# TABLE OF CONTENTS

# LIST OF ACRONYMS

**ETH** - Ethereum

**EVM** - Ethereum Virtual Machine

**IPFS** - Interplanetary File System

**PoS** - Proof of Stake

**PoW** - Proof of Work

**PoSt** - Proof of Space + Storage

**PoH** - Proof of History

**Tx** - Transaction

**Wei** - The smallest unit of ETH (10 power -18)

**Gwei** - 10 power 6 Wei

# OPCODES

| OPCODE | Name | Gas | Description |
|---|---|---|---|
| 51 | MLOAD | 3* | Memory load data |
| 52 | MSTORE | 3* | Memory store data |
| 54 | SLOAD | 100* | Storage load data |
| 55 | SSTORE | 100* | Storage store data |

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

This chapter introduces various concepts corresponding to the project domain and elaborates on the goals accomplished by the ideas behind the project. The basic concepts are as follows:

## 1.1 INTRODUCTION

### 1.1.1 Ethereum, EVM, and Smart-Contracts

An EVM-chain is a decentralized state machine that maintains state changes of the blockchain through cryptographically linked state objects called "Blocks". Each block contains transactions, these transactions are the state changes on the network. In 2018, Zubin Koticha [1][2], observed that the number of transactions in a block is defined by its gas limit. EVM-chains can be visualized as a global spanning computer that has hardware resources in the form of miners for execution, analogous to CPUs, and nodes analogous to Hard Disks. Smart contracts are programs written on a blockchain that can be executed by miners. Smart contracts get executed transparently to the code in the contact. This happens because there are no middle-men involved in the operations of a blockchain that can alter the data created or change the code of a smart contract once deployed onto the network. This functionality of Blockchains is called "immutability". Blockchains offer high levels of security and transparency as the blocks, once mined, cannot be changed without changing all the blocks created after it. Blocks are mined by miners and contain several transactions in a block. There exists a strong sense of centralization in today's world with respect to any operation on the internet. In essence, the services made available to the user can be disabled at the click of a button. Users have often paid the price of centralization in the forms of blocked accounts, operations being made unavailable to them, and long review hours to get their accounts back. Smart Contracts eliminate the centralization aspect and make the user experience more autonomous and based on cryptographical "truths" rather than "trust" in a random 3rd party.

### 1.1.2 Blockchain Data Storage

Blockchains need to keep data in smart contracts for computation purposes. Since the introduction of Blockchains, storage has been a fascinating aspect for many users. Nevertheless, there are apprehensions owing to the system's immutability and public data storage, making it a non-ideal place to store "sensitive" information as anyone can read the data from any smart contract, be it private or public access variables. Thus storing sensitive information on blockchains is not possible unless the chain is 100% privatized which essentially removes the decentralization aspect of the blockchain. In 2016, Daira Hopwood et al. [3] and in 2018, Alexey Pertsev et al. [4] proposed the use of advanced alternative algorithms such as ring-fields and Zero Knowledge algorithms but they not only required a new blockchain architecture but also a major investment into the field in the form of resources, computation and design. Additionally, they would also require people to migrate to the new chain. Storing smaller amounts of data is feasible but the main hurdle exists in storing large amounts of data on the blockchain due to the high cost of on-chain storage. This "hurdle" comes from the design of Ethereum, and how the EVM functions. In Ethereum the data of a smart contract is stored within the smart contract. Every node in the network stores a copy of the chain, and hence the number of replicas of the data is very large, and updating a single value requires all the nodes to update the data and then prove the updation to each other, so as to maintain consensus amongst them. Ethereum is also designed to be a black box system, where everything that happens in the network stays in the network and a network can't communicate with anything outside without specialized programs created to perform the transactions.

### 1.1.3 Statelessness in Blockchains, its problem, and the future of blockchains

As Ethereum and other EVM-blockchains are state machines, every transaction alters the state of the chain. Every state change requires payment in the form of gas to update the chain and sync it across all miners and nodes. Storing and modifying data is a high frequency state change as we perform permanent storage changes of smart contracts and also sync them. A stateless network would not require to store the data on-chain and would instead require data from an external source for every transaction. This is not a bad idea as it allows the data layer to be decoupled from the architecture of blockchains. As transactions now do not include a data storage or update (SSTORE) command but rather only the MSTORE command . The state change is now limited to only being affected by execution as the data update part of it is removed. This would imply that decoupling data from the chain allows for more gas to be spent on transactions, which means the execution part can become more complicated, both logically and space-wise. Decoupling the data from the chain would also mean that we no longer have a vendor lock-in problem and data can be transferred to any other chain. The future of blockchains is towards a Network of Chains and not having one chain superior to the other. This is because distributed systems are theorized to be able to only attain 2 features out of Scalability, Security, and Decentralization, this is similar to the CAP theorem. This inability of a chain to attain all 3 is also what led to multiple chains being created, each trying to tackle a different problem.

## 1.2 PROBLEM STATEMENT

Data stored on-chain in smart contracts has several issues such as increasing the blockchain's size, increasing the cost of execution, and also has a "vendor lock-in" problem similar to cloud providers to name a few. All of these issues are unsatisfactory in the long run as blockchain usage is undergoing an exponential growth. Existing solutions aim to offload data on IPFS, which is unreliable and not widely used because of its inability to be consistent at all time, a feature that is a core principle of blockchains, which are deterministic in nature. Another issue apart from the indeterministic property of IPFS is that, instead of the creator having the power to handle the data, it exists in the hands of the network, which leads to problems such as data deletion, data access control, and lack of modifiability. Hence, there is a need to create a modifiable, reliable, and provably secure data storage option for Web3.0.

## 1.3 PROJECT OBJECTIVES

The goal is to store on-chain smart contract data on a layer-2 chain through a collection of miners causing statelessness in a state-based chain to reduce gas fees, allow for more complicated program executions , and also allow cross-chain data communication (CCDC) while using existing physical resources such as inexpensive harddisks. We aim to create our own data storage platform similar to a database to achieve this as it has an edge over IPFS which is unreliable,inconsistent and indeterministic (elaborated in problem statement) when it comes to customizability according to needs, data flow, and flexibility to suit the operations. There also exists a Querying protocol called "The Graph", which is an implementation of GraphQL on Ethereum, but this does not execute data on the blockchain and its decentralization isn't deterministic or known. We propose a protocol that aims to take the best parts of the aforementioned protocols and create a new one that involves trustless distributed storage for users looking to store data, which also doubles as means for smart-contracts to store data cheaper than if it were to store the data within the contract. Today the cost of storing 1 KB on an EVM chain is 31.24x109 giga wei or 31.24 units of currency (109 gwei = 1 unit of currency) which on Ethereum (1 ETH = 2,690.64$ as on May 20th, 2022) amounts to 21,500$ or on a Matic chain, one of the cheapest EVM chains 30$ . This project aims to reduce this storage price by creating a new protocol for storage handling which distributes the data across all the participating miners or nodes, bringing the concept of fairness to a distributed environment, that is generally manipulatable due to a lack of a central authority by computers with more computing power than another, such as the case of PoW chains or luck as in PoS.

## 1.4 ORGANIZATION OF THE REPORT

The rest of the report is organized as follows:

**Chapter 2** elaborates on the Literature Survey of existing technology and publications that have been reviewed to ideate towards the current system.

**Chapter 3** elaborates on the Architecture diagrams, Data Flow Models and Module Descriptions of individual models involved in the project.

**Chapter 4** elaborates on the implementation of the project, and lists the algorithms used to develop a working model of the project.

**Chapter 5** elaborates on the results and various statistics and metrics inferred and measured from the model respectively.

**Chapter 6** concludes the entire project and advises future developments for the idea.

# CHAPTER 2

# LITERATURE SURVEY/RELATED WORK

This section introduces various reports, papers and statistics studied to understand the existing technology to present improvements and developments to implement the idea behind the thesis.

## 2.1 BLOCKCHAIN

In NARA Blockchain Whitepaper of 2019 [5], National Archives and Records Administration took up the complex task of surveying and compiling the history of Blockchain and listing out the working methodologies. It listed the most interesting features of blockchain and how the history dates back to Satoshi Nakamoto's Bitcoin Whitepaper. An interested reader can look into the Bitcoin whitepaper to gain a more working understanding. Inspired by Bitcoin, Vitalik Buterin elaborated on a lot of the basic principles with many added to it, and termed it Ethereum. These were the first signs of Blockchain being looked at as a tangible, workable, and practical entity, and not just a theoretical dream that would one day solve the problems of humanity, both from a financial and economic standpoint.

## 2.2 BLOCKCHAIN STORAGE

The first attempts towards Blockchain storage were described by Juan Benet, in 2014 [6]. The prospect of storing data in an immutable storage solution was always a dream for almost all solution scientists and hunters as it would finally solve a number of problems. Decentralization fascinated the technology market as it enabled the dream of transferring data and currency without any intermediate centralized entity or banks. The Filecoin Whitepaper [7] talked about the essential components to create a decentralized storage system, which in essence is a storage solution on its own. The paper extensively talked about why the current technology was used, what the alternatives were, and introduced many features such as integrity checks, proof of replication, proof of space-time, Client-Miner agreements to list a few.

Inspired by the above, this project takes further points from a few more literature papers that have the ability to make the premise stronger when combined.In 2007, Giuseppe Ateniese et al. [8] gave the idea of Proving Data Possession at Untrusted Stores, allowing a definite verification of the existence of data, without revealing neither the absolute nor the relative position of Data. The above techniques when combined with the regular IPFS system gives a new sense of decentralisation. In 2018, Ben Fisch et al. [9] talked elaborately about Scaling Proof of replication to minimize wastage of data when it is reassembled from the storage, by making the reconstruction path with alternative paths optimal. It gave an idea of methods to replicate and update data efficiently by allowing integrity checks. In 2019, Ethan Ceccheti et al. [10] proposed the usage of SDAGs to create a storage solution where nodes are in a graph. Data anonymity is thus implemented as no storage node knows what data resides where and with whom; they only know the nodes they have access to and don't know what data is stored on them, thereby anonymizing the data stored.

In 2020, Roman Mühlberger eet al. [11], proposed the use of oracle networks that are critical to making more use cases for blockchains, as blockchains by itself is a closed system, nothing goes in and nothing goes out unless a "client" interacts with the network through a smart contract changing state values on-chain. Having just a client perform, this brings a centralized link in the decentralized system, and as the saying goes the system is only as strong as its weakest link, essentially breaks down the system which is exposed in the numerous DeFi attacks where a centralized data source got compromised and caused millions of dollars to be transferred incorrectly. An oracle by its name is a greek word that brings data from an external source into a community, and such oracles if decentralized bring a more cryptographical truth based data source rather than a trust based one as seen by the Chainlink protocol.

In 2020, Vitalik Buterin et al. [12], proposed combining Ghost and Casper, wherein we added to the existing Proof-of-Work process with a Proof-of-Stake process. This is game-changing in many aspects, starting from being less resource exhausting and ecologically safer. This concept can be adapted to having a base Proof-of-Storage mechanism and a Proof-Stake verification and validation on top of the base-chain. This implementation creates more incentives for miners and helps regulate miners by penalizing miners who did incorrect computations and awarding those that did it right. The Proof-of-Storage is dynamic and uses vector algebra to produce a probability distribution that ensures that miners with different storage capacities have a fair and equal chance scaled based on their storage group.

# CHAPTER 3

# SYSTEM ARCHITECTURE AND DESIGN

This section introduces the architecture diagrams, module descriptions and data flow diagram of various aspects of the project.

## 3.1 MAINNET

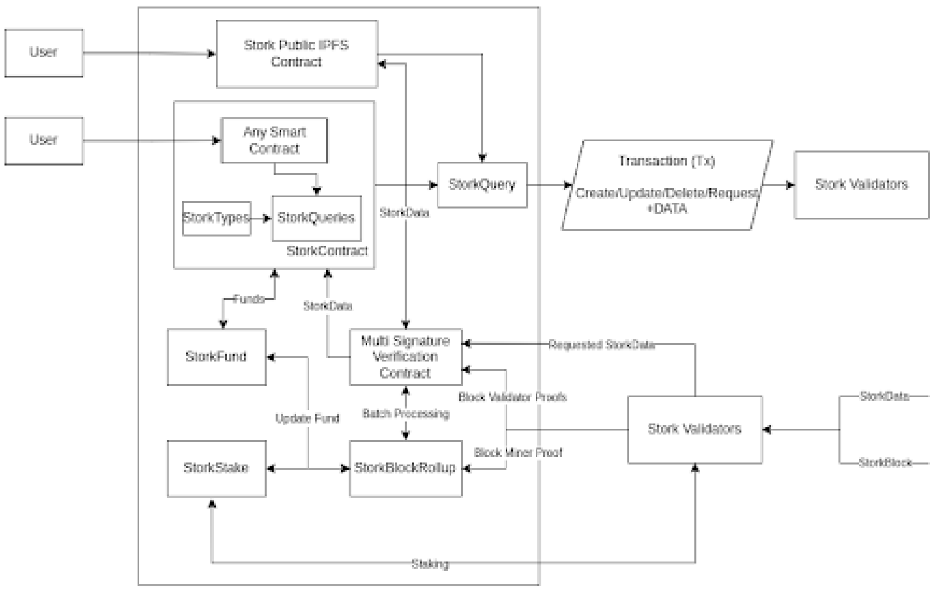


Figure 3.1: Architecture Diagram of the on-chain component

The Figure 3.1 shows the basic Archictecture diagram of the on-chain components, which are further elaborated in this section

### 3.1.1 StorkTypes

This is one of the 2 contracts that make up the StorkContract. It contains the format of the data types (Stork and Phalanx) that are required for a StorkContract to interact with StorkNet. It also stores information about the Storks and Phalanxes used in the program.

