May 2019

Storing IOT Data Securely in a Private Ethereum Blockchain

Vinay Kumar Calastry Ramesh
vinay.calastry@gmail.com

STORING IOT DATA SECURELY IN A PRIVATE ETHEREUM BLOCKCHAIN

By

Vinay Kumar Calastry Ramesh

Bachelor of Technology in Computer Science
Jawaharlal Nehru Technological University, Hyderabad
2014

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2019

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 30, 2019

This thesis prepared by

Vinay Kumar Calastry Ramesh

entitled

Storing IOT Data Securely in a Private Ethereum Blockchain

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Yoohwan Kim, Ph.D.
Examination Committee Chair

Ju-Yeon Jo, Ph.D.
Examination Committee Member

Fatma Nasoz, Ph.D.
Examination Committee Member

Satish Bhatnagar, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Abstract

Internet of Things (IoT) is a set of technologies that enable network-connected devices to perform an action or share data among several connected devices or to a shared database. The actions can be anything from switching on an Air Conditioning device remotely to turning on the ignition of a car through a command issued from a remote location or asking Alexa or Google Assistant to search for weather conditions in an area. IoT has proved to be game-changing for many industries such as Supply Chain, Shipping and Transportation providing updates on the status of shipments in real time. This has resulted in a huge amount of data created by a lot of these devices all of which need to be processed in real time.

In this thesis, we propose a method to collect sensor data from IoT devices and use blockchain to store and retrieve the collected data in a secure and decentralized fashion within a closed system, suitable for a single enterprise or a group of companies in industries like shipping where sharing data with each other is required. Much like blockchain, we envision a future where IoT devices can connect and disconnect to distributed systems without causing downtime for the data collection or storage or relying on a cloud-based storage system for synchronizing data between devices. We also look at how the performance of some of these distributed systems like Inter Planetary File System (IPFS) and Ethereum Swarm compare on low-powered devices like the raspberry pi.

Acknowledgements

I would like to thank my advisor, Dr. Yoohwan Kim, for helping me identify areas of research and picking a Thesis topic that I could identify with. His continuous support and mentorship helped me implement this thesis and write a report that accurately covered everything I have worked on throughout my Master's program. Over the course of this thesis, I have studied topics in IoT, Blockchains, Distributed Systems, and cyber-security and I am grateful to have a mentor whose expertise encompasses all these domains and more.

I would also like to acknowledge Dr. Ju-Yeon Jo, Dr. Fatma Nasoz and Dr. Satish Bhatnagar for their warm support and for being part of my thesis committee. I am thankful to Dr. Ajoy Datta for helping me out whenever I required his assistance. I would also like to thank all my professors for imparting their knowledge through various courses in the fields of Machine Learning, Big Data, Databases, Blockchains, and Algorithms.

I would also like to acknowledge my dad Ramesh Calastry, my mom Vijayalakshmi Calastry Ramesh and my sister Vandhana Calastry Ramesh for being my source of strength and confidence throughout my Master's program at UNLV.

Finally, I am thankful to all my friends, peers, and colleagues who have made my stay at UNLV one of the best experiences of my life and imparted significant life lessons to me.

Vinay Kumar Calastry Ramesh

University of Nevada, Las Vegas

May 2019

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Information of Things (IoT) is an exciting field in Computer Science with the potential to disrupt several industries such as Health, Energy and Utilities, Transportation, Shipping and Retail. Devices like the Apple Smartwatch has the capability to detect a heart attack and alert emergency services if needed using its EKG sensors. Recent research has suggested that a lot of these devices will be coming online in coming years and each of them will be generating small payloads of data in large quantities. This requires a secure and trusted storage model with means to protect that data from being stolen.

In recent years, blockchain has been rapidly gaining momentum in areas other than its most famous use-case as a crypto-currency to be used as a drop-in replacement for regular currency. Blockchain/ Distributed Ledger implementations such as Ethereum and Hyperledger Fabric have introduced a concept called programmable blockchain which has opened avenues in which blockchain can be effectively used to improve and optimize industries like Shipping, Supply chain, and Internet of Things (IoT). Rather than just storing data such as balance amounts of a certain wallet for a cryptocurrency in the blockchain, programmable blockchain has allowed us to store and execute code which can be used to store and retrieve all kinds of data in a decentralized manner. This concept in Ethereum is called Smart Contracts which can be written and deployed to any Ethereum network.

IoT security is a major concern for edge devices like Raspberry Pi where data privacy concerns need to be addressed, data confidentiality and integrity must be guaranteed if we are to trust the data from an IoT device.

1.1 Objective

This thesis will explore how to store IoT data on a combination of an on-chain (Ethereum Blockchain) and off-chain storage (IPFS and Ethereum Swarm) in an encrypted fashion and use it in a real-time publish-subscribe model without the use of any M2M protocols such as MQTT or CoAP. We will also estimate the performance of this system in terms of the number of transactions that can be executed per second and also try to optimize its performance.

1.2 Outline

In Chapter 1, we introduce the reader to IoT devices and the role they are poised to play in our lives and how we could integrate them with other technologies such as blockchain.

In Chapter 2, we look at the motivation that pushed us to consider research in Internet of Things and Blockchains, existing research in this arena and finally propose a new system to overcome existing issues in these areas.

In Chapter 3, we explain different blockchain technologies currently in use and how all these technologies are used to solve certain problems.

In Chapter 4, we describe IoT terminologies, protocols, and devices considered for use in our thesis. We will also explore how to establish a root-of-trust for these devices and enforce security features using external hardware.

In Chapter 5, we describe the proposed system of On-chain and Off-chain storage to store data from sensors in a decentralized fashion. We also describe the Smart Contract used to register IoT devices and store data.

In Chapter 6, we perform experiments regarding data storage using traditional databases as well as the proposed system using Ethereum Blockchain, IPFS and Swarm. To understand the cost of IoT security, we perform experiments to gauge the performance of the proposed system.

In Chapter 7, we will attempt to summarize findings from this thesis and end with a retrospection of performance measurements of these storage systems along with blockchain.

Chapter 2

Motivation and Background

2.1 Motivation

IoT devices have been plagued with issues for as long as they have existed as a technology and until very recently, these devices have been confined to use in hobby projects and never saw mainstream success until the power of cloud computing was leveraged to power devices such as Alexa Echo, Google Home Minis. This has also allowed internet controlled sensors to be used in smart refrigerators, cars, and ovens. There is a huge overhead to set up a comparable infrastructure in terms of storage and messaging protocols if we do not want to depend on these managed cloud solutions. Decentralized systems like blockchain, especially Ethereum [CD16] and all other derived blockchains do hold promise in the sense they provide a networking method to connect to other devices running the software and also provide storage solutions.

Security concerns are a very important issue to be addressed in these devices with the primary concern being a ”difficult to secure” storage mechanism for sensitive data such as private keys and secrets for APIs. Hardware Security Mechanisms or Trusted Platform Modules (TPM) aim to establish a hardware root-of-trust to encrypt storage devices and store sensitive data.

Blockchain is still a very nascent technology and has many issues of its own. Transaction rate per second is, by design, very low compared to traditional storage and processing systems and most blockchains are designed to be fully open and make no provisions for data privacy. Only the validity of the data is confirmed through Consensus mechanisms enforced by the blockchain network.

Data Storage on blockchain is expensive for large quantities of data. Coupled with the transaction rate issue, blockchain seems a bad way to store data. However, Decentralized technologies like Inter Planetary File System (IPFS) or Ethereum Swarm may be used to store the actual payload while the Blockchain implementation such as Ethereum can be used to store links to the actual data stored in these systems. The edge devices need not store either the links to the data or the actual payload itself. Any number of high-performance devices can be added or removed to take up the work of storing data on the storage system and blockchain.

This solves both the transaction rate and storage cost issues. We do not need to wait for the transactions to be mined and instead proceed to the next transaction. The miners on the network will work at their own pace and process these transactions and we will not be required to pay for storage costs which need to be paid otherwise. We can introduce centralization and assert control over the system by making all the IoT devices register on the network before they are allowed to send or receive data. This will help prevent unauthorized access to the network.

