbf453daf8
Saving successful migration to network...
  ... 0x6a13b6b3631abf639360376533dea9d19159305d4406719aac754f534ecbb71e
Saving artifacts...

Princess-MacBook-Air:truffleproject User1$ cd ./build/contracts

Princess-MacBook-Air:contracts User1$ ls
ConvertLib.json MetaCoin.json Migrations.json

Princess-MacBook-Air:contracts User1$ truffle test ./ethereumprivate/header.js
Using network 'development'.

Princess-MacBook-Air:contracts User1 cd /Users/User1/truffleproject/build/contrac



Princess-MacBook-Air:contracts User1 truffle test header.js
Using network 'development'.



  0 passing (1ms)

null Contract {
  _eth:
   Eth {
     _requestManager: RequestManager { provider: [Object], polls: {}, timeout: nu
     getBalance: { [Function: send] request: [Function: bound ], call: 'eth_getBa
     getStorageAt: { [Function: send] request: [Function: bound ], call: 'eth_get
     getCode: { [Function: send] request: [Function: bound ], call: 'eth_getCode'
```

```
getBlock: { [Function: send] request: [Function: bound ], call: [Function: b
getUncle: { [Function: send] request: [Function: bound ], call: [Function: u
getCompilers: { [Function: send] request: [Function: bound ], call: 'eth_get
getBlockTransactionCount:
 { [Function: send]
   request: [Function: bound ],
   call: [Function: getBlockTransactionCountCall] },
getBlockUncleCount:
 { [Function: send]
   request: [Function: bound ],
   call: [Function: uncleCountCall] },
getTransaction:
 { [Function: send]
   request: [Function: bound ],
   call: 'eth_getTransactionByHash' },
getTransactionFromBlock:
 { [Function: send]
   request: [Function: bound ],
   call: [Function: transactionFromBlockCall] },
getTransactionReceipt:
 { [Function: send]
   request: [Function: bound ],
   call: 'eth_getTransactionReceipt' },
getTransactionCount: { [Function: send] request: [Function: bound ], call: '
call: { [Function: send] request: [Function: bound ], call: 'eth_call' },
estimateGas: { [Function: send] request: [Function: bound ], call: 'eth_esti
sendRawTransaction: { [Function: send] request: [Function: bound ], call: 'e
signTransaction: { [Function: send] request: [Function: bound ], call: 'eth_
sendTransaction: { [Function: send] request: [Function: bound ], call: 'eth_
sign: { [Function: send] request: [Function: bound ], call: 'eth_sign' },
compile: { solidity: [Object], lll: [Object], serpent: [Object] },
submitWork: { [Function: send] request: [Function: bound ], call: 'eth_submi
getWork: { [Function: send] request: [Function: bound ], call: 'eth_getWork'
coinbase: [Getter],
```

```
   getCoinbase: { [Function: get] request: [Function: bound ] },
   mining: [Getter],
   getMining: { [Function: get] request: [Function: bound ] },
   hashrate: [Getter],
   getHashrate: { [Function: get] request: [Function: bound ] },
   syncing: [Getter],
   getSyncing: { [Function: get] request: [Function: bound ] },
   gasPrice: [Getter],
   getGasPrice: { [Function: get] request: [Function: bound ] },
   accounts: [Getter],
   getAccounts: { [Function: get] request: [Function: bound ] },
   blockNumber: [Getter],
   getBlockNumber: { [Function: get] request: [Function: bound ] },
   protocolVersion: [Getter],
   getProtocolVersion: { [Function: get] request: [Function: bound ] },
   iban:
    { [Function: Iban]
      fromAddress: [Function],
      fromBban: [Function],
      createIndirect: [Function],
      isValid: [Function] },
   sendIBANTransaction: [Function: bound transfer] },
transactionHash: '0xd0675334ec7276e05940c451c3b1db5ebe99a74631914837c24a008915d
address: undefined,
abi:
 [ { constant: true,
     inputs: [Array],
     name: 'checkhash',
     outputs: [Array],
     payable: false,
     type: 'function' },
   { constant: true,
     inputs: [Array],
     name: 'verifysignature',
```

```
    outputs: [Array],
    payable: false,
    type: 'function' },
  { constant: true,
    inputs: [Array],
    name: 'blockheader',
    outputs: [Array],
    payable: false,
    type: 'function' },
  { constant: true,
    inputs: [Array],
    name: 'balanceOf',
    outputs: [Array],
    payable: false,
    type: 'function' },
  { constant: false,
    inputs: [Array],
    name: 'MyToken',
    outputs: [],
    payable: false,
    type: 'function' },
  { constant: false,
    inputs: [Array],
    name: 'transfer',
    outputs: [],
    payable: false,
    type: 'function' },
  { constant: false,
    inputs: [],
    name: 'checkheader',
    outputs: [Array],
    payable: false,
    type: 'function' },
  { constant: false,
```