### 3.1.2 StorkQueries

This is the second contract that makes up the StorkContract. It contains the query operations that can be made by a StorkContract to interact with the StorkNet. The queries contain parameters of type Stork, StorkParameter, etc and they perform operations such as Create, Request, Update, Delete by ID and Parameters.

### 3.1.3 StorkContract

The StorkContract is a template contract built using StorkQueries, StorkTypes and also contains its own functionality such as data encoding, data decoding, and interactions with the StorkFunds Contract (explained in 3.1.3). The user created contract inherits this contract along and builds upon with the user's own logic and code.

### 3.1.4 Stork Public IPFS Contract

This contract allows for IPFS data to be indexed and queried. IPFS is a "neutral" storage type as it contains the data and the datatypes of the data within itself. This makes it compatible with the StorkNet storage system. Here web2.0 or web3.0 programs can use the IPFS library to create an IPFS file and push it onto the network for storage. As IPFS can only store and return data by IPFS CID, we adapt upon the existing benefits of IPFS by indexing the data which allows for modification of data.

### 3.1.5 StorkFunds

The contract receives funds from a StorkContract. The funds (in gwei) are converted into the maximum number of Txs that can be executed for the client based on the cost per transaction rate of the StorkNet, this parameter scales differently depending upon the network the contract is created on. The StorkBlockRollup contract then receives the client address and the number of Txs. The client can also request for a return of funds. The amount of funds being used (in gwei) is tracked so that should the client request it, the remaining funds can be transferred back.

### 3.1.6 StorkQuery

The query contract is used so that Validators can index the events emitted from the chain, this contract takes in the data that the client wants to emit as parameters. These events contain parameters such as Stork, StorkParams, StorkId, StorkType, the client address, and the destination network. Each of these events are considered a Transaction on StorkNet.

### 3.1.7 StorkStake

It enables staking and by that validates StorkValidators. The staked value (in gwei) by the validators is then locked for a certain time period. The validator can request for an early exit, and will be returned the staked amount + reward value with a reduced percentage if the full time period is not reached.

### 3.1.8 StorkBlockRollup

This is used to initiate the execution and proof of transactions on the network. We batch together several transactions into something called a "StorkBlock". This is done to reduce the gas required to update the chain on mainnet. This is also the contract that controls the number of Txs left for a StorkContract and holds the number of Txs executed by a StorkValidator for reward computation.

### 3.1.9 Multi Signature Verification Contract

This contract interacts with the StorkBlockRollup and the Validators by validating the incoming data from StorkNet. This is done so that the update on number of Txs processed by a validator or consumed by a contract are validated mathematically by multiple users.

### 3.1.10 Create/Update/Delete/Request

These are the operations performed by the contract and emitted as an event, and are known as Transactions. These events are of two types: by ID and by Parameters for the operations of Create, Update, Delete, and Request. These events are then picked up by StorkValidators who then process the data and push them onto the StorkChain.

### 3.1.11 Stork Validators

They are the ones responsible for storing the data emitted by the StorkContracts. They are required to stake an amount of the native token to be evaluated as a Validator. They also take part in the Verification process in the Multi Signature Verification Contract.

## 3.2 STORKCHAIN

The StorkChain has the following components

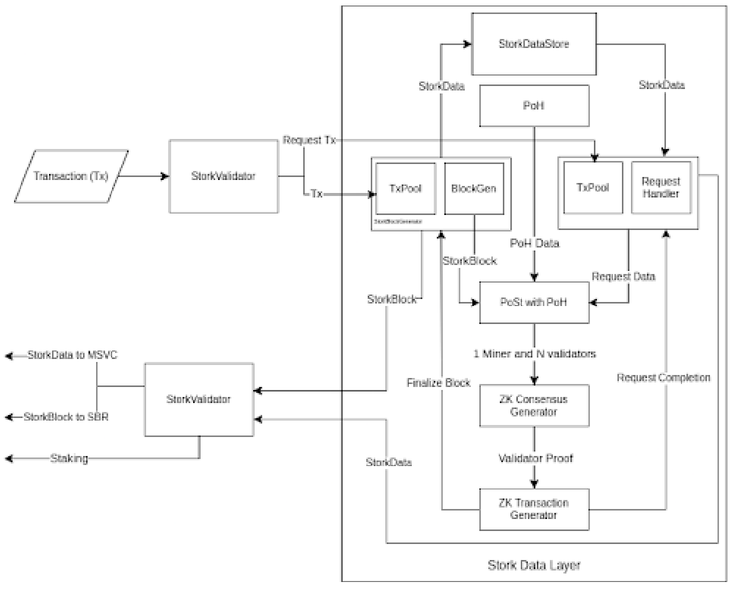


**Figure 3.2: Architecture Diagram of Decentralized Data Oracle Network**

The Figure 3.2 shows the Architecture Diagram of the Data Orable Network whose components have been elaborated in this section.

### 3.2.1 StorkValidator

The StorkValidator receives the data and operations from StorkQuery, collectively called a Transaction and forwards them to the StorkBlockGenerator for validation. StorkValidators are the members of StorkNet that get elected to "own the Block" that is being created and to also handle the block, this happens using the inhouse made PoSt+PoH algorithm.

### 3.2.2 StorkBlockGenerator

This contract creates a block of type StorkBlock that is similar to the one in the MSVC and the StorkBlockRollup contracts on the mainnet. This contains information about the validators that are interacting with the contract, the transactions, the client addresses, etc. These Txs are then packaged into a Block and then sent for consensus.

### 3.2.3 PoSt + PoH

The PoSt part uses the transaction counts of validators called the PoH stage and using it generates a list of Validators for the Tx based on a fair probability distribution using a dynamically growing key called the nonce that is generated from random values submitted by the validators. It then chooses a miner from the list of elected Validators for the Tx, and then runs again to provide validators with Txs.

### 3.2.4 ZK Consensus Generator

Here we use the concept of ZK-SNARKS and generate two challenges, one being the validator proof for the Block, that results in true only if the validators are used as input parameters. The second being a blockhash is also generated in the same way that uses the blockMiner's address, the Txs, etc. The blockhash and the validator proof are then added to the block.

### 3.2.5 ZK Transaction Generator

This contract also uses ZK-SNARKS and generates a ZK challenge for each Tx in the block, which requires the Tx's data and the validator of that Tx to result in true. These proofs validate the data's integrity and the validator at the same time with low computation power, as they are one way hashable functions.

### 3.2.6 Request Handler

This contract handles the data request operations that are requested by a StorkContract. As the network supports cross chain operability, the target network and client address are also supplied. Handling requests is a time sensitive operation and on-chain smart contracts should not wait extended periods for the data. This contract is a modification of StorkBlockGenerator with its own quicker consensus mechanism, so that client contracts can be supplied the data as soon as possible.

### 3.2.7 StorkDataStore

Data storage of StorkNet is managed using the StorkBlockGenerator, which receives Txs from validators. The transaction queries that are involved are Create, Update, and Delete. The Reading of data happens from the Request Handler contract, similar to StorkBlockGenerator a consensus phase occurs and a single miner is given permission to read the data and send it to the contract on mainnet that requested it.

# CHAPTER 4

# IMPLEMENTATION OF THE WORK

This section introduces the implementation aspects of all the Modules introduced in the architecture diagram. The algorithms used and methods introduced are elaborately described

## 4.1 APPROACH OF THE PROJECT

The approach was to create a modular approach to the design of blockchains on the data side of things, and to achieve this by offloading data from on-chain smart contracts to an off-chain. This new chain would then function as a data layer to the existing blockchain “layers” i.e. the execution and the consensus layers. The data layer would not function in the same way as the L-2 chains function. Users will not be able to deploy smart contracts on it, but instead interact with this chain through their own smart contracts on the execution layer. The interaction point from the execution layer is chain agnostic, it can be Ethereuem, an L1 chain or an L-2 chain like Matic or Avalanche.

L-2 chains are created to extend the capabilities of L-1 chains as they allow for faster and cheaper transactions, the reason being they don’t have the same restriction as an L-1 chain on some parts of the network architecture such as security or centralization. Once a lot of transactions are executed on these networks a block is generated and that is then dumped as proof on the L-1 chain (Ethereum) to show proof of execution.

As this project is a data layer and not an execution layer there has to be a way to interact with the data layer, this is where the querying approach came from. Using queries like how regular programs interact with databases, smart contracts and pretty much any program can query data with StorkNet, the only requirement being the data is stored in a “neutral” data format, such as JSON, IPFS, or StorkType, this makes StorkNet a web2 and web3 compatible database of sorts.

For the queryable part of the project, a library was created that allows for smart contracts to interact with StorkNet, the queries being Create, Request, Update, and Delete. As a blockchain is a black box and it can’t directly interact with anything external to it, the query handling is performed by “Ethereum compatible programs”. They are special programs that can interact with the network by reading data from functions, creating transactions, sending tokens, calling functions to execute, listening to events, etc.

The queries are picked up as event requests by StorkValidators that have taken part in the Staking process. They listen to these requests and collect them. Once each StorkValidator receives an event they push the event (now called a Transaction) onto the StorkNet. These transactions are then bundled together to form a block.

Once a block is formed, consensus of the block is attained by having the validators vote on the legitimacy of a transaction, if the transaction passes the test, it gets executed and the transaction’s details are added to the block. The network then approves a single validator, a miner who is chosen at random to submit the created Block on the mainnet.

As layer-2 solutions work in batches while supplying data to the mainnet, to reduce gas fees, Storknet uses the Multi Signature Verification and StorkBlockRollup contracts. These contracts help in processing and proving that the transactions as asked by the mainnet smart contract were executed in the right way, the most common proofs being ZK proofs (proofs where a challenge is generated, and can be publicly verified by anybody, all while not exposing the data in the proof) and more specifically ZK-SNARK (a non interactive version where a proof can be created, and challenged or validated by anyone, at any given time). These proofs are used to prove the legitimacy of the execution of queries, as fudging these proofs is quite not possible as they include one way hashing algorithms. The time it would take to figure out the data used to generate the proof is extremely high, they also contain the address of the validators, as the sender of a transaction can not be altered unless you know their private key, it adds an additional security element for non-interactive proofs.

## 4.2 MAINNET

The data storage process is divided into the Mainnet and the StorkOracle. This section elaborates upon the Mainnet.

### 4.2.1 StorkTypes

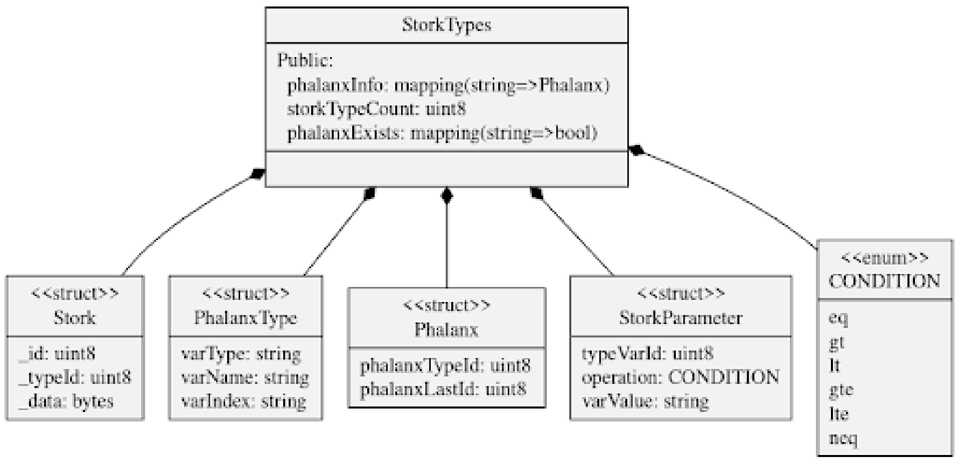


**Figure 4.1: StorkTypes**

StorkTypes describes the data types used in the custom StorkNet data. “Stork” is the fundamental unit of data in the StorkNet architecture and contains an _id, the dataType, and the data. The _id parameter is similar to mongoDB as in it is a unique identifier made by the network for a particular collection. A group of storks is called a “Phalanx”. A collection of Storks is similar to how MongoDB groups together data in a collection.