2.2 Previous work

Research in the Internet of Things and Blockchain has seen a major resurgence after a sudden interest in crypto-currencies and mining. Companies like IBM [ABB^{+}18], Amazon and Microsoft have contributed to the recent uptick in using blockchains for commercial applications.

Various research publications have presented novel solutions for telecommunications [KK18], voting [KKKH18], building decentralized social networking systems [XSML18], smart homes[XHLX18] and stock exchanges [PPM^{+}18] have been proposed. Using Public blockchains to store industrial data in encrypted format has also been studied extensively[PE18].

IPFS can also be used as a powerful replacement for Swarm, as evidenced by the fact that IPFS aims to solve most of the same problems that Swarm is being developed to fix. Ethereum + Swarm and Ethereum + IPFS have been studied to store user data [ACG^{+}18], Access Control of data [WZZ18] stored on the system, encryption of sensitive data on public blockchain networks [WZZ18] as well as storing IoT data [OY19] and IoT Marketplaces [DY18].

Research on the benchmark of EVM bytecodes [AAv18] and Object-oriented methods used for writing smart contracts [Heg18] have provided me with a significant understanding of writing efficient smart contracts following best practices.

Performance analysis of decentralized solutions has also been considered in various research publications [RD17] where a private network of ethereum was tested for its performance metrics. Curiously enough, performance metrics of Ethereum with IPFS or Swarm have not been thoroughly researched.

Current research in blockchain applications for IoT has mostly kept to ensuring privacy, authentication and other security considerations [Ksh17] for IoT devices and sensors. A prime area of research is to investigate whether the data collected and stored by IoT sensors can be securely retrieved and used by other sources, the various systems that can be used for such purposes and which ones provide the most ideal solution. Using insights gleaned from [Y17] and from [OY19], methods to integrate IoT devices in a blockchain infrastructure are studied. From [WZY^{+}18], ways to enforce SSL/TLS security are explored in this thesis.

2.3 Proposed Solution

In this thesis, we will build an encrypted storage model for IoT data where data generated by sensors is stored securely in multiple distributed nodes of either Ethereum Swarm (or) IPFS while the order of data collected is preserved using the Ethereum Blockchain and Smart Contracts. Setup of IPFS, Ethereum Swarm and Private Ethereum Blockchain is done and data is collected from Sensors such as the Digital Humidity and Temperature Sensor (DHT11). Edge Nodes (Non-mining light Ethereum and Swarm nodes) on tiny computers like Raspberry Pi 3B+ are used to collect and store these sensor readings to the network while miner nodes running on more powerful machines are used to process and validate transactions in the network. The strength of this model lies in its design where there is no single point of failure and any node, miner or non-miner can be added, removed and replaced without affecting the functioning of the overall network, provided another node is added before the previous one is removed. Control over the devices is maintained by using Smart Contracts to register a device [ZKS^{+}18] before it can send or receive data from the system. However, the presence of distributed devices adds a significant overhead of securing the information shared between them. Public-Private key encryption such as RSA can be coupled with symmetric encryption like Advanced Encryption Standard (AES) and authenticity is verified using Keyed-Hash Message Authentication Code (HMAC). We propose both hardware and software-based methods such as a Trusted Platform Module (TPM) to fully encrypt File system of the storage medium on the edge devices using Linux Unified Key System (LUKS) and store secret keys used for symmetric encryption on the device storage securely by locking it to the system with the private key of TPM.

Chapter 3

Blockchain Technology

3.1 Blockchain

A Blockchain is, put simply, a series of blocks put together in a chain. Most blockchains are transactional state machines which take in transactions (input) and generate a new state (output) and is shown in Figure 3.1

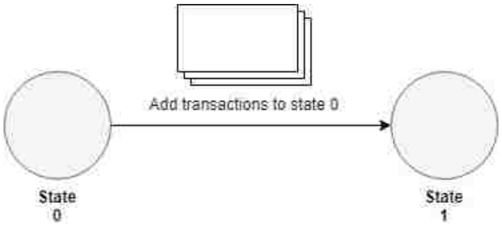


Figure 3.1: Transaction State

Blockchain or more accurately Distributed Ledger Technology was built to power the storage and transfer of a new form of currency called Crypto-currency, the first of which was called Bitcoin. This technology was invented by a person (or) a group of persons under a pseudo-name called Satoshi Nakamoto [Nak09]. Due to the runaway success of Bitcoin, it has become synonymous with blockchain today. But blockchain technology is not limited to serving as the underlying technology for a crypto-currency, rather it is just one subset of a much larger ecosystem consisting of a number of different use-cases. Only recently, Blockchain based use-cases have been explored in different applications like supply chains, loyalty programs, data sharing, and real estate.

Blockchain is decentralized by design [GBE+18] and having the same information across all nodes results it being a resource owned by everyone without a single point of failure.

We need to run complex cryptographic algorithms to validate these transactions when using blockchain. This can be done by anyone who runs the blockchain software and once successful in validating these transactions called consensus, a new block can be added to the chain. This process is called mining. A fee is paid to the successful miner in the form of BTC for bitcoin and ether for Ethereum. Table 3.1 explains a few differences between Bitcoin and Ethereum, two of the most popular blockchain implementations in existence.

Table 3.1: Differences between Bitcoin and Ethereum

Characteristic	**Bitcoin**	**Ethereum**
Year Introduced	2009	2015
Inventor	Satoshi Nakamoto	Vitalik Buterin, Gavin Wood
Currency Unit	Bitcoin	Ether
Smallest Unit	Satoshi	Wei
Symbol	btc	eth
Total Supply	21 Million btc	Unlimited
Block creation time	10 minutes	10 - 12 seconds
Intended Use	Payment Network	Programming Platform(dapp)
Mining Reward	12.5 btc	3 Eth
Mining Algorithm	SHA-256	Ethash

3.1.1 Blockchain Concepts

Chaining of Blocks

Blocks are generated and filled with transactions. When filled up, this block is broadcast over the network and mining is performed. Once complete, this block is chained with the previous block creating a ledger. This process can be repeated infinite number of times to create a never-ending chain of blocks. The process of chaining blocks is shown in Figure 3.2.

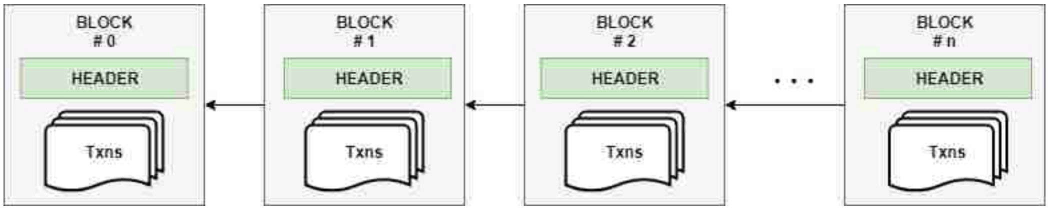


Figure 3.2: Block Chain

Each block stores the hash value of the previous block and if someone changes the contents of a historical block, its hash value changes and will not match the hashes stored in any subsequent blocks. When this chain is replicated across a number of different users, the blockchain copy of the person who changes the contents will not match with the contents of other users, thus creating a permanent, immutable decentralized chain of blocks.

Peer to Peer Discovery

Peer to Peer discovery[ADDPSHJ14] is the cornerstone of any blockchain. This is how different peers find each other and synchronize their state of the blockchain with them. P2P isn't new and has been used previously in applications such as Bit Torrent, Napster to name a few.

Distributed vs Decentralized Systems

Distributed Systems [NHE17] are still centralized and require a server to connect to each client and share data with them. Blockchains and other related systems are designed to be independent of a server and instead depend on P2P synchronization and group consensus mechanisms to communicate with each other and make decisions.