```
       inputs: [Array],
       name: 'accounthash',
       outputs: [Array],
       payable: false,
       type: 'function' } ] }
null Contract {
  _eth:
   Eth {
     _requestManager: RequestManager { provider: [Object], polls: {}, timeout: nu
     getBalance: { [Function: send] request: [Function: bound ], call: 'eth_getBa
     getStorageAt: { [Function: send] request: [Function: bound ], call: 'eth_get
     getCode: { [Function: send] request: [Function: bound ], call: 'eth_getCode'
     getBlock: { [Function: send] request: [Function: bound ], call: [Function: b
     getUncle: { [Function: send] request: [Function: bound ], call: [Function: u
     getCompilers: { [Function: send] request: [Function: bound ], call: 'eth_get
     getBlockTransactionCount:
      { [Function: send]
        request: [Function: bound ],
        call: [Function: getBlockTransactionCountCall] },
     getBlockUncleCount:
      { [Function: send]
        request: [Function: bound ],
        call: [Function: uncleCountCall] },
     getTransaction:
      { [Function: send]
        request: [Function: bound ],
        call: 'eth_getTransactionByHash' },
     getTransactionFromBlock:
      { [Function: send]
        request: [Function: bound ],
        call: [Function: transactionFromBlockCall] },
     getTransactionReceipt:
      { [Function: send]
        request: [Function: bound ],
```

```
   call: 'eth_getTransactionReceipt' },
getTransactionCount: { [Function: send] request: [Function: bound ], call: '
call: { [Function: send] request: [Function: bound ], call: 'eth_call' },
estimateGas: { [Function: send] request: [Function: bound ], call: 'eth_esti
sendRawTransaction: { [Function: send] request: [Function: bound ], call: 'e
signTransaction: { [Function: send] request: [Function: bound ], call: 'eth_
sendTransaction: { [Function: send] request: [Function: bound ], call: 'eth_
sign: { [Function: send] request: [Function: bound ], call: 'eth_sign' },
compile: { solidity: [Object], lll: [Object], serpent: [Object] },
submitWork: { [Function: send] request: [Function: bound ], call: 'eth_submi
getWork: { [Function: send] request: [Function: bound ], call: 'eth_getWork'
coinbase: [Getter],
getCoinbase: { [Function: get] request: [Function: bound ] },
mining: [Getter],
getMining: { [Function: get] request: [Function: bound ] },
hashrate: [Getter],
getHashrate: { [Function: get] request: [Function: bound ] },
syncing: [Getter],
getSyncing: { [Function: get] request: [Function: bound ] },
gasPrice: [Getter],
getGasPrice: { [Function: get] request: [Function: bound ] },
accounts: [Getter],
getAccounts: { [Function: get] request: [Function: bound ] },
blockNumber: [Getter],
getBlockNumber: { [Function: get] request: [Function: bound ] },
protocolVersion: [Getter],
getProtocolVersion: { [Function: get] request: [Function: bound ] },
iban:
 { [Function: Iban]
   fromAddress: [Function],
   fromBban: [Function],
   createIndirect: [Function],
   isValid: [Function] },
sendIBANTransaction: [Function: bound transfer] },
```

```
transactionHash: '0xd0675334ec7276e05940c451c3b1db5ebe99a74631914837c24a008915d
address: '0x6f477d90a822d58ca00fdd591fb1f1f454e20949',
abi:
 [ { constant: true,
     inputs: [Array],
     name: 'checkhash',
     outputs: [Array],
     payable: false,
     type: 'function' },
   { constant: true,
     inputs: [Array],
     name: 'verifysignature',
     outputs: [Array],
     payable: false,
     type: 'function' },
   { constant: true,
     inputs: [Array],
     name: 'blockheader',
     outputs: [Array],
     payable: false,
     type: 'function' },
   { constant: true,
     inputs: [Array],
     name: 'balanceOf',
     outputs: [Array],
     payable: false,
     type: 'function' },
   { constant: false,
     inputs: [Array],
     name: 'MyToken',
     outputs: [],
     payable: false,
     type: 'function' },
   { constant: false,
```