As a Phalanx is the collection of Storks of a particular kind, these Storks share common attributes that are called the Phalanx Metadata. These attributes are stored within the structures, “Phalanx” and “PhalanxType”. The former contains information about the size of the Phalanx and the Phalanx Id, which is useful in converting Solidity data-types into StorkNet compatible data-types called PhalanxType.

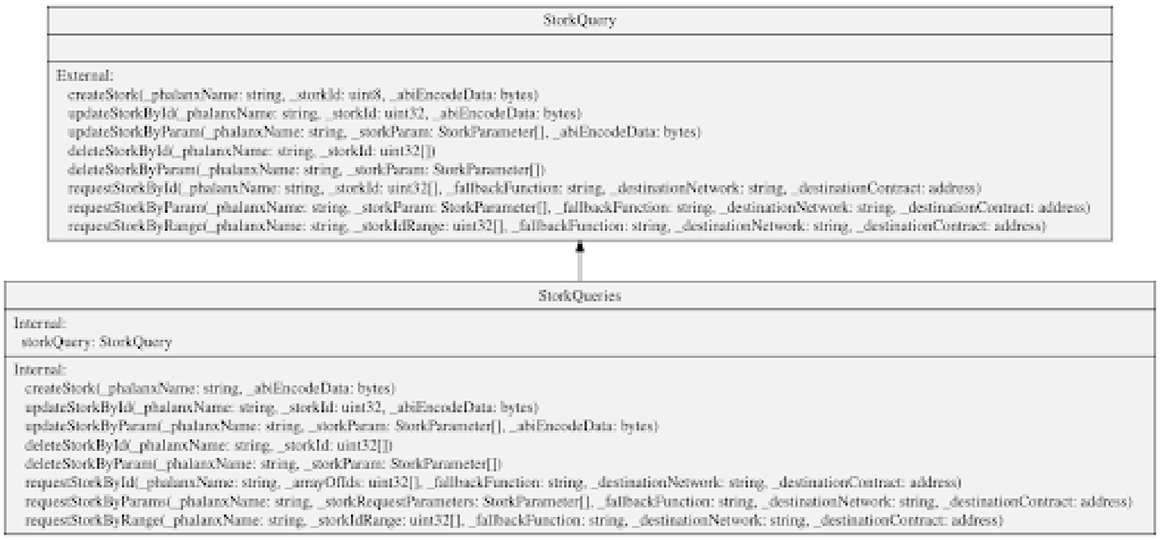
PhalanxType is the StorkNet compatible data-type that contains 3 strings, the first contains the Solidity variable type, the second the variable name in the Solidity smart contract, and the third is the index value, which is used in the case of arrays or mappings.

The reason for a custom data-type is that Solidity is a statically typed programming language that doesn’t have dynamic data-types. Therefore there is a need to create a custom “layer” that can convert between different types.

Moreover, as the data handling functions similar to a database, there are different operations that can be performed such as requesting, updating, deleting, or creating. These queries often require conditional evaluation such as equal to, greater than, lesser than or equal to, using the enum “CONDITIONS”. The conditional evaluation also benefits the users as it reduces the amount of data that needs to be transferred onto the main chain for execution, and as the amount of data transferred is lower, the cost of execution also dips.

In addition to these we have support functions such as assigning a name to each Phalanx, storing the number of Phalanxes created. The Figure 4.1 shows the StorkTypes structure.

### 4.2.2 StorkQueries



```
StorkQuery

External:
  createStork(_phalanxName: string, _storkId: uint8, _abiEncodeData: bytes)
  updateStorkById(_phalanxName: string, _storkId: uint32, _abiEncodeData: bytes)
  updateStorkByParam(_phalanxName: string, _storkParam: StorkParameter[], _abiEncodeData: bytes)
  deleteStorkById(_phalanxName: string, _storkId: uint32[])
  deleteStorkByParam(_phalanxName: string, _storkParam: StorkParameter[])
  requestStorkById(_phalanxName: string, _storkId: uint32[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)
  requestStorkByParam(_phalanxName: string, _storkParam: StorkParameter[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)
  requestStorkByRange(_phalanxName: string, _storkIdRange: uint32[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)

StorkQueries

Internal:
 storkQuery: StorkQuery

Internal:
  createStork(_phalanxName: string, _abiEncodeData: bytes)
  updateStorkById(_phalanxName: string, _storkId: uint32, _abiEncodeData: bytes)
  updateStorkByParam(_phalanxName: string, _storkParam: StorkParameter[], _abiEncodeData: bytes)
  deleteStorkById(_phalanxName: string, _storkId: uint32[])
  deleteStorkByParam(_phalanxName: string, _storkParam: StorkParameter[])
  requestStorkById(_phalanxName: string, _arrayOfIds: uint32[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)
  requestStorkByParams(_phalanxName: string, _storkRequestParameters: StorkParameter[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)
  requestStorkByRange(_phalanxName: string, _storkIdRange: uint32[], _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address)
```

**Figure 4.2: StorkQueries**

StorkQueries is essentially a contract that builds upon StorkTypes by using the Storks, StorkParameters, and Phalanxes to perform Query operations to off-load data onto the StorkNet. Each function listed in this contract has a companion Event that is used to emit the data off-chain so that it can be picked up by StorkNodes for processing.

They perform various operations that execute on StorkNet such as Creating Storks, Updating Storks by ID or Parameters, Deleting Storks by ID or Parameters. They also perform operations that send data to the Mainnet such as Requesting by ID, Parameters, or by ID Range.

The functionality of these nodes are very similar to The Graph, which is a Querying protocol for Ethereum, differing only in way they store data.The

Graph stores data on nodes that are javascript servers while StorkNet uses a blockchain, thus improving transparency and enforcing decentralization, another key difference is that the storage on The Graph is similar to IPFS, new IDs are created on every transaction.

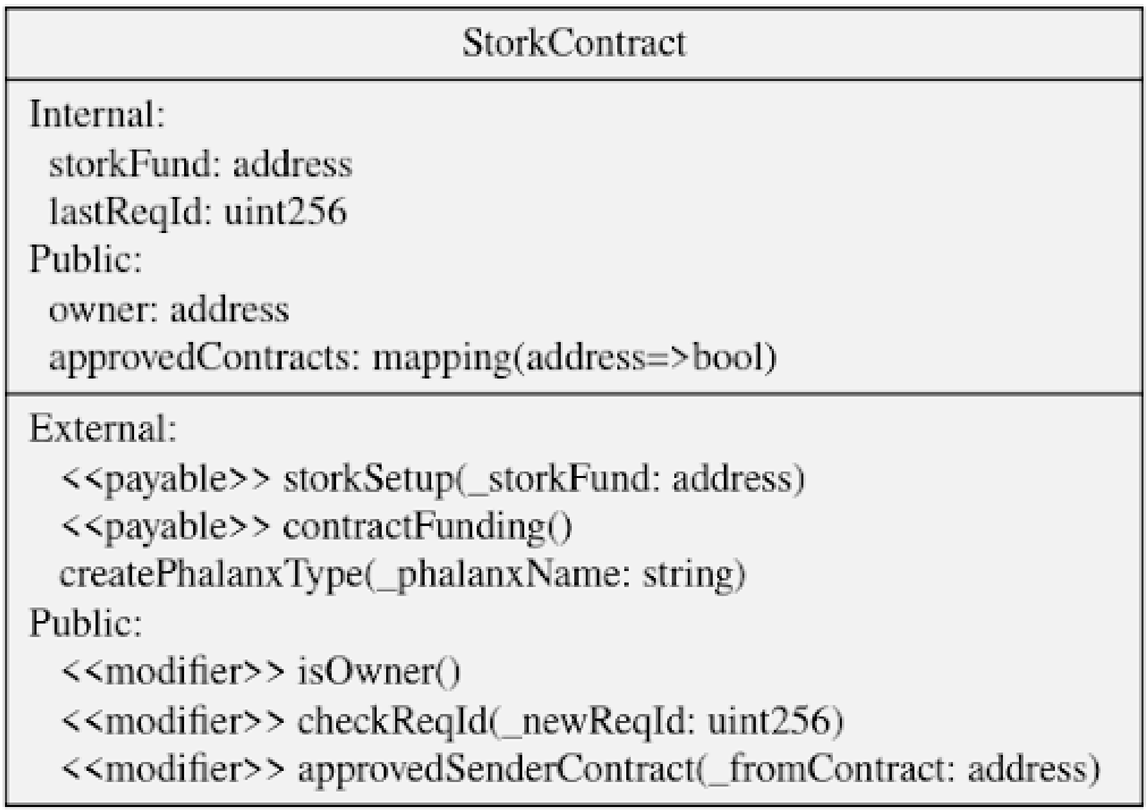
The “events” contain the PhalanxName, the StorkID or Parameter Array, the destination network name and the address of the smart contract on that network, and finally the Stork itself or the data being changed. As all data on EVMs are publicly accessible, even private data, exposing public data does not compromise the data itself.

The Request Queries contain a fallback function as the third parameter which is the name of the function that receives the requested data. The Figure 4.2 shows the StorkQueries structure.

### 4.2.3 StorkContract

The StorkContract further evolves on the StorkQueries and provides a Contract that can be extended by the Clients as seen in the Example Contract (explained in 4.2.9). This contract adds the Setup function that initiates the User’s contract by transferring some of the Native Tokens in order to “buy” transactions on the network.

It also sets up the other variables required for the protocol to run, such as the address of the StorkBlockRollup and the MultiSigVerification Contract. It also gets useful data such as the number of Txs left, the amount of cost per Tx, and the number of Txs left for the Contract. It also encodes and decodes the user created Solidity Data Types (structs, uint, int, bytes, strings, mappings, arrays, etc) into their StorkNet form. StorkDecode is used to split the data stored from the stork and return the _id parameter. The Figure 4.3 shows the StorkContract



```
StorkContract
Internal:
  storkFund: address
  lastReqId: uint256
Public:
  owner: address
  approvedContracts: mapping(address=>bool)
External:
   <<payable>> storkSetup(_storkFund: address)
   <<payable>> contractFunding()
   createPhalanxType(_phalanxName: string)
Public:
   <<modifier>> isOwner()
   <<modifier>> checkReqId(_newReqId: uint256)
   <<modifier>> approvedSenderContract(_fromContract: address)
```

**Figure 4.3: StorkContract**

structure.

### 4.2.4 StorkFund

StorkNet works on a pay-per-use model where the Clients fund their contracts with some gwei, (1000000 is the value of the smallest unit of currency in a EVM Chain). Using costPerTx we compute the number of transactions that a Client can be involved with. Once a client gets funded (from either themselves or externally) the number of transactions computed gets sent to the StorkBlockRollup contract so that it can maintain the latest computed Txs for a client.

Along with the number of Txs the amount of funds is also stored and

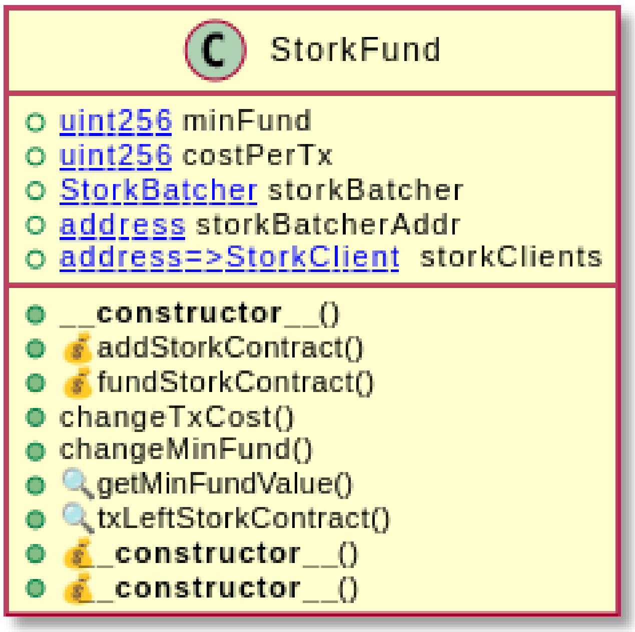


**Figure 4.4: StorkFund**

reduced overtime as more Txs execute, should the client choose to withdraw their funds, the funds are directly transferred. This "funds" variable is maintained because the Tx base cost can be changed in the future, which changes the cost of each transaction. The Client can be anyone, a StorkContract or a user address interacting with the Public IPFS Contracts, etc.

Finally, there are constructors called "receive" and "fallback" that are used for transferring the tokens between the contracts. The Figure 4.4 shows the structure of the StorkFund contract.