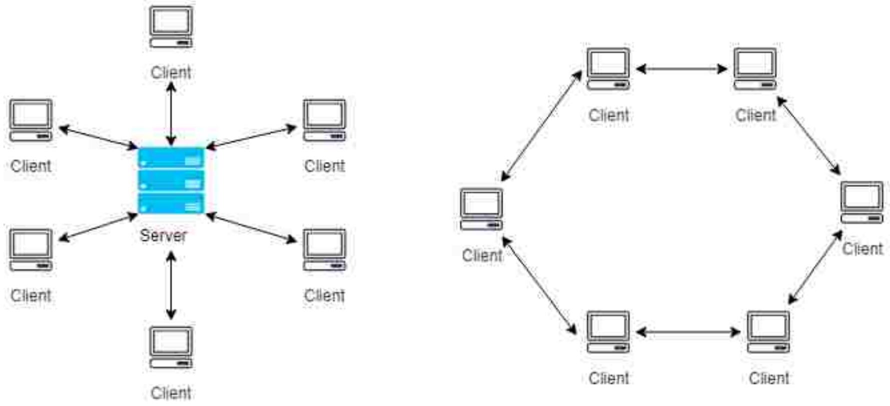


Figure 3.3: Distributed vs Decentralized Systems

Crypto-currency Mining

In simple terms, mining is the process of performing computationally intensive mathematical (or) more accurately cryptographic calculations. These calculations are so complex and require massive amounts of processing capacity and power to run. The computers that are successful in performing this operation are paid with a reward in the newly minted currency that didn't exist previously. This is a direct analogy to the mining of gold from the ground. By performing this process, the miners are validating the authenticity of the payment and adding these validated transactions to the network. Mining is what makes the payment network secure and trusted without a trusted and centralized third-party to validate the transaction which is the case in traditional methods. The cryptographic calculations performed by these miners is called Proof of Work (PoW). Mining is hard but verification is relatively easy and is performed by all the nodes in the system in a process. The Proof of Work which is needed to validate a transaction is called Consensus.

A new block is created roughly 10 minutes in Bitcoin whereas a new block is created every 10 - 12 seconds in Ethereum. Every block creates 12.5 new BTC in Bitcoin while a new block in ethereum generates 3 ether.

Distributed Apps

The term Distributed Apps(dapps) is used to refer to any application (web or standalone) which is able to interface and share data with the blockchain. Dapps differ from regular web or mobile apps because their back-end logic is written and run entirely in the blockchain.

3.1.2 Private vs Public blockchains

Bitcoin and Ethereum were primarily intended to be used as public blockchains [But15]. It means data stored in the chain is visible for everyone to read and write. However, these networks can be set up to run as private blockchains, process payments and run Smart Contract code. This basically means that an Access Control Layer is built over a regular blockchain to exercise control over who is allowed to read or write data to the blockchain. These private blockchains can also be called permissioned blockchains.

Table 3.2: Public vs Private Blockchain

Public Blockchain	**Private Blockchain**
Anyone can join (permissionless)	Requires permission to join (permissioned)
No one has control	Controlled by a person/organization
Typically used in Currencies	Used to store private data such as land records
Large number of Nodes	Less number of Nodes
Slower Transactions	Faster Transactions
Scalability limited by design	Design can be tweaked to allow more transactions
Good for Anonymity of users	Users must be known

3.2 Ethereum Blockchain

Ethereum is a popular implementation of blockchain and provides a platform to run smart contracts [But13]. These smart contracts can be used to facilitate monetary transactions and also store important data in a distributed ledger [EHH$^+$16]. Ethereum has two types of currencies, gas and ether. Gas is the ether that must be paid by nodes that run smart contracts. Ether is the crypto-currency in Ethereum which can be used to transfer money and to pay for gas to run smart contracts.

Ethereum introduces a number of concepts to enable sending and receiving ether, executing smart contract transactions and reading data to and from the blockchain in general.

3.2.1 Performance and Scale

As explained in 3.1.1, a new block in Ethereum is generated every 10 - 12 seconds. Because of an intentional limit on computation per block, transactions per second is limited to only an average of 15 per second. Other techniques such as Shading and off-chain storage are touted as ways to improve this scaling factor.

Transactions costs in gas must be borne by the Ethereum client that submits the transaction and is calculated in the following manner.

$$TxCost = gasLimit * gasPrice \tag{3.1}$$

Average gas limit per block (at the time of writing) is around 8,000,000. This value is decided by miners and changes every block. With an avg block time of 12 seconds and a gas price per transactions ranging between 20,000 to 60,000 on average, we can expect a range of 10 to 20 transactions per second.

3.2.2 Ethereum Wallet Address

Ethereum uses ECC (Elliptic Curve Cryptography) to generate public, private keys and ECDSA (Elliptic Curve Digital Signatures to verify and sign transactions. The figure 3.4 shows the elliptic curve called secp256k1 used in Ethereum (As well as Bitcoin and other major blockchains).

Two points are chosen and the public-private key pair is generated using them. The private key is used to sign transactions while the Keccak-256 (A variation of SHA-3) hash of the public key is taken and the rightmost 20 bytes is registered as the Ethereum address.

Addresses in Ethereum are of two types.

1. Externally Owned Accounts
2. Contract Accounts

Externally Owned Accounts are used by nodes to send and receive ether. These are the most commonly used accounts and have an associated private key.

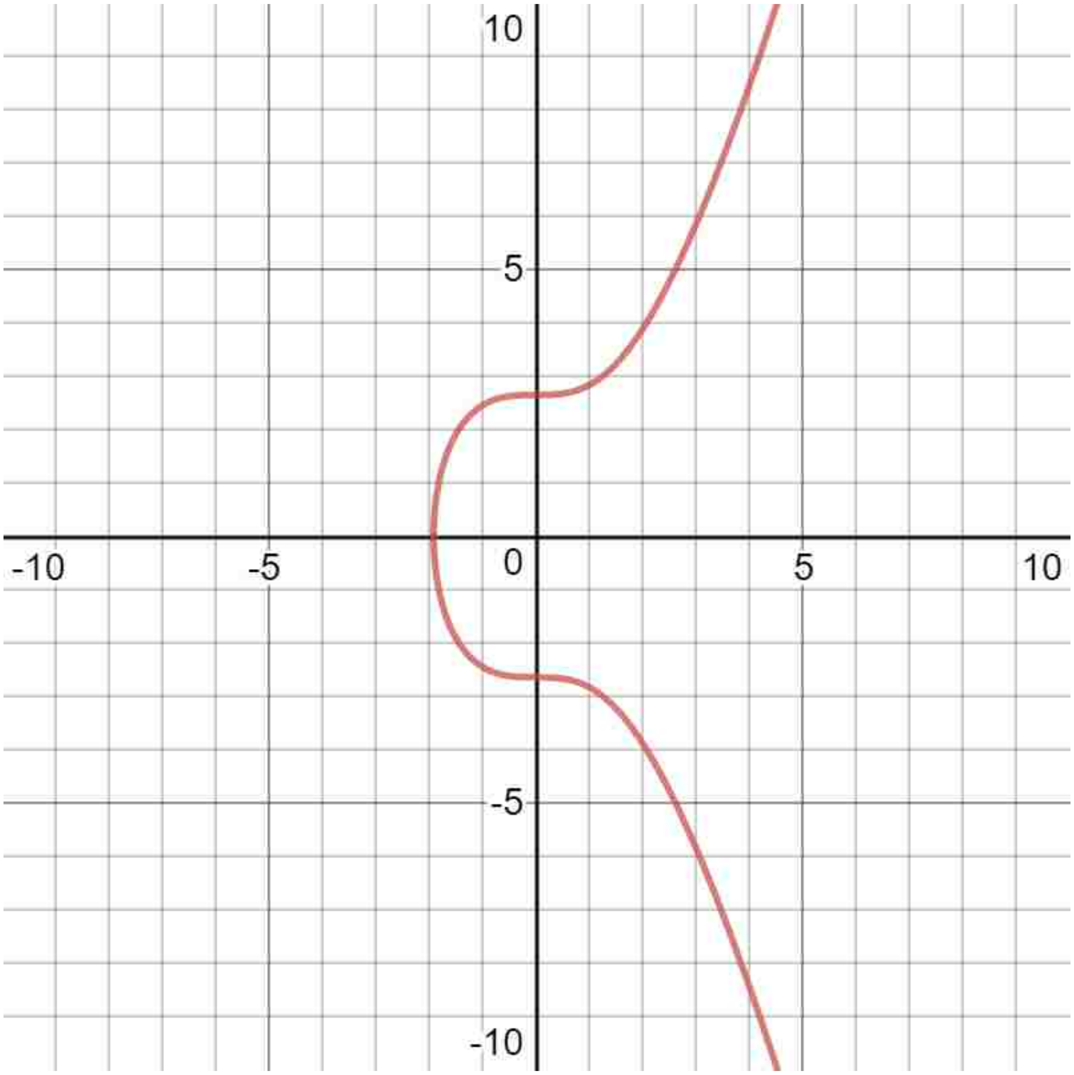


Figure 3.4: Elliptic Curve

Contract accounts hold contract code and their address is used to execute smart contracts. They do not have an associated private key and their address can be called by clients to execute code stored within it.