```
      inputs: [Array],
      name: 'transfer',
      outputs: [],
      payable: false,
      type: 'function' },
    { constant: false,
      inputs: [],
      name: 'checkheader',
      outputs: [Array],
      payable: false,
      type: 'function' },
    { constant: false,
      inputs: [Array],
      name: 'accounthash',
      outputs: [Array],
      payable: false,
      type: 'function' } ],
  checkhash:
   { [Function: bound ]
     request: [Function: bound ],
     call: [Function: bound ],
     sendTransaction: [Function: bound ],
     estimateGas: [Function: bound ],
     getData: [Function: bound ],
     bytes32: [Circular] },
  verifysignature:
   { [Function: bound ]
     request: [Function: bound ],
     call: [Function: bound ],
     sendTransaction: [Function: bound ],
     estimateGas: [Function: bound ],
     getData: [Function: bound ],
     'address,bytes32,uint8,bytes32,bytes32': [Circular] },
  blockheader:
```

```
 { [Function: bound ]
   request: [Function: bound ],
   call: [Function: bound ],
   sendTransaction: [Function: bound ],
   estimateGas: [Function: bound ],
   getData: [Function: bound ],
   uint256: [Circular] },
balanceOf:
 { [Function: bound ]
   request: [Function: bound ],
   call: [Function: bound ],
   sendTransaction: [Function: bound ],
   estimateGas: [Function: bound ],
   getData: [Function: bound ],
   address: [Circular] },
MyToken:
 { [Function: bound ]
   request: [Function: bound ],
   call: [Function: bound ],
   sendTransaction: [Function: bound ],
   estimateGas: [Function: bound ],
   getData: [Function: bound ],
   uint256: [Circular] },
transfer:
 { [Function: bound ]
   request: [Function: bound ],
   call: [Function: bound ],
   sendTransaction: [Function: bound ],
   estimateGas: [Function: bound ],
   getData: [Function: bound ],
   'address,uint256': [Circular] },
checkheader:
 { [Function: bound ]
   request: [Function: bound ],
```

```
    call: [Function: bound ],
    sendTransaction: [Function: bound ],
    estimateGas: [Function: bound ],
    getData: [Function: bound ],
    '': [Circular] },
  accounthash:
   { [Function: bound ]
    request: [Function: bound ],
    call: [Function: bound ],
    sendTransaction: [Function: bound ],
    estimateGas: [Function: bound ],
    getData: [Function: bound ],
    address: [Circular] },
  allEvents: [Function: bound ] }
Contract mined! address: 0x6f477d90a822d58ca00fdd591fb1f1f454e20949 transactionHa
```

Chapter 5

Discussion

5.1 Quantum Computing

The blockchain is strongly based on digital signatures generated through public-key cryptography schemes. These schemes are based on the hardness of prime factorization and the anonymity i.e identity of the users is based on private key. Quantum computers which are expected to solve in polynomial time problems related to the prime factorization will dramatically affect crypto currencies. Quantum computers will be able to break public crypto schemes (like RSA) which base their security on the exponential hardness of prime factorization.

One is the security of the elliptic curve cryptography system used when signing transactions. A quantum computer could deduce the private key (and take the funds) if it knows the public key. The private key can be computed solving the discrete logarithm problem, which is efficient in a quantum computer. It would just need a variation of Shor's factoring algorithm:

If you only use your address except for spending, the public key is only used once, when you use the funds. The hashing will mask your public key and protect the secret one. There might be a problem if some node sees your public key and tries to outrun you and send other transactions and the public keys already in the blockchain would be vulnerable.

The second thing is the hash. Grover's algorithm could be used to find collisions. There are bounds you can apply directly to collisions (finding two sequences which hash to the same value). Collisions that are necessary to replace blocks is not efficient

in quantum computers.

Pulling a 51 percent attack, amassing a lot of computer power than building a scalable quantum computer, in the short/medium term. The discrete logarithm problem and finding collisions for SHA256 are only hard for classical computers.

In extreme cases;

The Elliptic Curve Digital Signature Algorithm would be broken because quantum computers can easily decrypt the private key using the public key, anyone with a quantum computer can extract cryptocurrencies using the corresponding public key.

Hashing would become exponentially difficult. There's already a predicted escalation in mining difficulty due to the advent of ASIC, and quantum computers would create a spike in mining difficulty to which ASIC mining effects pale in comparison. This would lead to hyperinflation.

The hashing advantage of quantum computer is curtailed by block mining limitations as "The difficulty is the measure of how difficult it is to find a new block compared to the easiest it can ever be. It is recalculated every 2016 blocks to a value such that the previous 2016 blocks would have been generated in exactly two weeks had everyone been mining at this difficulty. This will yield, on average, one block every ten minutes. As more miners join, the rate of block creation will go up. As the rate of block generation goes up, the difficulty rises to compensate which will push the rate of block creation back down." This means that it will drastically increase the mining difficulty, exponentially more than ASIC miner already have. This gives miners with quantum computers, a major advantage, to the point of being considered a monopoly. Unless quantum computers either become publicly available or are given their own class for hashing purposes, to limit their mining advantage

Miners with access to quantum computers have an unfair mining advantage, which can be used to manipulate the value and distribution of crypto currencies.

Quantum computer's hashing power can also be used as voting power. If a coalition of people with scalable quantum computers could generate enough hashes to comprise over 51 percent of the total hashes,they could use that power to greatly manipulate the blockchain network.

Chapter 6

Conclusion

Rather than having each and every node computing each and every smart contract, with the use of two blockchains, one private chain running a proof of authority consesus algorithm and another simulating a public chain running a proof of work consesus algorithm, we explore the possible interoperability between the two; public and private distributed ledgers as a hybrid blockchain. An asset is moved on a a private test chain to a public blockchain, the signature is verified on the private chain and the transaction is tracked on the public chain.

This mechanism allows only for a subset of nodes to verify each transaction and as long as there enough nodes verifying each transaction, the system is still highly secure but the nodes are not too many so that the system can still process as many transactions in parallel.

The advantages of this design include high throughput, low latency and anonymity.

Chapter 7

Bibliography

Bibliography

[1] BitCoin. https://bitcoin.org/bitcoin.pdf. Accessed: 2017-06-14.

[1.1] Mauve Paper. https://scalingbitcoin.org/MauvePaper-VitalikButerin. Accessed: 2017-06-14.

[1.2] Private Network. https://github.com/ethereum/go-ethereum/wiki/Private-network. Accessed: 2017-06-14.

[1.3] Proof of Authority. https://github.com/ethereum/EIPs/issues/225. Accessed: 2017-06-14.

[1.4] Block Header. https://github.com/ethereum/yellowpaper. Accessed: 2017-06-14.

[2] Proof of Stake. https://bitcointalk.org/index.php?topic=27787.0. Accessed: 2017-06-14.

[3] Public and Private BlockChains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/. Accessed: 2017-06-14.

[4] Sharding FAQ. https://github.com/ethereum/wiki/wiki/Sharding-FAQ. Accessed: 2017-06-14.

[5] Micropayment Channel. https://bitcoin.org/en/developer-guide#micropayment-channel. Accessed: 2017-06-14.

[6] Ethereum-Development-Tutorial. https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial. Accessed: 2017-06-14.

[7] BTC Relay. http://btc-relay.readthedocs.io/en/latest/frequently-asked-questions.html#how-does-btc-relay-work. Accessed: 2017-06-14.

[8] Side Chains. https://blockstream.com/sidechains.pdf. Accessed: 2017-06-14.

[9] Ethereum Introduction. http://ethdocs.org/en/latest/introduction/index.html. Accessed: 2017-06-14.

Appendices

Appendix A

Source code of an Ethereum Smart Contract to accept a header for a private blockchain