### 4.2.5 StorkStake

At the time of writing of this thesis (July 2022), the general consensus in the blockchain community is to migrate towards a PoS architecture, where the actors stake in a certain amount in order to get benefits from the network and this staked amount is then returned to them at the end of the time period with

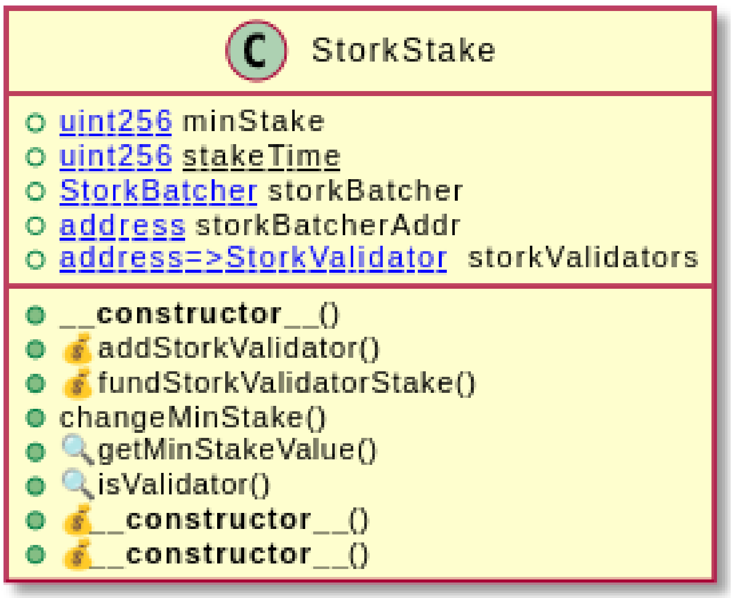


**Figure 4.5: StorkStake**

some rewards provided they behave like good actors and do not try to cheat or act out of line.

This PoS design creates an incentive among the Validators to behave correctly as there are penalties through fines that are imposed upon the validator by the network, should they be found guilty. It also fixes the Zero-stake problem wherein a Miner in a PoW has nothing to lose should they misbehave. A PoS system has greater upsides not only on the economic level by reducing the amount of resources required for computation, but also on the policial level as it allows for anyone to join the network and participate as a validator, making the network more decentralized.

This contract enables staking and by that it validates a StorkValidator, the stake value (in gwei) is then locked for a certain time period, and then the StorkBatcher contract is informed that a new address has been validated to be a StorkValidator. The validator can request for an early exit, before the full time period of the lock is reached and will be returned the staked amount + reward value with a percentage cut as penalty.

Finally, there are constructors called “receive” and “fallback” that are used for transferring the tokens between the contracts. The Figure 4.5 shows the StorkStake structure.

### 4.2.6 StorkBlockRollup

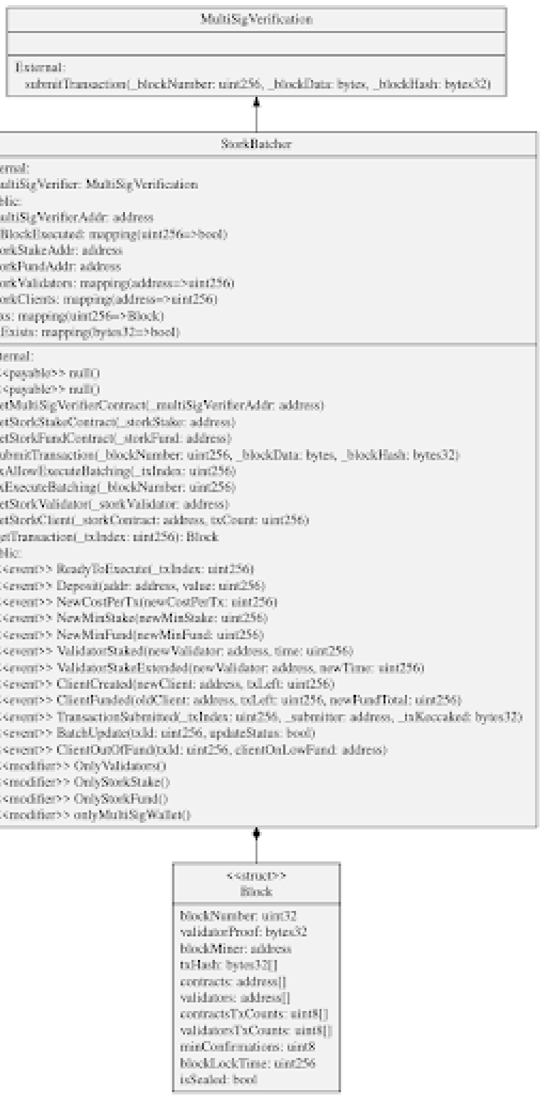

MultiSigVerification
External:
submitTransaction(_blockNumber: uint256, _blockData: bytes, _blockHash: bytes32)
StorkBatcher
Internal:
multiSigVerifier: MultiSigVerification
Public:
multiSigVerifierAddr: address
isBlockExecuted: mapping(uint256=>bool)
storkStakeAddr: address
storkFundAddr: address
storkValidators: mapping(address=>uint256)
storkClients: mapping(address=>uint256)
Txs: mapping(uint256=>Block)
txExists: mapping(bytes32=>bool)
External:
<<payable>> null()
<<payable>> null()
setMultiSigVerifierContract(_multiSigVerifierAddr: address)
setStorkStakeContract(_storkStake: address)
setStorkFundContract(_storkFund: address)
submitTransaction(_blockNumber: uint256, _blockData: bytes, _blockHash: bytes32)
txAllowExecuteBatching(_txIndex: uint256)
txExecuteBatching(_blockNumber: uint256)
setStorkValidator(_storkValidator: address)
setStorkClient(_storkContract: address, txCount: uint256)
getTransaction(_txIndex: uint256): Block
Public:
<<event>> ReadyToExecute(_txIndex: uint256)
<<event>> Deposit(addr: address, value: uint256)
<<event>> NewCostPerTx(newCostPerTx: uint256)
<<event>> NewMinStake(newMinStake: uint256)
<<event>> NewMinFund(newMinFund: uint256)
<<event>> ValidatorStaked(newValidator: address, time: uint256)
<<event>> ValidatorStakeExtended(newValidator: address, newTime: uint256)
<<event>> ClientCreated(newClient: address, txLeft: uint256)
<<event>> ClientFunded(oldClient: address, txLeft: uint256, newFundTotal: uint256)
<<event>> TransactionSubmitted(_txIndex: uint256, _submitter: address, _txKeccaked: bytes32)
<<event>> BatchUpdate(txId: uint256, updateStatus: bool)
<<event>> ClientOutOfFund(txId: uint256, clientOnLowFund: address)
<<modifier>> OnlyValidators()
<<modifier>> OnlyStorkStake()
<<modifier>> OnlyStorkFund()
<<modifier>> onlyMultiSigWallet()
<<struct>>
Block
blockNumber: uint32
validatorProof: bytes32
blockMiner: address
txHash: bytes32[]
contracts: address[]
validators: address[]
contractsTxCounts: uint8[]
validatorsTxCounts: uint8[]
minConfirmations: uint8
blockLockTime: uint256
isSealed: bool

**Figure 4.6: StorkBlockRollup**

StorkBlockRollup processes a StorkBlock that contains all the Txs

executed by the StorkNet. The miner (the validator chosen to create the block on mainnet) submits a ZK proof hash, which undergoes two more hashes, the first converts the array of hashes for each transaction into a hash
**txHashed = keccak256(txHasher[])**
And the second uses msg.sender to generate the hash proof
**keccak256(batchIndex,msg.sender,txHashed,cid)**

As the proof hash includes the msg.sender's details which is validated by the Ethereum network, it is unique for each address and hence the proof will not go through if any other address sends the transaction as it can't be spoofed on mainnet.

After the submission of a Tx, the StorkBlockContract calls the MSVC (explained in 4.2.7) that allows other validators to vote on the Block, if it passes then the Block moves on to the Execution stage where all the involved parties, the clients and the validators get updated.

Here the array of clients, an array of validators, and transactions performed are used in the TxHash Proof with the hashTx array to check if the data is maintained, and if the validator is the approved one. If they are, then the transaction goes ahead and executes them and the updates are made. The Figure 4.6 shows the StorkBlockRollup structure.

### 4.2.7 Multi Signature Verification Contract

Multi Signature Verification Contract is used to validate the incoming batches of Tx from StorkValidators so as to minimize transaction costs thus making the use of the protocol cheaper. The flow of a Tx is as follows. A selected Validator from the pool of Tx Validators "submits" a transaction, and the other validators "confirm" the transaction, the confirmation happens as below

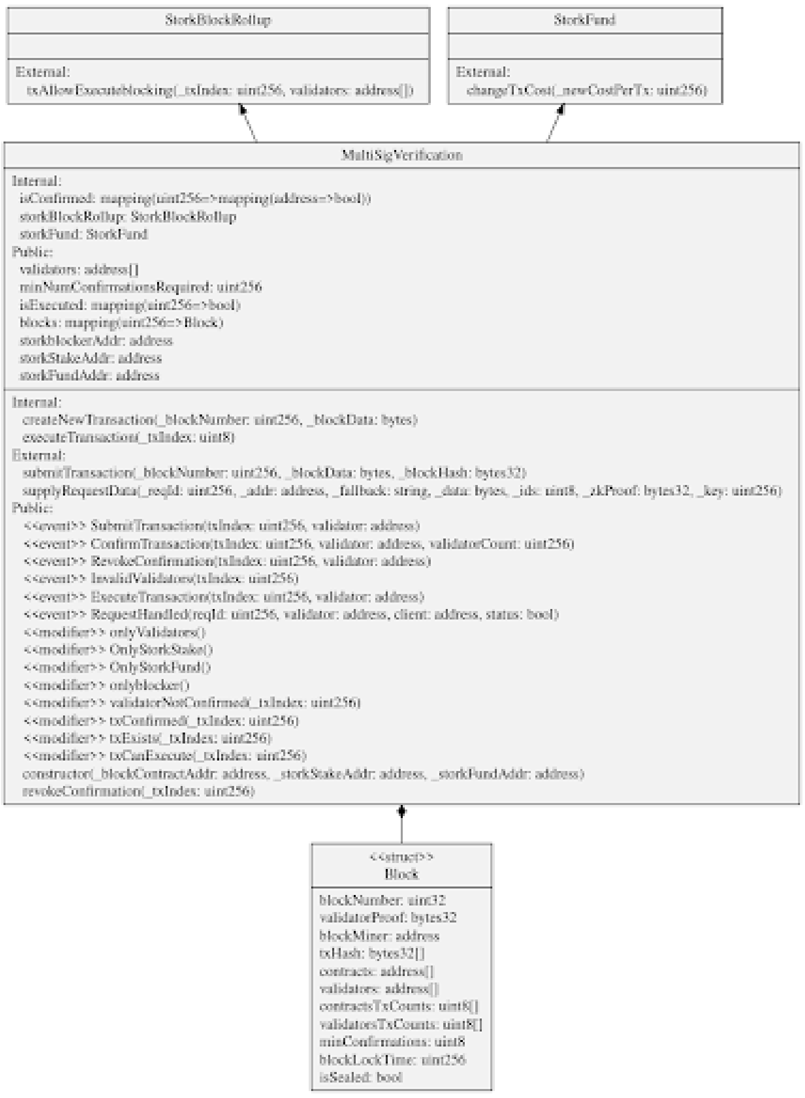

StorkBlockRollup
External:
txAllowExecuteblocking(_txIndex: uint256, validators: address[])
StorkFund
External:
changeTxCost(_newCostPerTx: uint256)
MultiSigVerification
Internal:
isConfirmed: mapping(uint256=>mapping(address=>bool))
storkBlockRollup: StorkBlockRollup
storkFund: StorkFund
Public:
validators: address[]
minNumConfirmationsRequired: uint256
isExecuted: mapping(uint256=>bool)
blocks: mapping(uint256=>Block)
storkblockerAddr: address
storkStakeAddr: address
storkFundAddr: address
Internal:
createNewTransaction(_blockNumber: uint256, _blockData: bytes)
executeTransaction(_txIndex: uint8)
External:
submitTransaction(_blockNumber: uint256, _blockData: bytes, _blockHash: bytes32)
supplyRequestData(_reqId: uint256, _addr: address, _fallback: string, _data: bytes, _ids: uint8, _zkProof: bytes32, _key: uint256)
Public:
<<event>> SubmitTransaction(txIndex: uint256, validator: address)
<<event>> ConfirmTransaction(txIndex: uint256, validator: address, validatorCount: uint256)
<<event>> RevokeConfirmation(txIndex: uint256, validator: address)
<<event>> InvalidValidators(txIndex: uint256)
<<event>> ExecuteTransaction(txIndex: uint256, validator: address)
<<event>> RequestHandled(reqId: uint256, validator: address, client: address, status: bool)
<<modifier>> onlyValidators()
<<modifier>> OnlyStorkStake()
<<modifier>> OnlyStorkFund()
<<modifier>> onlyblocker()
<<modifier>> validatorNotConfirmed(_txIndex: uint256)
<<modifier>> txConfirmed(_txIndex: uint256)
<<modifier>> txExists(_txIndex: uint256)
<<modifier>> txCanExecute(_txIndex: uint256)
constructor(_blockContractAddr: address, _storkStakeAddr: address, _storkFundAddr: address)
revokeConfirmation(_txIndex: uint256)
<<struct>>
Block
blockNumber: uint32
validatorProof: bytes32
blockMiner: address
txHash: bytes32[]
contracts: address[]
validators: address[]
contractsTxCounts: uint8[]
validatorsTxCounts: uint8[]
minConfirmations: uint8
blockLockTime: uint256
isSealed: bool

**Figure 4.7: Multi Signature Verification Contract**

$\sum_{i=0}^{validators} ValidatorChallengeXORaddress(validator_i)$
until it reduces to
$\sum_{i=0}^{validators} address(validator_i) XORaddress(validator_i) = address(0)$

Once it gets the minimum required confirmations (i.e. a 0 in the challenge), the StorkBlockRollup gets updated to allow executions on that batch Tx.