3.2.3 Crypto-Currency in Ethereum

Ethereum pays crypto-currency called ether to successful miners. This can be subdivided into smaller units up to absolute smallest unit called wei. The name Wei is a tribute to one of the earliest visionaries of cryptocurrency, Wei Dai, the author of an early blockchain based protocol and cryptocurrency called b-money [Dai98].

Like any other currency, ethereum is not immune from volatility and ranges widely. To prevent volatility in the amount (called gas) that must be paid for executing transactions on the Ethereum network, it is independent from actual Ether exchange rates.

Gas can be defined as a special unit used in Ethereum to calculate costs incurred for executing that transaction. The gas limit must be mentioned in the transaction and must be reasonable. If the limit is too low, it is unlikely that any miners will process that transaction. Once the transaction is executed, the miner that processes that transaction will be paid in ether corresponding to the gas that was spent. The remaining gas is returned to the account that requested the transaction.

Table 3.3 shows different denominations in Ethereum and their respective exchange rate with ether.

Table 3.3: Denominations in Ethereum

Name	Conversion Rate	Special Name	In Honor Of
wei	10^{-18}	-	Wei Dai
kwei	10^{-15}	ada	Ada Lovelace
mwei	10^{-12}	babbage	Charles Babbage
gwei	10^{-9}	shannon	Claude Shannon
micro	10^{-6}	szabo	Nick Szabo
milli	10^{-3}	finney	Harold Finney
ether	1	-	-
kether	10^{3}	einstein	Albert Einstein
mether	10^{6}	-	-
gether	10^{9}	-	-
tether	10^{12}	-	-

3.2.4 Mining Algorithm

Ethereum uses Eth-Hash algorithm [Woo14] for finding cryptographic hash for a block (process of mining). Before mining starts, a large Directed Acyclic Graph (DAG) is created. The mining process attempts to solve a certain condition in it. This process is Proof of Work (PoW) in ethereum and is designed to be verified by other nodes very fast in linear time on a CPU using fewer resources.

Nonce is guessed by miners to build a block and add it to the chain. Guessing exact nonce is incredibly hard, and so a difficulty value is set and modified after each block. If a block is mined before the average mining time (10 - 12 seconds), the difficulty in increased and is reduced when the mining time exceeds the average mining time.

3.2.5 Genesis block

Genesis block is the first block of the entire chain and is named block 0 in Ethereum. This has to be created manually for private networks and contains a hash of zeros. This block is represented in the form of a JSON file and is downloaded first when a new node is started and added to an existing Ethereum Network.

```
{
    ”nonce”: ”0x0000000000000042”,
    ”mixhash”:
    ”0x0000000000000000000000000000000000000000000000000000000000000000”,
    ”difficulty”: ”0x400”,
    ”alloc”: {},
    ”coinbase”: ”0x0000000000000000000000000000000000000000”,
    ”timestamp”: ”0x00”,
    ”parentHash”:
    ”0x0000000000000000000000000000000000000000000000000000000000000000”,
    ”extraData”: ”0x436861696e536b696c6c732047656e6573697320426c6f636b”,
    ”gasLimit”: ”0xffffffff”,
    ”config”: {
        ”chainId”: 63723,
        ”homesteadBlock”: 0,
        ”eip155Block”: 0,
        ”eip158Block”: 0
    }
}
```

3.2.6 Networking Modes in Ethereum

Public Ethereum Network

There are two kinds of public networks maintained by Ethereum Foundation. First is the main network with a network id of 1. There are other test networks which can be used by developers to test features of their distributed apps built using ethereum.

Table 3.4 provides a brief explanation of different main and test networks.

Table 3.4: Public Ethereum Networks

Network	**Type**	**Network ID**	**Network Status**
main	Main	1	Online
morden	Test	2	Retired
ropsten	Test	3	Online
rinkeby	Test	4	Online

Private Ethereum Network

Private networks can be set up in both closed and open modes. All nodes in this network need to be started with an ID other than the ones used by public networks. Another requirement is that all the nodes in the private network need to be initialized with the same genesis block. A random number of **63723** was chosen for this thesis. If we require that the private network should not be accessed by anyone with the network id and genesis block, we can turn off the discovery mode for other peers and instead add them manually.

3.2.7 Geth

Geth is a client for Ethereum, implemented in Golang and used for connecting to the Ethereum network. Other implementations in C++ and Python are also available. Once installed, it can be initiated with genesis block (the first block in the blockchain), started and connected to different networks available in Ethereum. These networks are identified by their network ID. Different nodes with same genesis block and network ID can synchronize with each other and mine transactions. Network ID: 1 is for the main network, 2 and 3 are reserved for Test networks. We can use a different network ID for setting up a private network. Different versions for Windows, Linux, MacOS and Android are available.

Table 3.5: Example eth and web3 commands in geth

API	Command	Function
geth	geth attach	Opens access to blockchain
eth	eth.coinbase	Address of the primary account
eth	eth.getBalance()	Returns balance in wei
eth	eth.sendTransaction()	Sends ether in wei between accounts
eth	eth.pendingTransactions	Returns list of pending transactions
eth	eth.coinbase	Address of the primary account
web3	web3.fromWei()	Converts wei to Ether
web3	eth.toWei()	Converts Ether to wei
miner	miner.start()	Starts mining on the node
miner	miner.stop()	Stops mining on the node
admin	admin.nodeInfo.enode	Gets the ENODE value of the geth node

3.2.8 Geth Synchronization modes

Geth can be run in 3 different modes: Full, Fast or light each with its own usage and purpose.

Full

This is the default mode of geth when running without any startup options. It requests the entire database state, gets headers, body and validates every element starting from the genesis block.

Fast

This mode gets block headers, body and doesn't process any transaction until it reaches the current block. Thereafter, it works exactly like full synchronization mode.

Light

This mode only gets current state from blockchain. To verify elements, it needs to request full nodes of previous states. This is perfect for edge devices like raspberry pi where we do not want to store large amounts of data. However, it requires a full node dedicating some of its resources to serve and support requests from a light node.

3.2.9 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is a virtual environment that runs on all geth nodes and executes code stored in smart contracts. A variety of programming languages such as Solidity, Vyper, LLL, etc are available for writing Contracts which can be compiled to EVM byte code.

3.2.10 Interfacing Methods

Depending on whether we want to access our node from same machine or from different machines on the network, Inter-Process Communication (IPC) or Remote Procedure Calls (RPC) are used.

Inter Process Communication

Inter-Process Communication (IPC) is the preferred method of connecting to an ethereum node from processes running on the same system. If the node is started without any options, the only method available to connect to that node is through IPC. Geth creates a geth.ipc file in its home directory and must be referenced when we want to connect to the system. For security reasons, IPC is limited to localhost and cannot be run from another machine.

Remote Procedure Calls

Remote Procedure Calls should be used if the ethereum node is running on a different machine. Geth provides HTTP as well as WebSocket proxies to connect to ethereum. Additionally, the format used to interface with ethereum is called JSON-RPC. It is the interface using which a front-end application written in any programming language can communicate with the business logic and data stored in the blockchain.