```
 pragma solidity ^0.4.0;
contract VerifyHeader {

    struct Header2 {
        bytes32 parenthash;
        bytes32 unclehash;
        address coinbase;
        bytes32 root;
        bytes32 txhash;
        bytes32 receipthash;
        bytes32 bloom;
        uint difficulty;
        uint number;
        uint gaslimit;
        uint gasused;
        uint timestamp;
        bytes32 mixdigest;
        uint nonce;
    }


    struct signature{
```

```
        address p;
        bytes32 hash;
        uint8 v;
        bytes32 r;
        bytes32 s;
    }

    struct HeaderConstants{
        uint64 epochlength;
        uint64 blockPeriod;
        int extraVanity;
        int extraSeal;
        uint nonceAuthVote;
        uint nonceDropVote;
        bytes32 unclehash;
        uint diffInTurnSignatures;
        uint diffNoTurnSignatures;
    }



    Header2 [] public blockheader;
    /*signature [] public blocksignature;*/
    /*address  private _addressp; */

    function checkheader()
    public returns(uint) {
        if (blockheader.length >= 508){
         return blockheader.length;
        }
    }

    /* Calculate AuthoritySignature hash for the sealer account address */
     function accounthash(address p)
```

```
    public returns (bytes32) {
         return keccak256(p);
    }

    /* Calculate ECPublic Key of sealer's account hash */


     /* Verify Signature hash for the sealer account address */
    function verifysignature(address p, bytes32 hash, uint8 v, bytes32 r, bytes32
      constant returns(bool) {
        return ecrecover(hash, v, r, s) == p;
    }


    /* Check if parent hash matches previous block */
    function checkhash(bytes32 parenthash)
      constant returns(bool) {
        if (parenthash == 0x0000000000000000000000000000000000000000000000000000
      return true;
           }
      }
}
    /* selling EC20 tokens; from ethereum.org */