There is also a revoke option should StorkNet detect before the execution of the Tx that for some reason it should be recalled, any of the validators re-enters the ValidatorChallenge function and changes the state, essentially undoing the previous confirmation.

$$
\begin{gathered}
ValidatorChallenge_{state_i} = ValidatorChallengeXORaddress(validator_i) \\
ValidatorChallenge_{state(i+1)} = \\
ValidatorChallenge_{state(i)}XORaddress(validator_i) = \\
ValidatorChallengeXORaddress(validator_i)XORaddress(validator_i) \\
address(validator_i)XORaddress(validator_i) = address(0) \\
ValidatorChallenge_{state(i+1)} = ValidatorChallenge_{state(i)}XORaddress(0) \\
ValidatorChallenge_{state(i+1)} = ValidatorChallenge_{state(i)}
\end{gathered}
$$

The number of Txs handled by the Validators is also sent to the StorkBlockRollup so as to compute their rewards for being a part of StorkNet.

This contract also receives the StorkData that was requested by a StorkContract from the elected Miner to handle the request. It verifies the challenge produced by that miner and if successful submits the data to the StorkContract. The Figure 4.7 shows the structure of Multi Signature Verification Contract.

### 4.2.8 Stork Validators

They are the ones responsible for handling all the data flows. On the mainnet side they listen to all the events on the network for queries to operate on. These validators are split into “shards” where each shard is in charge of a group of a specific network. They listen to events and also interact with the Multi Sig Verification Contract and process Txs.

### 4.2.9 Example Contract (Student Grades)

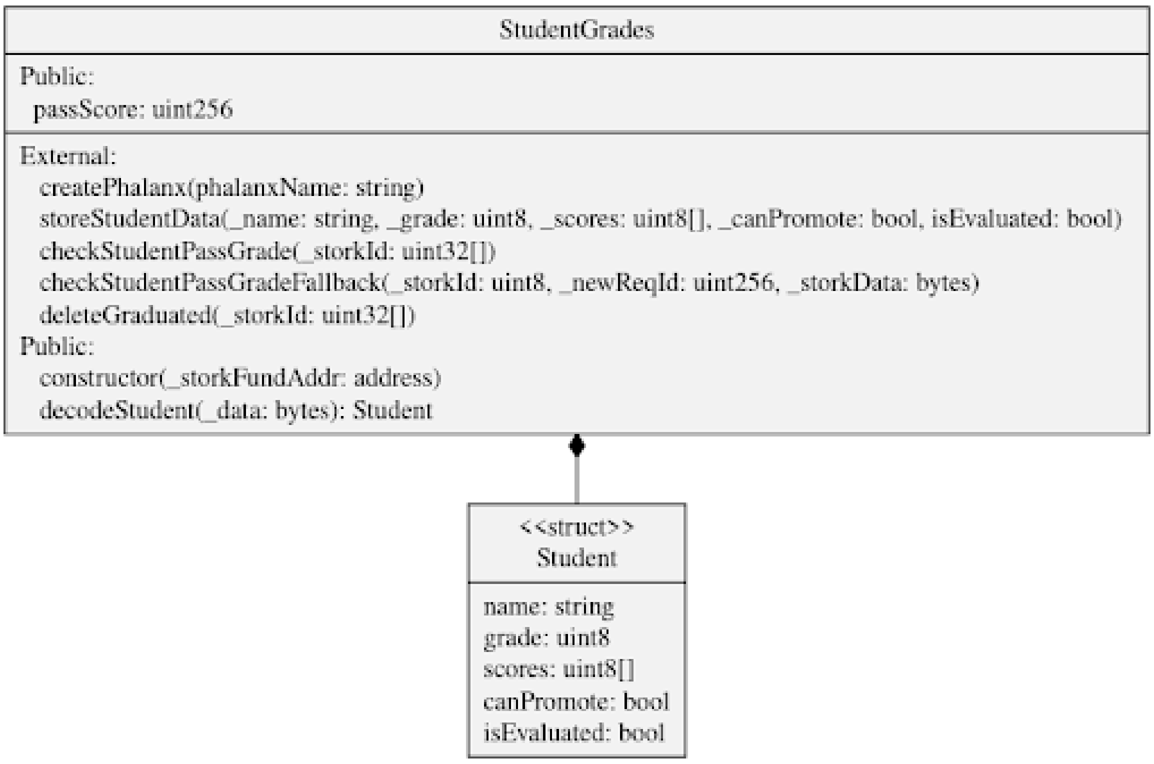

StudentGrades
Public:
passScore: uint256
External:
createPhalanx(phalanxName: string)
storeStudentData(_name: string, _grade: uint8, _scores: uint8[], _canPromote: bool, isEvaluated: bool)
checkStudentPassGrade(_storkId: uint32[])
checkStudentPassGradeFallback(_storkId: uint8, _newReqId: uint256, _storkData: bytes)
deleteGraduated(_storkId: uint32[])
Public:
constructor(_storkFundAddr: address)
decodeStudent(_data: bytes): Student
<<struct>>
Student
name: string
grade: uint8
scores: uint8[]
canPromote: bool
isEvaluated: bool

**Figure 4.8: Example Contract: Student Grades**

This is a contract made out of the StorkContract, and contracts made from StorkContract are simply called StorkContracts. The example here is to store the data about a student such as the student's name, grade, test scores, hasPassed, and isEvaluated.

When the storeStudentData function is called with an object of struct Student, it calls the CreateStork function that emits the event. The StorkValidators pick this event up and adds it to StorkNet's mempool of Txs. There is also a function called checkStudentPassGrade() that takes in an _id and Requests for Stork, with a fallback function named checkStudentPassGradeFallback(). This request event is picked up by the Validators and performs the Multi Sig Verification, when that completes the Validator that is chosen to submit is given the rights to send the data to the contract. The Validator sends the data to the

Multi Sig Verification that forwards the data to the fallback function to the StorkContract that sent the request. The StorkContract checks with the Multi Signature Verification Contract if the sender is the approved one, and if so gets the data and then checks if the sum of the test scores is greater than the passScore.

If the score is greater the hasPassed attribute is set to true else false. In either case the isEvaluated attribute is set to true, and then the Update Query is emitted on the data. The final Query showcased is the Delete Query where it sends a Delete By Parameter Query to delete all students that have passed and are currently in 10th grade. The Figure 4.8 shows the example contract structure.

## 4.3 STORKCHAIN

The StorkChain has various components that make up the L-2 storage. It not only stores the data but also performs data integrity tests, consensus, and other core blockchain implementations making it a virtual blockchain ontop of a L-2 chain

### 4.3.1 StorkValidator

StorkValidator allows StorkContracts to interact with the StorkNet, by listening to events emitted by the StorkQuery contract. Once it obtains a new Query it sends the content of the event to the StorkBlockGenerator where a voting process happens by all StorkValidators based on the validity of the Tx. After this the StorkBlockGenerator takes control of the executions and gets back to the StorkValidators after the block is produced. Here a single validator is elected as the miner, and few more as validators. These specially designated

StorkValidators then interact with the StorkBatcher and MultiSig Verification Contract.

### 4.3.2 StorkBlock

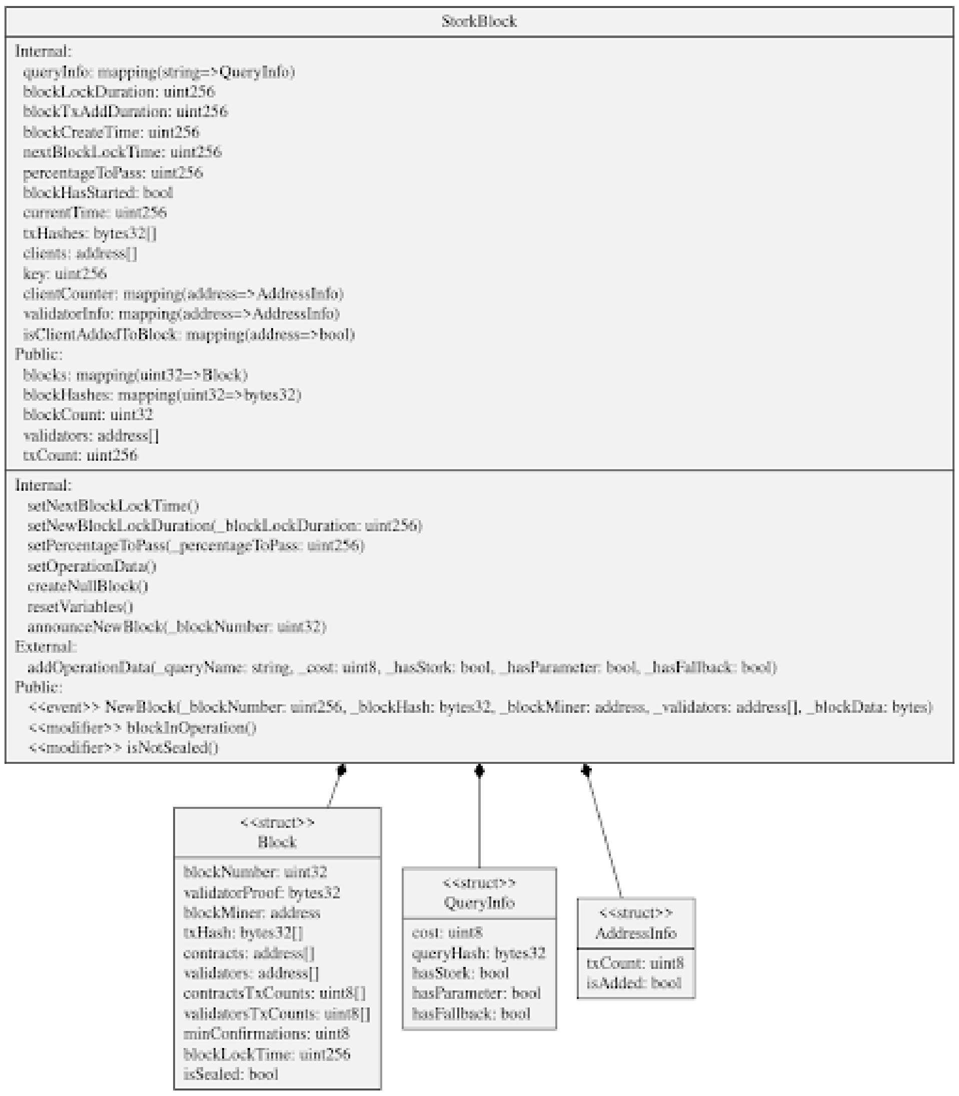

StorkBlock
Internal:
queryInfo: mapping(string=>QueryInfo)
blockLockDuration: uint256
blockTxAddDuration: uint256
blockCreateTime: uint256
nextBlockLockTime: uint256
percentageToPass: uint256
blockHasStarted: bool
currentTime: uint256
txHashes: bytes32[]
clients: address[]
key: uint256
clientCounter: mapping(address=>AddressInfo)
validatorInfo: mapping(address=>AddressInfo)
isClientAddedToBlock: mapping(address=>bool)
Public:
blocks: mapping(uint32=>Block)
blockHashes: mapping(uint32=>bytes32)
blockCount: uint32
validators: address[]
txCount: uint256
Internal:
setNextBlockLockTime()
setNewBlockLockDuration(_blockLockDuration: uint256)
setPercentageToPass(_percentageToPass: uint256)
setOperationData()
createNullBlock()
resetVariables()
announceNewBlock(_blockNumber: uint32)
External:
addOperationData(_queryName: string, _cost: uint8, _hasStork: bool, _hasParameter: bool, _hasFallback: bool)
Public:
<<event>> NewBlock(_blockNumber: uint256, _blockHash: bytes32, _blockMiner: address, _validators: address[], _blockData: bytes)
<<modifier>> blockInOperation()
<<modifier>> isNotSealed()
<<struct>>
Block
blockNumber: uint32
validatorProof: bytes32
blockMiner: address
txHash: bytes32[]
contracts: address[]
validators: address[]
contractsTxCounts: uint8[]
validatorsTxCounts: uint8[]
minConfirmations: uint8
blockLockTime: uint256
isSealed: bool
<<struct>>
QueryInfo
cost: uint8
queryHash: bytes32
hasStork: bool
hasParameter: bool
hasFallback: bool
<<struct>>
AddressInfo
txCount: uint8
isAdded: bool

**Figure 4.9: StorkBlock**

This contract contains details about the “Blocks” that are generated by the network along with the information about the Queries. The Blocks are chained together by their parent’s block hash. The lifespan of a block consists of two phases: the ProposeTx phase and the Execution phase. This is so that we can control the size of the blocks and the transactions included in them.