3.3 Smart Contract

A smart contract is a Class like code that lives in the Ethereum blockchain (EVM). Methods declared within the Smart Contract can be run as a transaction by any node which has enough ether to pay for running that transaction. This contract can be accessed by the outside world through distributed apps (also called dapps). Solidity compiler (solc) is used to compile these smart contracts into EVM bytecode (containing machine-code like instructions compiled from solidity) and ABI Definition containing meta-data like variables and methods used in smart contract.

A typical Smart Contract consists of variables, setter and getter methods with access modifiers to regulate access exactly like Object-Oriented languages such as C++ and Java. Setter methods involve changing the state of the data contained within in blockchain and thus incur a transaction charge which must be paid in gas. Getter methods only retrieve the data stored within the full node or from a peer and do not incur any charge.

3.3.1 Solidity

Solidity is a Turing complete Contract-oriented programming language available for Ethereum to write Smart contracts. It is a high-level language supporting all features of a modern programming language such as static-typing, inheritance, and complex user-defined data types. This enables us to build smart contracts capable of building distributed apps like Maintaining Land ownership records, Voting, Auction and Betting apps among others.

3.3.2 Remix

Smart contracts can be written and tested using a browser-based IDE called Remix. Additionally, plugins and frameworks to create and test Smart Contracts are available in many popular IDE and text editors.

3.3.3 Smart Contract Concepts

A typical Smart Contract looks like a Class definition written in languages like C++ or Java. In addition to commonly available object-oriented principles such as constructors, methods, variables, visibility modifiers, Inheritance, arrays, and loops; solidity provides a few additional capabilities to bolster the usability of Smart Contracts for solving different use-cases.

solc compiler

Smart Contracts need to be compiled to EVM byte-code and ABI definition before the Virtual Machine in an ethereum-node can access and execute transactions and methods mentioned in the blockchain. Ethereum tools like Remix and Truffle are bundled with this compiler. The version used to compile the smart contract is mentioned at the very top of the solidity code.

```
pragma solidity ˆ0.4.24;
```

0.4.24 is the version of the solidity compiler to be used and the symbol before the version number implies that the source file would not be valid above 0.4.24

Variables

Variables in Solidity are statically-typed and must be declared only once. The compiler throws an error if a variable is declared more than once throughout the source file. The table 3.6 explains some of the commonly used data types available in solidity.

Table 3.6: Data Types in Solidity

Data Type	**Value Types**	**Example**
boolean	True,False	True
integer	uint, int	8, -20
address	hexadecimal string	0x123
String	String literals	”foo”

Arrays

Arrays can be of both dynamic or static types. Use of Arrays, either static or dynamic is discouraged in solidity as they can grow to large sizes and require large amounts of gas to process simple operations such as lookup of data. The below example shows a static array of size 7.

```
uint[] a = new uint[](7);
```

Mapping

Mapping is encouraged to be used within solidity in place of arrays. Mapping is defined as a key-value pair of any data type for both key and value. They can be thought of as hash tables that are initialized when declared, contain every possible value and are mapped initially to a byte representation of all zeros. The key is not stored in the mapping, only its keccak-256 hash is stored and used to match its corresponding value. The below example shows how mappings can be declared and used.

```
//declare the mapping
mapping(address => uint) public balances;

//store value in a mapping
balances[addressSender] = newBalance;
```

Structs

Structs are similar to the C language structs and are used to store values of a number of different data types. Structs can be used within the mapping data type and are used extensively in solidity. An example struct is shown below:

```
//struct to store the temperature, humidity, and timeStamp
struct SensorData{
    uint64 temperature;
    uint64 humidity;
    string dataStorageTime;
}
```

Constructor

Constructors in solidity provide the same functionality as constructors in any other object-oriented programming language. The below example is taken from the smart contract used in this thesis.

```
//Constructor for the Smart Contract
constructor() public {
    currentID = 1;
    createdBy = msg.sender;
}
```

Methods

Methods are functions defined within a smart contract and can be used to modify or retrieve state of the blockchain. However, methods that serve as setters, i.e modify state of blockchain cannot return an output. Calling such methods using web3 returns only their transaction hashes and gas must be paid in ether for services used. Methods that do not modify the blockchain and simply return the state of a value stored in their local blockchain copy and do not incur any gas costs. Example of a method to set a value is shown below.

```
//register IoT device
function registerDevice(address addressToAdd) ownerOnly public {
    if(devicePresent(addressToAdd)) {
        emit deviceEvent(addressToAdd,”DEVICE ALREADY REGISTERED”);
    }
        trustedAddresses[addressToAdd] = true;
        emit deviceEvent(addressToAdd,”SUCESSFULLY REGISTERED”);
}
```

Events

Events are special methods that can be emitted when a certain condition is met in a Smart Contract. For example, To indicate completion of a successful transaction or to intimate the occurrence of error. These events can be listened to by another smart contract or from other sources through web3. An example event is indicated below.

```
//Transaction successful
event setFileHashEvent(
    address indexed _from,
    string _message
);

//Event triggered using emit keyword
emit setFileHashEvent(msg.sender, ”FILEHASH TXN CALLED”);
```

Modifiers

Modifiers are used when we want to prevent certain kind of users from accessing the blockchain. A basic form of Access Control can be established within a smart contract using Modifiers. These modifiers can then be applied to methods that are run only if the conditions in modifier are met. Example of a modifier is given below.

```
//Modify some functions to be executed only by Contract creator
modifier ownerOnly {
    require(msg.sender == createdBy);
    _;
}
```

3.3.4 Web3

Web3 is a framework developed by Ethereum Foundation and enables a developer to build a client (web application or mobile app) which can interact directly with the Ethereum Blockchain. Functionality provided by web3 includes getting wallet address of current node, sending ether to some other node, etc. It can also be used to connect to a Smart Contract which is deployed in the blockchain and execute methods defined in them. These functions can be anything from transferring ownership of land from one person to another person, transferring ether or casting a vote for a candidate in an election. Web3 is available in javascript, python and other languages. Web3.js is, by far, the most popular framework available which enables a user to connect a web page to some functionality in the back-end Smart contract. Most IOT devices have excellent support for C and Python. web3.py was used in favor of web3.js as an interface between IoT devices and Ethereum Blockchain for this thesis.

3.3.5 Truffle

Truffle is a framework which allows rapid development and deployment of Smart Contracts in Ethereum. Additional functions such as retrieving the smart contract's address and ABI definition can also be done using truffle.

Table 3.7 describes a few commands that allow truffle to compile and deploy smart contracts.

Table 3.7: Example truffle commands

Command	**Function**
truffle compile	Compiles a smart contract to EVM bytecode
truffle migrate	Deploys the smart contract to a network mentioned in its configuration
truffle console	Accesses the meta-data for the smart contract

Table 3.8 provides commands that can be run in the truffle console.

Table 3.8: truffle console commands

Command	**Function**
ContractName.address	Returns the address of the Smart Contract
ContractName.abi	Returns the abi metadata of Smart Contract

3.3.6 Ganache

Ganache is a framework which can be useful for testing Smart Contracts during development. It provides a bootstrapped private blockchain with test accounts pre-loaded with ether.

3.4 Alternate Decentralized Storage Solutions

Ethereum allows data storage via Smart contract variables. This storage comes with a steep cost and as the size of the data stored increases, the cost to load transactions to the blockchain also increases.

Another major disadvantage of using Ethereum as the sole data storage mechanism is that data stored would have to contend with severe limitations placed on variables in Solidity. This is by design and storing huge volumes of data on the blockchain is discouraged by Ethereum to prevent abuse by choking the network causing Denial of Service and other network attacks. The entire data has to be replicated across all nodes and this extreme level of redundancy is not required for our use-case.

To overcome these limitations, data can be stored in alternate decentralized storage solutions such as Ethereum Swarm or Inter Planetary File System. We will compare performance using both these solutions in this thesis. Additional storage schemes such as Siacoin [VC14], storj [WBBB14] etc. are in various stages of development.