    /* This creates an array with all balances */
    mapping (address => uint256) public balanceOf;

    /* Initializes contract with initial supply tokens to the creator of the cont
    function MyToken(
        uint256 initialSupply
        ) {
        balanceOf[msg.sender] = initialSupply;              // Give the creator a
    }
```

```
    /* Send coins */
    function transfer(address _to, uint256 _value) {
        require(balanceOf[msg.sender] >= _value);           // Check if the sende
        require(balanceOf[_to] + _value >= balanceOf[_to]); // Check for overflow
        balanceOf[msg.sender] -= _value;                    // Subtract from the
        balanceOf[_to] += _value;                           // Add the same to th
    }
}
```

Source code of an Ethereum Smart Contract to track this transaction on a public blockchain

```
  ragma solidity ^0.4.0;
import "./POA2.sol";
contract TrackBalance {

    address private _trustedaddress;

    function ProofofAuthorityTracker(address p) {
        _trustedaddress= p;
    }

    function processTransaction(uint256 _value) returns (int256) {
        log0("processTransaction called");

        // only allow trusted address
        if (msg.sender == _trustedaddress) {
            log1("processTransaction address _to, ", bytes32(_value));
            return 1;
        }

        log0("processTransaction failed");
        return 0;
    }
```

```
}
```

Web3 deploy of an Ethereum Smart Contract to verify the header on a private blockchain

```
var browser_poatest2_sol_verifyheaderContract = web3.eth.contract([{"constant":tr
var browser_poatest2_sol_verifyheader = browser_poatest2_sol_verifyheaderContract
   {
     from: web3.eth.accounts[0],
     data: '0x6060604052341561000f57600080fd5b5b61077e8061001f6000396000f3006060
     gas: '4300000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + ' transacti
    }
 })
```

Web3 deploy of an Ethereum Smart Contract to track this transaction on a public blockchain

```
var browser_tracker2_sol_trackbalanceContract = web3.eth.contract([{"constant":fa
var browser_tracker2_sol_trackbalance = browser_tracker2_sol_trackbalanceContract
   {
     from: web3.eth.accounts[0],
     data: '0x6060604052341561000f57600080fd5b5b61023d8061001f6000396000f3006060
     gas: '4300000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + ' transacti
    }
 })
```

Source code of an Ethereum Smart Contract for a Payment Channel

```
   pragma solidity ^0.4.0;

contract Channel {

address public channelSender;
address public channelRecipient;
uint public startDate;
uint public channelTimeout;
mapping (bytes32 => address) signatures;

function Channel(address to, uint timeout) payable {
channelRecipient = to;
channelSender = msg.sender;
startDate = now;
channelTimeout = timeout;
}

function CloseChannel(bytes32 h, uint8 v, bytes32 r, bytes32 s, uint value){

address signer;
bytes32 proof;

// get signer from signature
signer = ecrecover(h, v, r, s);

// signature is invalid, throw
if (signer != channelSender && signer != channelRecipient) throw;

proof = sha3(this, value);

// signature is valid but doesn't match the data provided
if (proof != h) throw;

if (signatures[proof] == 0)
```

```
signatures[proof] = signer;
else if (signatures[proof] != signer){
// channel completed, both signatures provided
if (!channelRecipient.send(value)) throw;
selfdestruct(channelSender);
}

}

function ChannelTimeout(){
if (startDate + channelTimeout > now)
throw;

selfdestruct(channelSender);
}

}
```

Source code; BTC Relay

```
A contract that can process Bitcoin transactions relayed to it via
BTC Relay.  This stores the Bitcoin transaction hash and the Ethereum block
number

contract BitcoinProcessor {
    uint256 public lastTxHash;
    uint256 public ethBlock;

    address private _trustedBTCRelay;

    function BitcoinProcessor(address trustedBTCRelay) {
        _trustedBTCRelay = trustedBTCRelay;
    }

    // processTransaction should avoid returning the same
```

```
    // value as ERR_RELAY_VERIFY (in constants.se)
    //
    // this exact function signature is required as it has to match
    // the signature specified in BTCRelay
    function processTransaction(bytes txn, uint256 txHash) returns (int256) {
        log0("processTransaction called");

        // only allow trustedBTCRelay
        if (msg.sender == _trustedBTCRelay) {
            log1("processTransaction txHash, ", bytes32(txHash));
            ethBlock = block.number;
            lastTxHash = txHash;
            // parse & do whatever with txn
            // For example, you should probably check if txHash has already
            // been processed, to prevent replay attacks.
            return 1;
        }

        log0("processTransaction failed");
        return 0;
    }
}
```

Source code for Verifying Block Header Signatures