The Create, Update, and Delete operations are not dependent on the time required for the operation to execute but they do depend on the sequence in which they occur. The Queries are sequenced in the order in which the transactions of the queries are received as the Sequence only matters from the local perspective of each Contract and not the overall global sequence of all contracts. It is these blocks that get processed and finally emitted which is then picked up by the validators and then submitted to mainnet for proof. The Figure 4.9 shows the StorkBlock structure.

### 4.3.3 StorkBlockGenerator

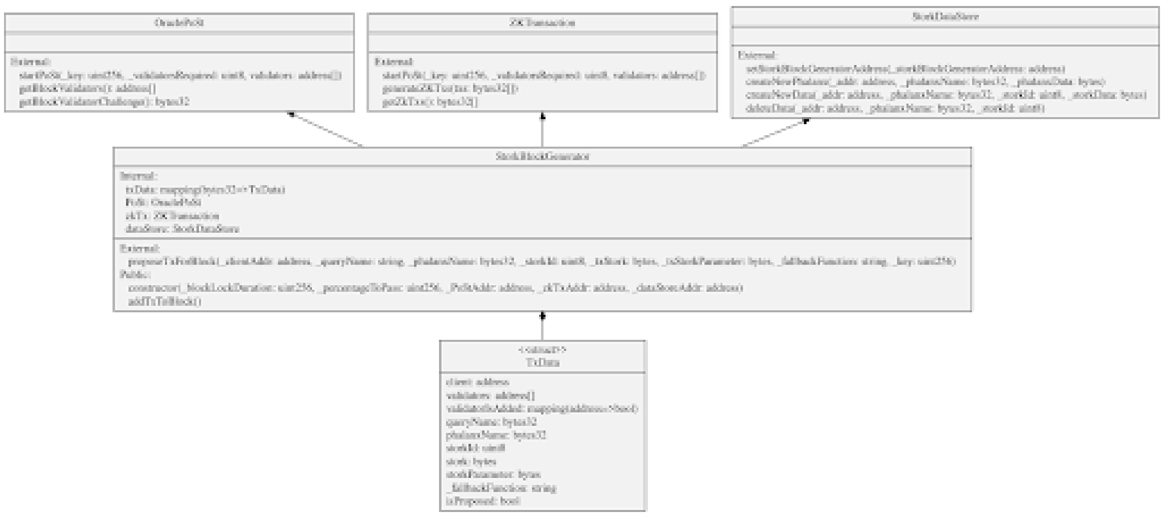

OraclePoSt
External:
startPoSt(_key: uint256, _validatorsRequired: uint8, validators: address[])
getBlockValidators(): address[]
getBlockValidatorChallenge(): bytes32
ZKTransaction
External:
startPoSt(_key: uint256, _validatorsRequired: uint8, validators: address[])
generateZKTxs(txs: bytes32[])
getZkTxs(): bytes32[]
StorkDataStore
External:
setStorkBlockGeneratorAddress(_storkBlockGeneratorAddress: address)
createNewPhalanx(_addr: address, _phalanxName: bytes32, _phalanxData: bytes)
createNewData(_addr: address, _phalanxName: bytes32, _storkId: uint8, _storkData: bytes)
deleteData(_addr: address, _phalanxName: bytes32, _storkId: uint8)
StorkBlockGenerator
Internal:
txData: mapping(bytes32=>TxData)
PoSt: OraclePoSt
zkTx: ZKTransaction
dataStore: StorkDataStore
External:
proposeTxForBlock(_clientAddr: address, _queryName: string, _phalanxName: bytes32, _storkId: uint8, _txStork: bytes, _txStorkParameter: bytes, _fallbackFunction: string, _key: uint256)
Public:
constructor(_blockLockDuration: uint256, _percentageToPass: uint256, _PoStAddr: address, _zkTxAddr: address, _dataStoreAddr: address)
addTxToBlock()
<<struct>>
TxData
client: address
validators: address[]
validatorIsAdded: mapping(address=>bool)
queryName: bytes32
phalanxName: bytes32
storkId: uint8
stork: bytes
storkParameter: bytes
_fallbackFunction: string
isProposed: bool

**Figure 4.10: StorkBlockGenerator**

As StorkNet is a virtual blockchain, i.e. a blockchain built upon another one, in this case being Ethereum. This contract creates a "Block" on StorkNet, which is analogous to the blocks on Ethereum wherein a block contains several transactions. The functionality is similar to how PoS works, we have validators uploading newer transactions and voting on their legitimacy, after a minimum number of transaction confirmations are reached the Tx is added to the block and then "sealed". When it is sealed the only things that can be added are the miner's address, the validators, and the proofs.

These votes on transactions are obtained by computing a transaction's hash and comparing the number of times the same hash gets generated by various validators. After the ProposeTx phase has timed out the votes per transaction is then compared with the minimum number of votes to pass, which is a fraction of the total number of validators in the network proposing transactions. If a transaction passes the test it is then executed and packaged into the block. The Figure 4.10 shows the StorkBlockGenerator structure.

### 4.3.4 PoSt + PoH

This contract provides a fair playfield for all the miners seeking work in the network. The inputs are the miner's address and their transaction history.

There are n groups based on the required depth=n; the first (G1) consists of the miners with the largest stake, the second(G2) with a lesser amount of stake, up to (Gn). For a PoSt with depth=n, we have a PoSt vector that consists of a left-sided diagonal vector of dimensions (n x n). The elements of each row are generated cyclically based on the range defined by (n - row number) to (n). This vector is then multiplied with the number of miners (PoStMiners) in every group as a (n x 1) dimensional standing vector which generates the PoSt Elements at a Level L, which is used to generate a fair probability distribution that dynamically changes size.

Some modifiers allow different groups to have an edge on each Layer L. We construct a graph of depth of n made up of L1, L2, ... Ln. The top job (L1) requires only G1 members, so they have a stake-scaling factor of n. The

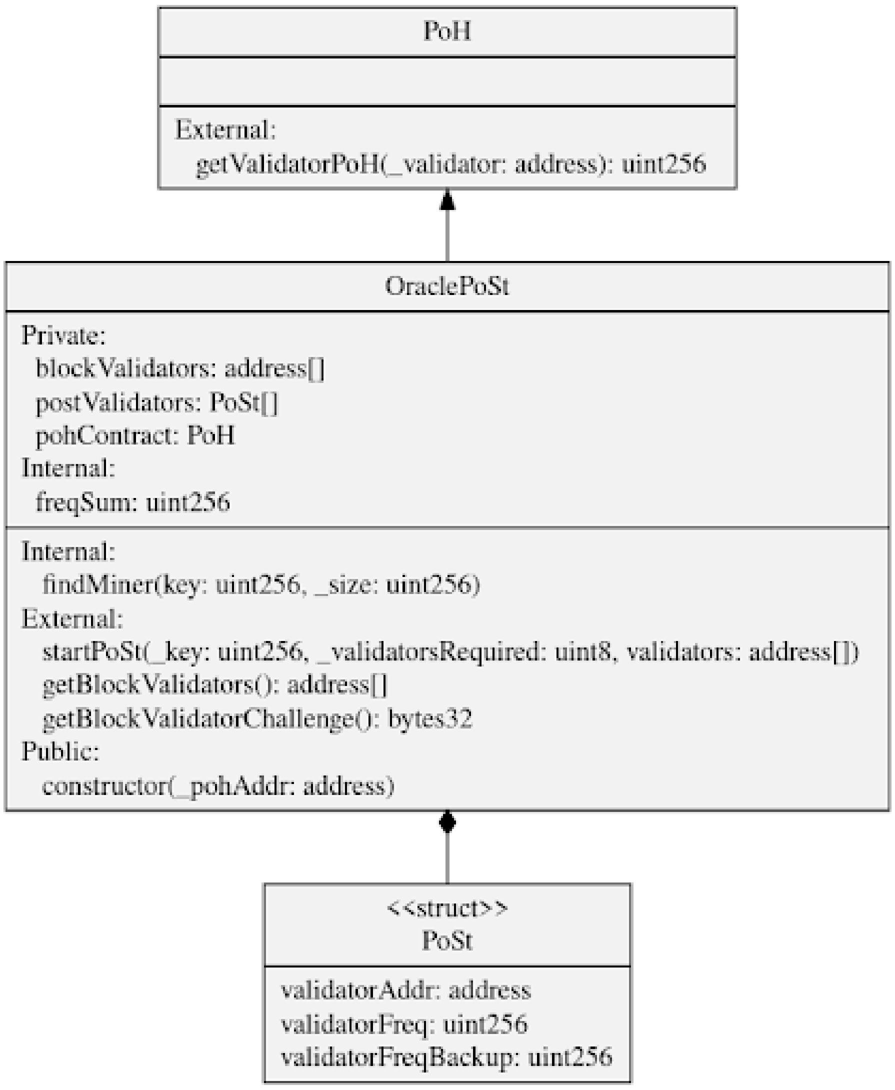

PoH
External:
getValidatorPoH(_validator: address): uint256
OraclePoSt
Private:
blockValidators: address[]
postValidators: PoSt[]
pohContract: PoH
Internal:
freqSum: uint256
Internal:
findMiner(key: uint256, _size: uint256)
External:
startPoSt(_key: uint256, _validatorsRequired: uint8, validators: address[])
getBlockValidators(): address[]
getBlockValidatorChallenge(): bytes32
Public:
constructor(_pohAddr: address)
<<struct>>
PoSt
validatorAddr: address
validatorFreq: uint256
validatorFreqBackup: uint256

**Figure 4.11: POST + POH**

second tier (L2) job requires G1 and G2 members; here, G1 members chosen for the topmost are not included for G2, so their stakeScore is set to 0. G1 members here have a stake-scaling factor of (n-1), while G2 members have a stake-scaling factor of (n). In (Ln), the G1 members have a scaling factor of (n- (n-1) ) while G2 have (n-(n-2) ) and Gn has n.

The standing vector provides us with the number of miners at Lx, and we perform the modulo operation of the nonce with the standing vector's value for Lx provides us with the initial validator for the given Lx, upon selection of a validator as we remove them from that particular staking the standing vector's value gets updated. More validators are selected by repeating the above process until the desired number of validators is obtained.

$$PoSt = \begin{bmatrix} 3 & 0 & 0 \\ 2 & 3 & 0 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix}$$

$$PoStMiner = \begin{bmatrix} 3 \\ 4 \\ 4 \end{bmatrix} = \begin{bmatrix} G1 \\ G2 \\ G3 \end{bmatrix}$$

$$PoSt^T = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

$$PoStElements(L) = PoSt.PoStMiner = \begin{bmatrix} 9 \\ 18 \\ 23 \end{bmatrix}$$

$$PoStProbability(L) = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} \frac{Gn}{9} & 0 & 0 \\ 0 & \frac{Gn}{18} & 0 \\ 0 & 0 & \frac{Gn}{23} \end{bmatrix}$$

$$PoStProbability(L) = \begin{bmatrix} \frac{3*3}{9} & \frac{2*3}{18} & \frac{1*3}{23} \\ 0 & \frac{3*4}{18} & \frac{2*4}{23} \\ 0 & 0 & \frac{3*4}{23} \end{bmatrix}$$

$$PoStProbability = \begin{bmatrix} \frac{9}{9} & \frac{6}{18} & \frac{3}{23} \\ 0 & \frac{12}{18} & \frac{8}{23} \\ 0 & 0 & \frac{12}{23} \end{bmatrix}$$

$$P(G1) = \frac{(\frac{9}{9} + \frac{6}{18} + \frac{3}{23})}{3} = 0.4879$$
$$P(G2) = \frac{(\frac{0}{9} + \frac{12}{18} + \frac{8}{23})}{3} = 0.3381$$
$$P(G3) = \frac{(\frac{0}{9} + \frac{0}{18} + \frac{12}{23})}{3} = 0.1740$$
$$P(G) = P(G1) + P(G2) + P(G3) = 1.00$$

After this from the set of validators, a miner is generated using the same algorithm. This also allows us to secretly select the miner as the input is an array of addresses from which one is randomly generated, by a process called the minerChallenge which can be executed to return true only by the miner and no one else. After that the same algorithm is run through to link a group of Txs with the validators that were elected. The Figure 4.11 shows the POST + POH structure.