Both Swarm and IPFS offer excellent solutions to establish a decentralized storage layer for the internet utilizing a server-less hosting platform. They offer a layer of incentivization for participating nodes to ensure that the data stored in them will not be deleted and also propose a block storage model where larger documents are served in blocks and can be fetched in parallel. They provide integrity protection by content addressing (filehash) and offer decentralized Domain Name Resolution.

3.4.1 Ethereum Swarm

Ethereum Swarm [ES] is a storage solution that aims to utilize a bit-torrent like Peer to Peer (P2P) protocol to store data in a decentralized fashion distributed in a large number of nodes. It is part of the geth tool-chain and uses a lot of technologies invented by the Ethereum foundation. The current implementation is at version 0.3.x or Proof of Concept 3.

It provides an HTTP proxy to access data stored across nodes. A file stored in Swarm returns a file hash uniquely identifying that resource.

Its primary objective is to allow dapps to store data efficiently for applications such as messaging, data streaming and mutable resource updates. Since it is endorsed directly by the Ethereum Foundation, swarm is a pretty good choice for this thesis.

3.4.2 Inter Planetary File System

The Inter-Planetary File System (IPFS) [IPF14] is a storage solution, very similar to Ethereum Swarm and solves a lot of similar issues faced by Swarm. However, it has been around for much longer than Ethereum and is much easier to set up and use.

Like Swarm, it returns a file hash for data stored in the IPFS network. Unlike Ethereum Swarm, it provides a new protocol to access content stored in its nodes. It also provides a User Interface to access the files stored on a node graphically.

3.4.3 Comparison between Storage Solutions

Table 3.9 shows a simple comparison between different storage solutions including regular Databases.

Table 3.9: Comparison of Storage Solutions

Comparison	IPFS	Swarm	RDBMS	NoSQLDB
Decentralization	Yes	Yes	No	No
Distributed	Yes	Yes	Yes	Yes
Single Point of Failure	No	No	Yes	Yes
Is Production Ready	Yes	No	Yes	Yes
Privacy of Data	No	No	Yes	Yes
Speed	Slow	Slow	Fast	Fast
Flexibility of Data Format	Flexible	Flexible	Not Flexible	Flexible

Table 3.10 shows technical differences between IPFS and Swarm.

Table 3.10: Technical differences between IPFS and Swarm

Comparison	**IPFS**	**Swarm**
Storage Component	Distributed Hash Tables	Immutable Content Addressed chunkstore
Cloud Hosting Like Service	No	Yes
Integration with Ethereum	None	Full
Network Layer	libp2p	devp2p
Incentivation Layer	FileCoin	Ethereum

Chapter 4

IoT Technology

4.1 IoT Systems and Protocols

Traditionally, Information of Things devices connect with each other over the network using a number of client-server messaging protocols also called Machine to Machine protocols (M2M). CoAP (Constrained Application Protocol), MQTT (Message Queuing Telemetry Transport) and HTTP are most used among many available M2M protocols.

4.1.1 M2M protocols

IoT devices use M2M protocols to communicate with other devices, often with other devices which connect to actual databases, Web APIs or output displays. They require a server running a message broker and a number of clients which connect and get data from the server and other clients through dedicated channels. These can be either request-response models(HTTP, CoAP) or publish-subscribe models(MQTT, AMQP).

However, messages can also be transferred between devices using Smart Contracts sending messages using Ethereum Blockchain. A centralized server is no longer required if we use Ethereum instead of relying on M2M.

4.2 IoT devices

Low powered Single Board Computers are used for Edge computing operations such as displaying data to LCD devices and collecting data from sensors. Popular devices include Raspberry Pi, Arduino and well as many other devices from well-known manufacturers like Intel and Nvidia.

4.2.1 Raspberry Pi 3

A raspberry pi (Version used is 3B) is a cheap (costs), credit card sized general purpose computer with a 1.2 GHz ARM Cortex A53 quad-core processor with 1GB of RAM. It can many linux based OSes like Raspbian (the default Linux OS from Adafruit, manufacturer of Raspberry Pi) and third-party OSes like Ubuntu Snappy Core, Windows IOT Core, and RISC OS. The device also provides a set of General-Purpose IO headers which can be used to connect several sensors and display devices to this machine which makes it a suitable candidate for use in our thesis. It doesn't provide any BIOS (Basic Input/Output System) and can be booted from a USB flash drive or externally connected hard disk. However, a micro SD card is the standard way to install an OS and boot up the system.

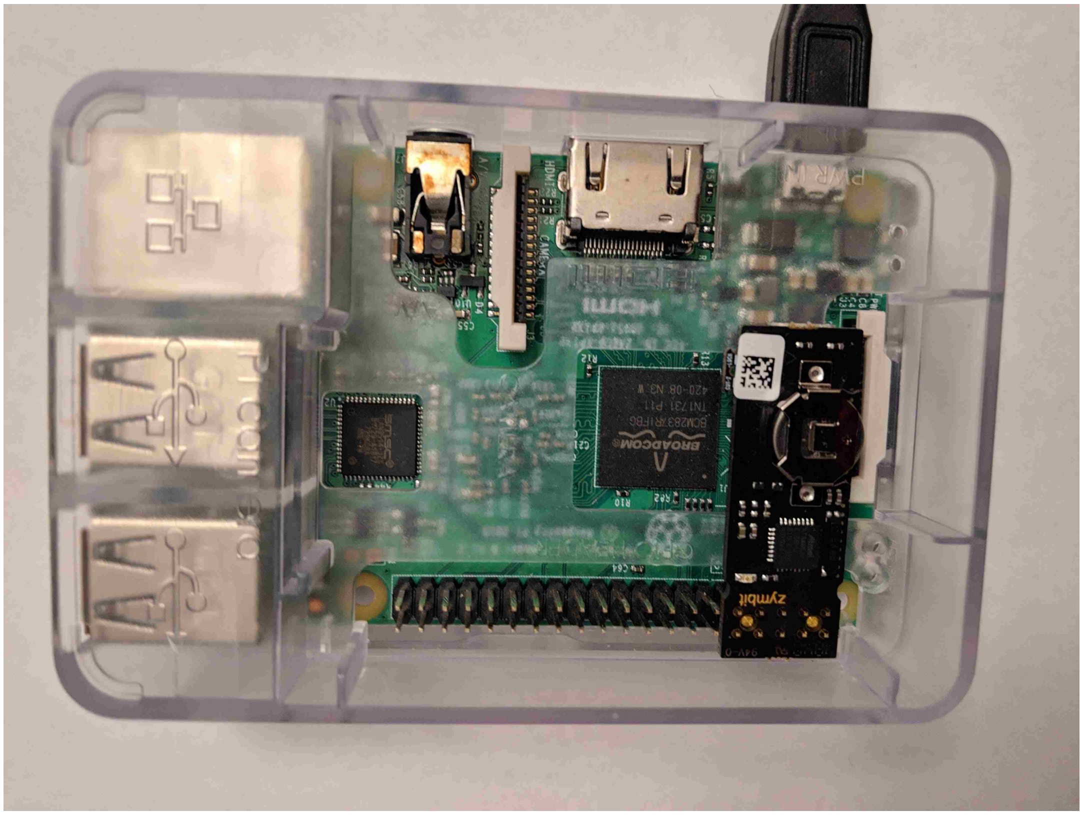


Figure 4.1: Raspberry Pi 3B

4.2.2 Rasbpian

Raspbian Stretch Lite is used as the Linux-kernel based OS for the Raspberry Pi. This allows us to run everything from general-purpose programming languages like Python and C to binaries for running blockchain and other storage software and user-developed code. A full, desktop version is also available for Raspbian (Raspbian Stretch). Figure 4.1 describes a Raspberry Pi 3 (Model B) attached to a power supply.

4.2.3 General Purpose I/O

A raspberry pi 3B+ model has 40 General Purpose I/O (GPIO) pins used for different purposes like reading data, providing a voltage of 3v3 or 5V or allowing serial protocols such as I2C.