```
contract VerifySignature {
    address owner;

    function VerifySignature() {
        owner = msg.sender;
    }

    function priv() returns(bytes32) {
        return
        0xeab5f6141b4c66877f178f8b87c804d380af6d5404edc249d2c388dbcc542977;
```

```
}

function r() returns(bytes32) {
    return
    0x0b7effb7704f726bc64139753dc2d0a3929af2309dd2930ad7a722f5b214cf6e;
}

function s() returns(bytes32) {
    return
    0x73a461ce418e9e483f13a98c0cba5cddf07f647ea1d6ba2e88d494dfcd411c9c;
}

function v() returns(uint8) {
    return 32;  //equal to 0x20
}

function v2() returns(uint8) {
    return v()-4;
}

function hashToSign() returns(bytes32) {
    return
    0x58e2f335bbd6f2b0da93eae19342e7309654fbfeed9a214a1e5d835ac09cc226;
}

function expectedAddress() returns(address) {
    return
    0x31031df1d95a84fc21e80922ccdf83971f3e755b;
}

function isValid() returns(bool) {   //this returns true!!!
    return expectedAddress()==testECRecover();
}
```

```
    function testECRecover() returns(address) {
        return ecrecover(
            hashToSign(),
            v2(),
            r() ,
            s()
            );
    }