### 4.3.5 ZK Transaction

Generates two zk-SNARK, that prove the validator list called the validatorChallenge and the blockHash that contains the metadata of the block with the miner's address. These generators are used by the mainnet smart contracts StorkBlockRollup and MSVC to verify the proofs whenever a new Block is submitted to them. They maintain the integrity of the Block and also provide access control for them. These are then added to the block. The Figure

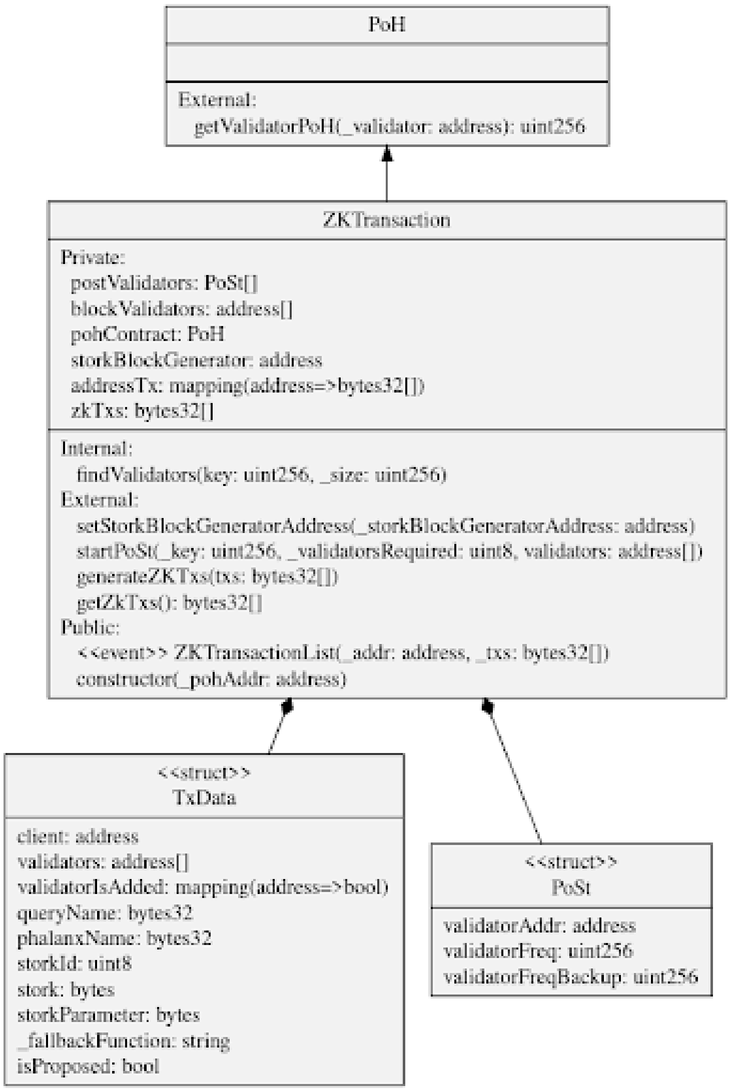

PoH
External:
getValidatorPoH(_validator: address): uint256
ZKTransaction
Private:
postValidators: PoSt[]
blockValidators: address[]
pohContract: PoH
storkBlockGenerator: address
addressTx: mapping(address=>bytes32[])
zkTxs: bytes32[]
Internal:
findValidators(key: uint256, _size: uint256)
External:
setStorkBlockGeneratorAddress(_storkBlockGeneratorAddress: address)
startPoSt(_key: uint256, _validatorsRequired: uint8, validators: address[])
generateZKTxs(txs: bytes32[])
getZkTxs(): bytes32[]
Public:
<<event>> ZKTransactionList(_addr: address, _txs: bytes32[])
constructor(_pohAddr: address)
<<struct>>
TxData
client: address
validators: address[]
validatorIsAdded: mapping(address=>bool)
queryName: bytes32
phalanxName: bytes32
storkId: uint8
stork: bytes
storkParameter: bytes
_fallbackFunction: string
isProposed: bool
<<struct>>
PoSt
validatorAddr: address
validatorFreq: uint256
validatorFreqBackup: uint256

**Figure 4.12: ZK Transaction**

4.12 shows the ZK Transaction structure.

$$\forall validators \in addr = address(validator)$$

$$Challenge : validatorChallenge \cdot \prod_{i=0}^{i=n-1} keccak256(addr_i) \wedge keccak256(addr_{i+1})$$

$$\text{Proof} : \prod_{i=0}^{i=n-1} (keccak256(validatorChallenge, msg.sender)))$$

This upgrades the Tx hash array by looping them through the hash circuit and including the validator as a parameter for each Tx hash. The newly generated TxHash array is then used to replace the TxHash in the block. This allows for verification by the mainnet contracts about the data being submitted and the submitter of the data, aka the validator.

Challenge : keccak256($Tx_i, validator_i$)
Proof : $(keccak256(Tx_{rawdata}, msg.sender)))$

### 4.3.6 RequestHandler

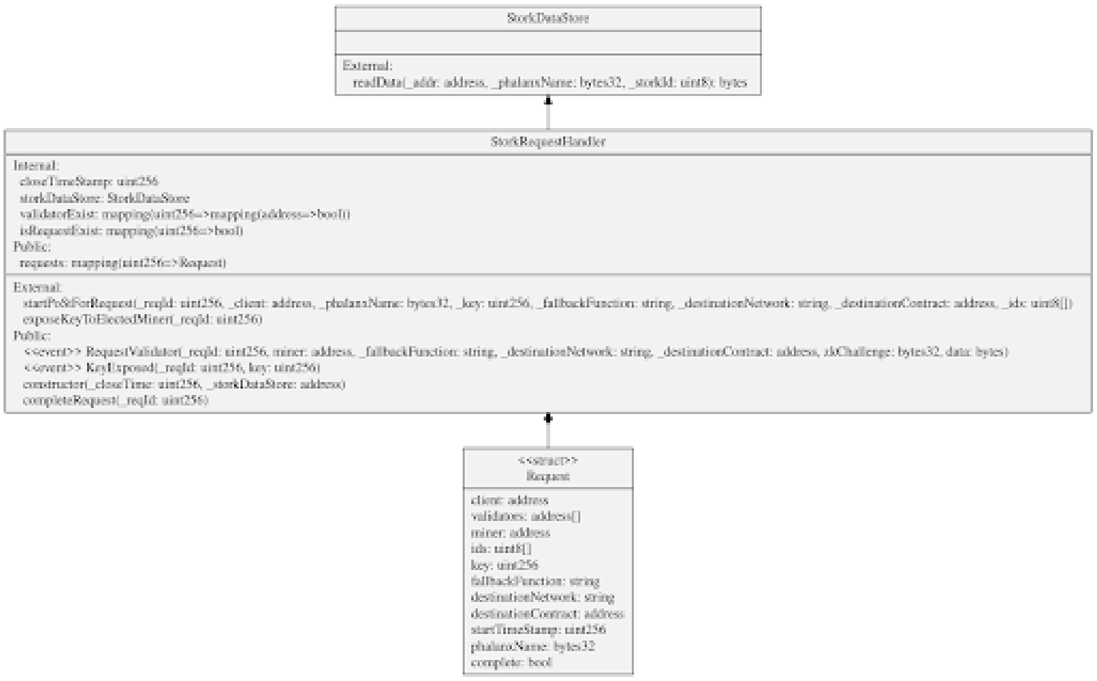
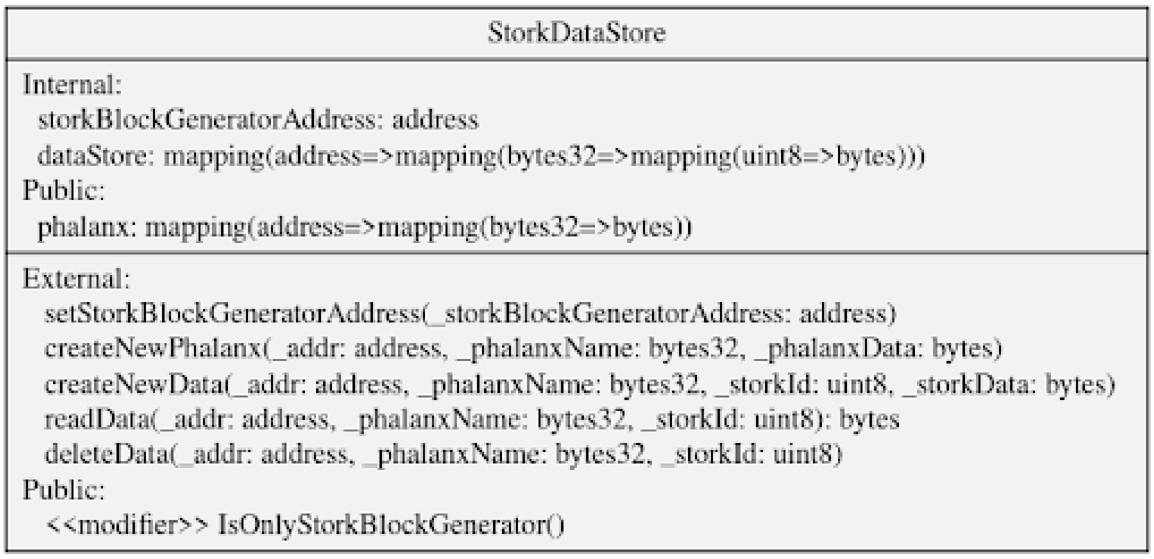

StorkDataStore
External:
readData(_addr: address, _phalanxName: bytes32, _storkId: uint8): bytes
StorkRequestHandler
Internal:
closeTimeStamp: uint256
storkDataStore: StorkDataStore
validatorExist: mapping(uint256=>mapping(address=>bool))
isRequestExist: mapping(uint256=>bool)
Public:
requests: mapping(uint256=>Request)
External:
startPoStForRequest(_reqId: uint256, _client: address, _phalanxName: bytes32, _key: uint256, _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address, _ids: uint8[])
exposeKeyToElectedMiner(_reqId: uint256)
Public:
<<event>> RequestValidator(_reqId: uint256, miner: address, _fallbackFunction: string, _destinationNetwork: string, _destinationContract: address, zkChallenge: bytes32, data: bytes)
<<event>> KeyExposed(_reqId: uint256, key: uint256)
constructor(_closeTime: uint256, _storkDataStore: address)
completeRequest(_reqId: uint256)
<<struct>>
Request
client: address
validators: address[]
miner: address
ids: uint8[]
key: uint256
fallbackFunction: string
destinationNetwork: string
destinationContract: address
startTimeStamp: uint256
phalanxName: bytes32
complete: bool

**Figure 4.13: Request Handler**

Functions as a minimalist version of PoSt and ZKTransaction. This is required as the time taken to generate a block is much larger than the time a contract can spend waiting for data, not that there is a time constraint set by the blockchain but rather speed at which executions occur is important and can't be made too slow when compared on traditional smart contracts where data is stored on chain.

The contract opens a time period in which it accepts requests and once that's complete it finds out the miner and emits an event letting them know that they can get the data from DataStore by proving their identity. The Figure 4.13 shows the Request Handler structure.

### 4.3.7 StorkDataStore



```
StorkDataStore

Internal:
  storkBlockGeneratorAddress: address
  dataStore: mapping(address=>mapping(bytes32=>mapping(uint8=>bytes)))
Public:
  phalanx: mapping(address=>mapping(bytes32=>bytes))

External:
   setStorkBlockGeneratorAddress(_storkBlockGeneratorAddress: address)
   createNewPhalanx(_addr: address, _phalanxName: bytes32, _phalanxData: bytes)
   createNewData(_addr: address, _phalanxName: bytes32, _storkId: uint8, _storkData: bytes)
   readData(_addr: address, _phalanxName: bytes32, _storkId: uint8): bytes
   deleteData(_addr: address, _phalanxName: bytes32, _storkId: uint8)
Public:
   <<modifier>> IsOnlyStorkBlockGenerator()
```

**Figure 4.14: StorkDataStorage**

This is where the data is stored in mappings that links Storks with the network the contract is on with the address of a client and then the storkId.Data can be written into this contract only through the BlockGenerator contract, and reading requires verification of the validator. The Figure 4.14 shows the StorkDataStore structure.

# CHAPTER 5

# RESULTS AND PERFORMANCE ANALYSIS

This chapter provides the results and performance analysis of the entire project.

## 5.1 BLOCKCHAIN

| | Ease of Use | Blockchain | Consensus | Modifiable Data |
|---|---|---|---|---|
| StorkNet | Join By Staking | Yes | PoSt + PoH | Yes |
| Filecoin | Contract Sign | Yes | NA | NA |
| IPFS | Very High | NA | NA | NA |

**Table 5.1: Property Table 1**

| | Decentralized | Payment | Queryable | Data Controlled by |
|---|---|---|---|---|
| StorkNet | Pure | Native Token | Yes | Creator |
| Filecoin | Partial | Yes, own wallet | NA | Creator |
| IPFS | Pure | NA | NA | Network |

**Table 5.2: Property Table 2**

The Tables 5.1 and 5.2 show the comparison of different properties with respect to three different file storage mechanisms

- The StorkNet protocol achieves true blockchain decentralization as the entire protocol is an L-2 chain unlike the IPFS and FileCoin.
- StorkNet exists as a library on the mainnet that other contracts can import and use the protocol. In IPFS and FileCoin, it's up to the user to resend the data every time thereby not allowing for automation of jobs.