Sensors are usually transistors and are connected to one of the below pins.

1. GPIO headers for sending or receiving data

2. Ground header

3. 3v3 or 5V pins for maintaining voltage

Figure 4.2 shows the different types of pins available in the raspberry pi required for different purposes.

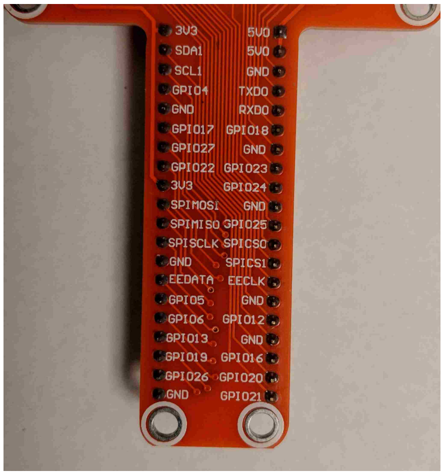


Figure 4.2: GPIO Pins on Raspberry Pi 3B

4.2.4 Digital Humidity and Temperature Sensor

Sensors like Digital Humidity and Temperature Sensor (DHT11) are used to collect telemetry data, temperature and humidity readings in this case, and are read by python scripts running on the raspberry pi using the signals received from GPIO header pins provided on the device.

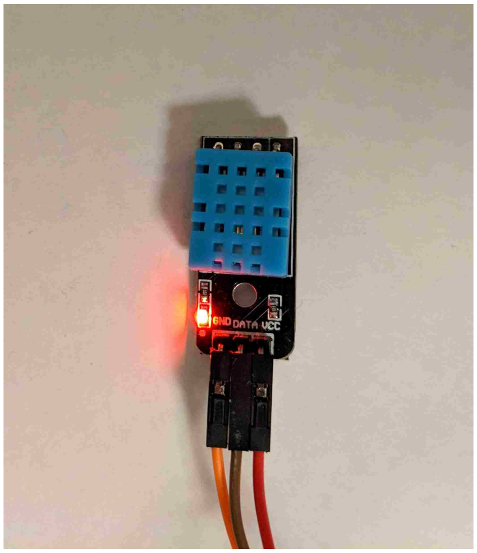


Figure 4.3: DHT11 Sensor

4.2.5 RGB LED

An RGB LED can light up individually in colors of RED, BLUE or GREEN or in groups providing different colors. It can be programmed GPIO modules in C/ Python and are available as part of the Raspbian OS. Supplying power to the GPIO header which is connected to the corresponding colors pin in the LED results in that color being displayed.

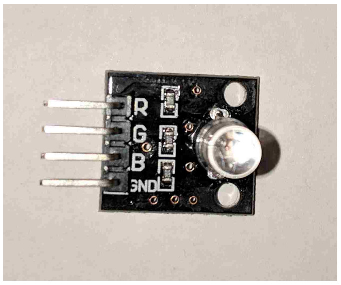


Figure 4.4: RGB LED

4.2.6 I2C LCD

I2C is a serial protocol through which low-powered devices such as an LCD can communicate with the host device (raspberry pi) through the general purpose i/o pins. An LCD with 16x2 display matrix and a connected I2C backpack is used to display the latest data saved to blockchain and is used for demonstration purposes.

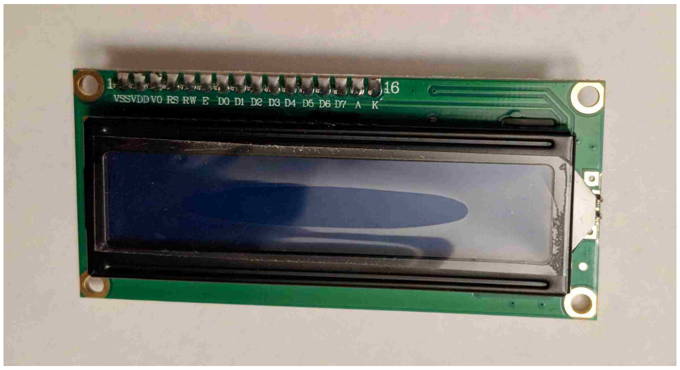


Figure 4.5: I2C LCD

4.2.7 Hardware Security Module

Tiny Edge devices like the Raspberry Pi 3 provide little to no security. The primary storage mechanism is a micro SD card which can be easily imaged and data including encryption keys can be stolen easily as data is stored in plain text. There is also no storage for sensitive data such as secret keys for cloud-based APIs or encryption keys used for authentication. Add-on Hardware Security Modules, also called Trusted Platform Modules (TPM) can be used to overcome some of the security problems posed above. In this thesis, Zymkey 4i was used which is an add-on module for the Raspberry Pi.

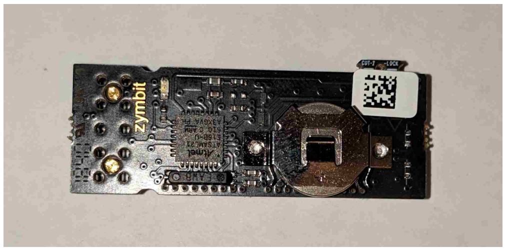


Figure 4.6: Trusted Platform Module

4.3 IoT Security

IoT devices are expected to have a huge economic impact in coming years and are expected to grow exponentially in numbers. This provides a huge incentive for hackers to try and steal sensitive information for sale or other malicious intents. Devices connected to the internet are vulnerable to serious issues if they are not sufficiently protected. Another challenge faced by these edge devices is their weak processing and memory capabilities. We rely on a combination of Public key cryptography, Symmetric encryption, and Hashing to transfer keys securely between devices and encrypt and sign telemetry data.

4.3.1 TPM

Trusted Platform Module (also called TPM or ISO/IEC 11889) [Kim19] is an international standard for a secure micro controller that protects hardware using integrated cryptographic keys. This standard was first developed by Trusted Computing Group (TCG) and was later standardized by International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) in 2009. Current version for the standard is TPM 2.0.

Two TPMs were considered for use with the raspberry pi, the Optiga TPM from Infineon and Zymkey 4i from Zymbit security. Zymkey 4i was ultimately selected for use with the raspberry pi as it had much better support for setup, encryption and setting up LUKS. 4i is the current version of the TPM while 3i and 2i are older versions of the product.

Zymbit 4i is used as the TPM for raspberry pi and is pictured in Fig 4.6 It occupies the first 10 GPIO pins, including one 3v3 pin, 2 5V pins, 2 Ground pins, GP104, and the I2C bus. Before installing it, the I2C interfacing option is first activated on the Raspberry Pi. Finally, the required software modules needed for accessing the hardware keys stored on the TPM device are installed. Once setup is completed, the root file system is encrypted using LUKS and dm-crypt using the private hardware key stored on TPM.

Data Locker keys

Zymkey 4i, provides a public-private ECDSA key pair for signature generation and verification as well as two keys, one-way and shared. One-way key is used to lock down sensitive files such as secret keys to the machine while the shared key is used to encrypt data saved to cloud services such as Azure and Amazon Web Services (AWS).

Linux Unified Key Setup

The storage medium (SD card in this case) is encrypted fully using LUKS [BV16] (Linux Unified Key Setup) and dm-crypt. The dm-crypt implementation uses a single Master key to encrypt/decrypt file system contents shared by several users or services. Moreover, this single master key needs to be changed frequently to avoid being cracked. This might not always be feasible and the alternative, LUKS uses a hierarchical key management system to simplify managing keys for each service/user, providing them access to services and revoking them when required. However, all keys are encrypted using the single master key which must be stored on the device storage. Since the SD card can be removed and accessed easily, master key is at risk of being stolen. The Hardware Security Module/ TPM overcomes this issue by locking the LUKS Master key using an encryption key stored within the hardware.