    function kill(){
        if (msg.sender == owner) suicide(msg.sender);
    }
}
```

Source code for a Genesis Block

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <inttypes.h>
#include <ctype.h>
#include <string.h>
#include <time.h>

//Copied from Bitcoin source
const uint64_t COIN = 100000000;
const uint64_t CENT = 1000000;

uint32_t OP_CHECKSIG = 172; // This is expressed as 0xAC
bool generateBlock = false;
uint32_t startNonce = 0;
uint32_t unixtime = 0;
```

```
typedef struct {
/* Hash of Tx */
uint8_t merkleHash[32];

/* Tx serialization before hashing */
uint8_t *serializedData;

/* Tx version */
uint32_t version;

/* Input */
uint8_t numInputs; // Program assumes one input
uint8_t prevOutput[32];
uint32_t prevoutIndex;
uint8_t *scriptSig;
uint32_t sequence;

/* Output */
uint8_t numOutputs; // Program assumes one output
uint64_t outValue;
uint8_t *pubkeyScript;

/* Final */
uint32_t locktime;
} Transaction;

// test
void byteswap(uint8_t *buf, int length)
{
int i;
uint8_t temp;

for(i = 0; i < length / 2; i++)
{
```

```
temp = buf[i];
buf[i] = buf[length - i - 1];
buf[length - i - 1] = temp;
}
}

//two functions are borrowed from cgminer.
char *bin2hex(const unsigned char *p, size_t len)
{
char *s = malloc((len * 2) + 1);
unsigned int i;

if (!s)
return NULL;

for (i = 0; i < len; i++)
sprintf(s + (i * 2), "%02x", (unsigned int) p[i]);

return s;
}

size_t hex2bin(unsigned char *p, const char *hexstr, size_t len)
{
int ret = 0;
size_t retlen = len;

while (*hexstr && len) {
char hex_byte[4];
unsigned int v;

if (!hexstr[1]) {
return ret;
}
```

```
memset(hex_byte, 0, 4);
hex_byte[0] = hexstr[0];
hex_byte[1] = hexstr[1];

if (sscanf(hex_byte, "%x", &v) != 1) {
return ret;
}

*p = (unsigned char) v;

p++;
hexstr += 2;
len--;
}

if (len == 0 && *hexstr == 0)
ret = retlen;

return ret;
}

Transaction *InitTransaction()
{
Transaction *transaction;

transaction = calloc(1, sizeof(*transaction));
if(!transaction)
{
return NULL;
}

// Set some initial data that will remain constant throughout the program
transaction->version = 1;
transaction->numInputs = 1;
```

```
transaction->numOutputs = 1;
transaction->locktime = 0;
transaction->prevoutIndex = 0xFFFFFFFF;
transaction->sequence = 0xFFFFFFFF;
transaction->outValue = 50*COIN;

// We initialize the previous output to 0 as there is none
memset(transaction->prevOutput, 0, 32);

return transaction;
}

int main(int argc, char *argv[])
{
Transaction *transaction;
unsigned char hash1[32], hash2[32];
char timestamp[255], pubkey[132];
uint32_t timestamp_len = 0, scriptSig_len = 0,
pubkey_len = 0, pubkeyScript_len = 0;
uint32_t nBits = 0;

if((argc-1) < 3)
{
fprintf(stderr, "Usage: genesisgen [options]
<pubkey> \"<timestamp>\" <nBits>\n");
return 0;
}

pubkey_len = strlen(argv[1]) / 2;
// One byte is represented as two hex characters
timestamp_len = strlen(argv[2]);

if(pubkey_len != 65)
{
```

```
fprintf(stderr, "Invalid public key length! %s\n", argv[1]);
return 0;
}

if(timestamp_len > 254 || timestamp_len <= 0)
{
fprintf(stderr, "Size of timestamp is 0
or exceeds maximum length of 254 characters!\n");
return 0;
}

transaction = InitTransaction();
if(!transaction)
{
fprintf(stderr, "Could not allocate memory! Exiting...\n");
return 0;
}

strncpy(pubkey, argv[1], sizeof(pubkey));
strncpy(timestamp, argv[2], sizeof(timestamp));
sscanf(argv[3], "%lu", (long unsigned int *)&nBits);

pubkey_len = strlen(pubkey) >> 1;
scriptSig_len = timestamp_len;

// Encode pubkey to binary and prepend pubkey size, then append the OP_CHECKSIG
byte
transaction->pubkeyScript = malloc((pubkey_len+2)*sizeof(uint8_t));
pubkeyScript_len = hex2bin(transaction->pubkeyScript+1, pubkey, pubkey_len);
transaction->pubkeyScript[0] = 0x41; // A public key is 32 bytes X coordinate,
32 bytes Y coordinate and one byte 0x04, so 65 bytes i.e 0x41 in Hex.
pubkeyScript_len+=1;
transaction->pubkeyScript[pubkeyScript_len++] = OP_CHECKSIG;
```

```
// Encode timestamp to binary
transaction->scriptSig = malloc(scriptSig_len*sizeof(uint8_t));
uint32_t scriptSig_pos = 0;


// size of the nBits is calculated
if(nBits <= 255)
{
transaction->scriptSig[scriptSig_pos++] = 0x01;
transaction->scriptSig[scriptSig_pos++] = (uint8_t)nBits;
}
else if(nBits <= 65535)
{
transaction->scriptSig[scriptSig_pos++] = 0x02;
memcpy(transaction->scriptSig+scriptSig_pos, &nBits, 2);
scriptSig_pos+=2;
}
else if(nBits <= 16777215)
{
transaction->scriptSig[scriptSig_pos++] = 0x03;
memcpy(transaction->scriptSig+scriptSig_pos, &nBits, 3);
scriptSig_pos+=3;
}
else //else if(nBits <= 4294967296LL)
{
transaction->scriptSig[scriptSig_pos++] = 0x04;
memcpy(transaction->scriptSig+scriptSig_pos, &nBits, 4);
scriptSig_pos+=4;
}


//  PUSH 1 byte on the stack which in this case is 0x04 or just 4
transaction->scriptSig[scriptSig_pos++] = 0x01;
transaction->scriptSig[scriptSig_pos++] = 0x04;
```

```
transaction->scriptSig[scriptSig_pos++] = (uint8_t)scriptSig_len;

scriptSig_len += scriptSig_pos;
transaction->scriptSig = realloc
(transaction->scriptSig,
scriptSig_len*sizeof(uint8_t));
memcpy(transaction->scriptSig+scriptSig_pos,
(const unsigned char *)timestamp, timestamp_len);