- StorkNet also doubles down as a way to allow for gas fee reduction which is also seen in IPFS and FileCoin.
- The three protocols also allow for cross-chain data transfers.
- The queryable feature allows for database like transactions such as Request data, Update data, Delete data, and Create data, this allows for data modifications.
- As StorkNet runs on a blockchain it has its own consensus mechanism which is not seen in IPFS or FileCoin.

The following are the results comparing the use of a StorkContract against a traditional smart contract. Do note that the StorkContract contains all the functions provided and can be slimmed down to match the exact use case of the user, which is how it should be done. This would further reduce the gas costs due to reduced code.

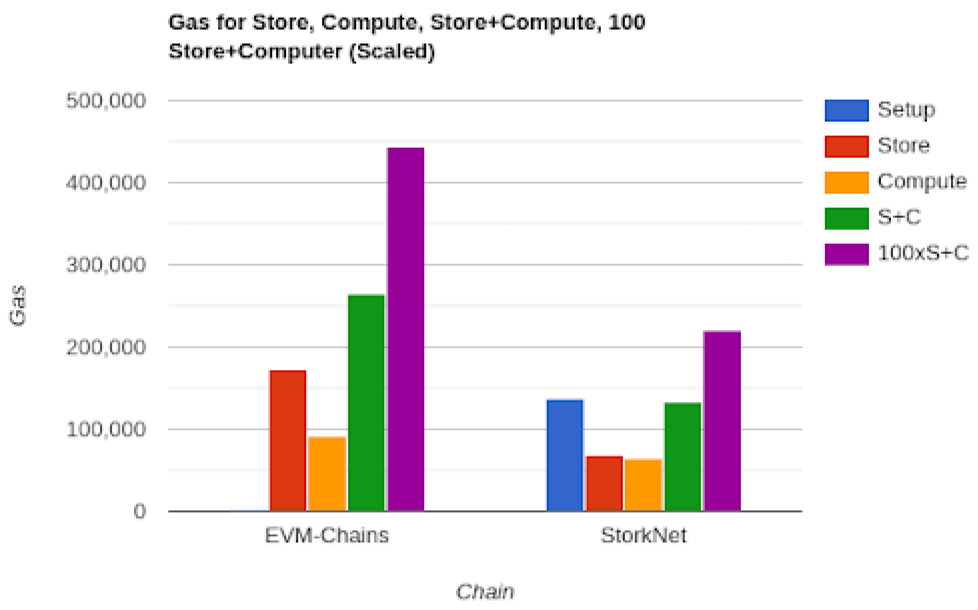


**Figure 5.1: Graph Comparison of Gas Costs**

The StorkNet protocol has a "setup fee" which is essentially just the extra functions being added to the contract through the StorkContract library. However after that the cost to Store and Compute drops significantly.

The final metric, the 100xS+C has been scaled down by a factor of 100. This graph shows to us that even if the user does not intend to use StorkNet for cross-chain data transfers, they can still use it for gas savings.

As StorkNet has an initial cost for smart contract deployment we plotted this chart to see at what number of transactions would the use of StorkNet be cheaper. It is clearly visible from Figure 5.1 that at 0 Txs (i.e. at deployment and setup) StorkNet costs much more than EVM chains. Therefore there is a minimum number of transactions that is required to make the network profitable. We can also note that the growth in amount of gas saved is very large, as seen at 100 Transactions.

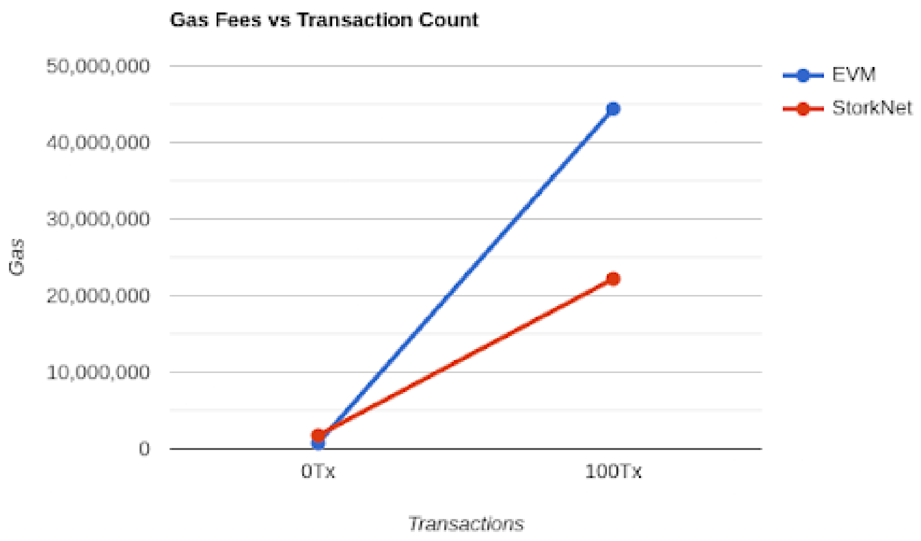
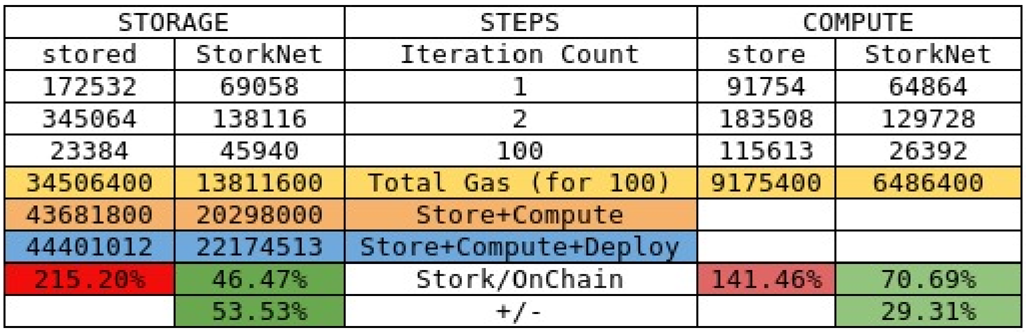


**Figure 5.2: Line Chart Comparing the Gas Cost vs Transaction Count**

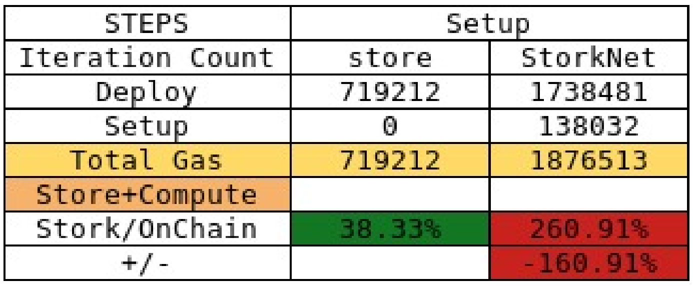
In the full breakdown of the gas costs we see that deploying and setting up a StorkContract is about 2.6x that of a regular contract from the Figure 5.2.



| STORAGE | | STEPS | COMPUTE | |
|---|---|---|---|---|
| stored | StorkNet | Iteration Count | store | StorkNet |
| 172532 | 69058 | 1 | 91754 | 64864 |
| 345064 | 138116 | 2 | 183508 | 129728 |
| 23384 | 45940 | 100 | 115613 | 26392 |
| 34506400 | 13811600 | Total Gas (for 100) | 9175400 | 6486400 |
| 43681800 | 20298000 | Store+Compute | | |
| 44401012 | 22174513 | Store+Compute+Deploy | | |
| 215.20% | 46.47% | Stork/OnChain | 141.46% | 70.69% |
| | 53.53% | +/- | | 29.31% |

**Figure 5.3: Storage Cost Comparison**

We also note that compute cost is about 30% lesser per transaction and the cost of storage is about 60% lower. When extended to performing 100 Transactions the overall savings (while including the setup costs) is about 53.53% from the Figure 5.3.



| STEPS | Setup | |
|---|---|---|
| Iteration Count | store | StorkNet |
| Deploy | 719212 | 1738481 |
| Setup | 0 | 138032 |
| Total Gas | 719212 | 1876513 |
| Store+Compute | | |
| Stork/OnChain | 38.33% | 260.91% |
| +/- | | -160.91% |

**Figure 5.4: Tabular Gas Breakdown**

The breakpoint however from when the contract recovers from the massive deploy cost is around 8 Transactions as observed from Figure 5.4

Extra cost of Deployment = 1017269

Savings per Tx = 130364

Breakpoint = 1017269/130364 = 7.8 = 8 Txs

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

The section concludes the project. It describes the scope for extension and improvement

## 6.1 CONCLUSIONS

Through StorkNet, we successfully demonstrate the reduction of gas cost for transactions on an EVM network. We work on cross chain interoperability of data and function executions, by implementing a queryable protocol that exists on the mainnet that can perform CRUD operations. This further shows the use of PoS as means of validating the validators while also allowing them to exit the stake contract terms early. In addition, we practically experience the use of a Pay-to-use model for the funding of StorkContracts and successfully demonstrate the use of an upgradable virtual blockchain on top of the L-2 blockchain.We also use a new consensus mechanism called Proof-of-Staking-Transactions to ensure that nodes have an equitable distribution and the return to the miners is fairly processed. This also includes ZK proofs to ensure the validity of the data being handled.

## 6.2 FUTURE WORK

We further intend on converting the virtual blockchain into a full L-2 chain and increasing the number of queries, such as Request by parameters, update by parameters, etc. This will enable us to perform more rigorous security testing in our futuristic work, thereby reducing the deployment cost of StorkContracts.

# REFERENCES

[1] Ethereum Whitepaper. https://ethereum.org/en/whitepaper/.

[2] Zubin Koticha. Introduction to Blockchain through Cryptoeconomics - Part 1: Bitcoin. https://medium.com/blockchain-at-berkeley/introduction-to-blockchain-through-cryptoeconomics, 2018.

[3] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, page 1, 2016.

[4] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4. 2019.

[5] NARA Blockchain White Paper 2019. https://www.archives.gov/files/records-mgmt/policy/nara-blockchain-whitepaper.pdf, 2019.

[6] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System.* https://arxiv.org/abs/1407.3561, 2014.

[7] Filecoin: A Decentralized Storage Network. https://filecoin.io/filecoin.pdf.

[8] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 598–609, New York, NY, USA, 2007.

[9] Ben Fisch, Joseph Bonneau, Nicola Greco, and Juan Benet. *Scaling Proof-of-Replication for Filecoin Mining*. https://research.protocol.ai/publications/scaling-proof-of-replication-for-filecoin-mining/, 2018.

[10] Ethan Cecchetti, Ben Fisch, Ian Miers, and Ari Juels. Pies: Public incompressible encodings for decentralized storage. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1351–1367, New York, NY, USA, 2019.

[11] Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. In *Lecture Notes in Business Information Processing*, pages 35–51. Springer International Publishing, 2020.

[12] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. *Combining GHOST and Casper*. https://arxiv.org/abs/2003.03052, 2020.

[13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014.

[14] Tal Moran and Ilan Orlov. *Simple Proofs of Space-Time and Rational Proofs of Storage*, pages 381–409. Springer International Publishing, 2019.

[15] Jacob Eberhardt and Jonathan Heiss. *Off-Chaining Models and Approaches to Off-Chain Computations*. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL'18, page 7–12, New York, NY, USA, 2018.

[16] Quanqing Xu, Chao Jin, Mohamed Faruq Bin Mohamed Rasid, Bharadwaj Veeravalli, and Khin Mi Mi Aung. *Blockchain-based decentralized content trust for docker images*. *Multimedia Tools and Applications*, 77(14):18223–18248, Jul 2018.

[17] Anthony Sassano. Decentralize Everything. https://thedailygwei.substack.com/p/decentralize-everything-the-daily, 2021.

[18] Preethi Kasireddy. How does Ethereum work, anyway. https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway, 2017.

[19] Mining Whitepaper. https://docs.ethhub.io/using-ethereum/mining/.

[20] Register Keeper Upkeep for a Contract. https://docs.chain.link/docs/chainlink-keepers/register-upkeep/.

[21] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, Sergey Nazarov, Alexandru Topliceanu, Florian Tramor, and Fan Zhang. *Chainlink 2*. https://www.smartcontractresearch.org/t/research-summary-chainlink-2-0, April 2021.

[22] Matic Network Whitepaper. https://whitepaper.io/coin/matic-network.

[23] Chia Coin. https://www.chia.net/.

[24] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.