// assuming some values will have the same size
uint32_t serializedLen =
4    // tx version
+1   // number of inputs
+32  // hash of previous output
+4   // previous output's index
+1   // 1 byte for the size of scriptSig
+scriptSig_len
+4   // size of sequence
+1   // number of outputs
+8   // 8 bytes for coin value
+1   // 1 byte to represent size of the pubkey Script
+pubkeyScript_len
+4;   // 4 bytes for lock time

// Now let's serialize the data
uint32_t serializedData_pos = 0;
transaction->serializedData = malloc(serializedLen*sizeof(uint8_t));
memcpy(transaction->serializedData+serializedData_pos, &transaction->version,4);
serializedData_pos += 4;
memcpy(transaction->serializedData+serializedData_pos, &transaction->numInputs,1)
serializedData_pos += 1;
memcpy(transaction->serializedData+serializedData_pos, transaction->prevOutput,
32);
```

```
serializedData_pos += 32;
memcpy(transaction->serializedData+
serializedData_pos,&transaction->prevoutIndex,4);
serializedData_pos += 4;
memcpy(transaction->serializedData+serializedData_pos, &scriptSig_len, 1);
serializedData_pos += 1;
memcpy(transaction->serializedData+serializedData_pos, transaction->scriptSig,
scriptSig_len);
serializedData_pos += scriptSig_len;
memcpy(transaction->serializedData+serializedData_pos, &transaction->sequence,4);
serializedData_pos += 4;
memcpy(transaction->serializedData+serializedData_pos, &transaction->numOutputs,
serializedData_pos += 1;
memcpy(transaction->serializedData+serializedData_pos, &transaction->outValue,8);
serializedData_pos += 8;
memcpy(transaction->serializedData+serializedData_pos, &pubkeyScript_len, 1);
serializedData_pos += 1;
memcpy(transaction->serializedData+serializedData_pos, transaction->pubkeyScript,
pubkeyScript_len);
serializedData_pos += pubkeyScript_len;
memcpy(transaction->serializedData+serializedData_pos, &transaction->locktime, 4)
serializedData_pos += 4;

// Now that the data is serialized
//hash it with SHA256 and then hash that result to get merkle hash
SHA256(transaction->serializedData, serializedLen, hash1);
SHA256(hash1, 32, hash2);


memcpy(transaction->merkleHash, hash2, 32);

char *merkleHash = bin2hex(transaction->merkleHash, 32);
byteswap(transaction->merkleHash, 32);
char *merkleHashSwapped = bin2hex(transaction->merkleHash, 32);
```

```
char *txScriptSig = bin2hex(transaction->scriptSig, scriptSig_len);
char *pubScriptSig = bin2hex(transaction->pubkeyScript, pubkeyScript_len);
printf("\nCoinbase: %s\n\nPubkeyScript: %s\n\nMerkle Hash: %s\nByteswapped:
%s\n",txScriptSig, pubScriptSig, merkleHash, merkleHashSwapped);

//if(generateBlock)
{
printf("Generating block...\n");
if(!unixtime)
{
unixtime = time(NULL);
}

unsigned char block_header[80], block_hash1[32], block_hash2[32];
uint32_t blockversion = 1;
memcpy(block_header, &blockversion, 4);
memset(block_header+4, 0, 32);
byteswap(transaction->merkleHash, 32); // We swapped it before, so do it again.
memcpy(block_header+36, transaction->merkleHash, 32);
memcpy(block_header+68, &unixtime, 4);
memcpy(block_header+72, &nBits, 4);
memcpy(block_header+76, &startNonce, 4);

uint32_t *pNonce = (uint32_t *)(block_header + 76);
uint32_t *pUnixtime = (uint32_t *)(block_header + 68);
unsigned int counter, start = time(NULL);
while(1)
{
SHA256(block_header, 80, block_hash1);
SHA256(block_hash1, 32, block_hash2);

unsigned int check = *((uint32_t *)(block_hash2 + 28));
// The hash is in little-endian, so we check the last 4 bytes.
if(check == 0) // \x00\x00\x00\x00
```

```
{
byteswap(block_hash2, 32);
char *blockHash = bin2hex(block_hash2, 32);
printf("\nBlock found!\nHash: %s\nNonce: %u\nUnix time: %u", blockHash,
startNonce, unixtime);
free(blockHash);
break;
}

startNonce++;
counter+=1;
if(time(NULL)-start >= 1)
{
printf("\r%d Hashes/s, Nonce %u\r", counter, startNonce);
counter = 0;
start = time(NULL);
}
*pNonce = startNonce;
if(startNonce > 4294967294LL)
{
//printf("\nBlock found!\nHash: %s\nNonce: %u\nUnix time: %u", blockHash,
startNonce, unixtime);
unixtime++;
*pUnixtime = unixtime;
startNonce = 0;
}
}
}


// cleanup
free(merkleHash);
free(merkleHashSwapped);
free(txScriptSig);
```

```
free(pubScriptSig);
free(transaction->serializedData);
free(transaction->scriptSig);
free(transaction->pubkeyScript);
free(transaction);

return 0;
}
```

Source code

```
//generic verification of a signature

def verify_sign(public_key_loc, signature, data):
 '''
 Verifies with a public key from whom the data came that it was indeed
 signed by their private key
 param: public_key_loc Path to public key
 param: signature String signature to be verified
 return: Boolean. True if the signature is valid; False otherwise.
 '''
 from Crypto.PublicKey import RSA
 from Crypto.Signature import PKCS1_v1_5
 from Crypto.Hash import SHA256
 from base64 import b64decode
 pub_key = open(public_key_loc, "r").read()
 rsakey = RSA.importKey(pub_key)
 signer = PKCS1_v1_5.new(rsakey)
 digest = SHA256.new()
 # Assumes the data is base64 encoded to begin with
 digest.update(b64decode(data))
 if signer.verify(digest, b64decode(signature)):
     return True
 return False
```