Chapter 5

Proposed System

5.1 System Architecture

The proposed system explained in 5.1 consists of a number of applications running on a Local machine (miner nodes) whose sole purpose is to validate transactions called by set methods in smart contract. One of the raspberry pis called IoT Producer will be used to gather readings. The IoT producer runs geth in light mode, swarm, and ipfs to send data to rest of the network. The other raspberry pi called IoT consumer is used to get data from the network and send them to other services, APIs or LCD displays. Networking between geth, ipfs and swarm nodes occur in a decentralized and distributed manner.

Figure 5.1 explains the different components used to build the proposed system.

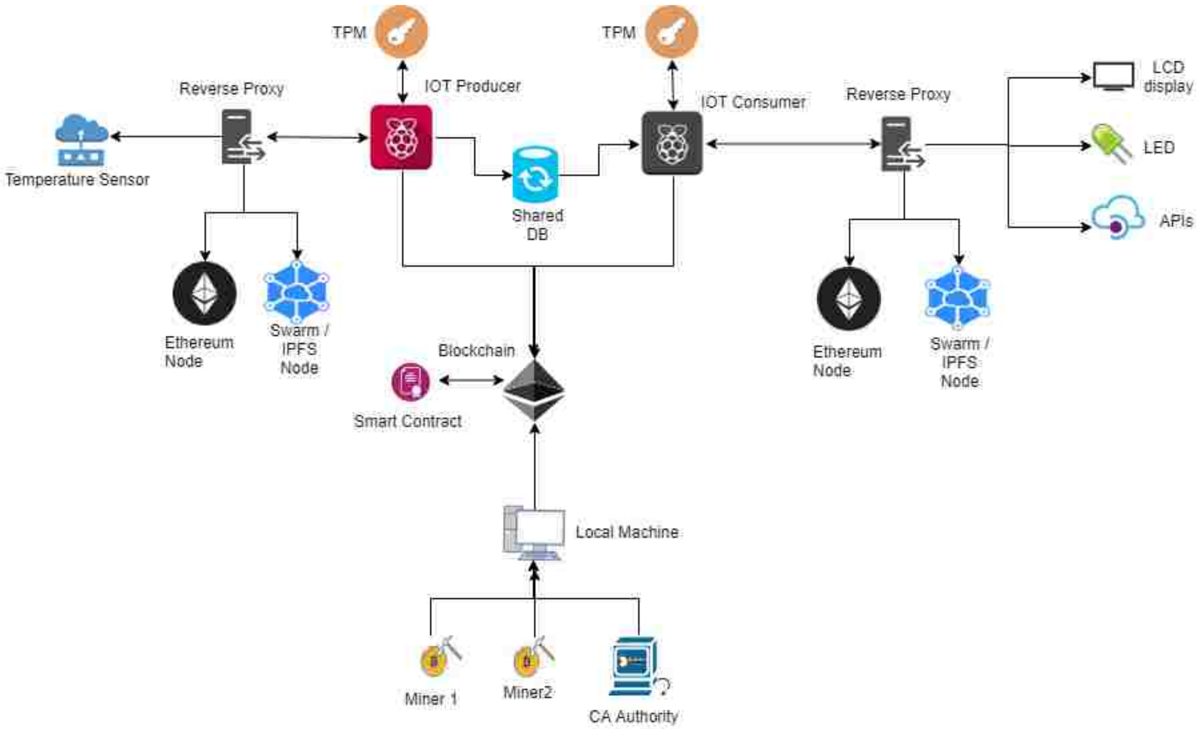


Figure 5.1: Implementation Diagram

5.1.1 Local Machine

The test machine validating mining operations is a laptop with Core i7-7700HQ processor rated at 2.8GHz with 4 physical cores and 16 GB RAM. This machine runs two geth miner nodes (for redundancy). One of the miner nodes dedicates a percentage (25%) of its resources to serve requests from light geth nodes.

5.1.2 IoT Edge Nodes

Two identical Raspberry Pis are set up and each one serves different purposes. One of the raspberry pi's is used to read data from sensors while the other is used to output the sensor data to a display device or to an external API.

The raspberry pi model used is Pi 3 Model B with a 1.2 GHz quad-core ARM Cortex A53 processor and 1 GB DDR2 RAM. A 32GB class 10 SD card encrypted by LUKS and dm-crypt serves as the storage medium. Raspbian Stretch Lite is installed on devices.

IoT Producer

The IoT Producer is used to read data from all the sensors, encrypt and store data in Proposed network. Geth node running in this machine is a light node which depends on an external mining node to get full information about blocks in the chain. Ethereum is merely used to store references to data collected by the system. It is set up to be able to connect to instances of swarm or IPFS, or even run its own instances to store the actual data in encrypted form.

For testing and demo purposes, a Digital Humidity and Temperature Sensor (DHT11) is attached to the raspberry pi to GPIO22 pin for data, 3v3 for voltage and ground pin.

IoT Consumer

The IoT Consumer is used for demonstration purposes in this thesis. Like the IoT Producer, it connects to an external swarm or ipfs or runs its own instances along with a light version of geth. It retrieves the required reference from Ethereum blockchain and requests the referred content from swarm or IPFS.

Two IoT devices are connected to the IoT Consumer for testing and demo purposes. One is a 3 color LED which is connected to GPIO4, GPIO3 and GPIO2 pins of the consumer and a ground pin. The second device is an I2C LCD which is connected to the SDA1 and SCL1 pins for I2C bus, 3v3 for power and ground pin.

5.2 Security Algorithm

We use a couple of cryptography techniques to transfer secret keys, encrypt data using these keys as well as hashing methods to sign messages and verify them.

5.2.1 Cryptography Algorithms used

Public Key Cryptography

Public Key Cryptography uses a key pair of public and private keys to encrypt data as well as provide verification of the authenticity of a message.

Ron Rivest, Adi Shamir, and Leonard Adleman developed the RSA algorithm which utilizes a public-private key pair to encrypt and decrypt data. It can also be used for verifying signatures. The public key of a recipient is used to encrypt data on a machine which generates the data which can then be used by the recipient to decrypt the data using their private key. Due to payload size constraints, it is typically used to transfer symmetric encryption keys.

2048 bit Public-Private key pairs are generated by all the Raspberry Pis on the network and the public keys are sent to the host machine.

Equation 5.1 describes the encryption of Symmetric Key using the Public Key of the Raspberry Pi.

$$Cipher = E(K_Pi, PlainText) \tag{5.1}$$

Equation 5.2 describes transfer of Encrypted Symmetric Key from host to the pi.

$$Host \Rightarrow Pi : \{SymmetricKey\}_{K_Pi} \tag{5.2}$$

Equation 5.3 describes decryption of Symmetric Key using Private Key of raspberry pi.

$$PlainText = D(K^{-1}{}_Pi, Cipher) \tag{5.3}$$

In this thesis, we encrypt the received Symmetric Key using TPM's AES key and decrypt it only when required. This is described in Equation 5.4.

$$CipherTPM = E(K_{tpm}, PlainText) \tag{5.4}$$

Symmetric Encryption

Symmetric Encryption uses a single secret key to encrypt and decrypt data. Symmetric key encryption is used to encrypt or decrypt huge volumes of data and is generally faster than Public Key Cryptography. The downside is that the secret key must be stored safely and changed frequently.

Advanced Encryption Standard (AES) utilizes a strong secret key to encrypt actual data. We use this algorithm to encrypt and store data in Swarm or IPFS. Since the proposed system requires many devices sharing same key, using public key cryptography was considered for this thesis.

The Host machine generates a 256 bit AES key and sends it to all the raspberry pi machines connected to the network.

The raspberry pis encrypt and store these secret keys on their file systems and use them to send their Payload in encrypted format to Storage systems.

The IoTProducers encrypt and send data as shown in 5.5

$$Cipher = E(K_{aesSecret}, PlainText) \tag{5.5}$$

The IoTConsumers decrypt and use data as shown in 5.6

$$PlainText = D(K_{aesSecret}, Cipher) \tag{5.6}$$

Hashing

Hashing is the process of generating a set of characters that can uniquely identify a particular input. In Cryptography, they are used mainly to generate a signature for a larger input and encrypt with a sender's private key which can be decrypted only using sender's public key, thus ensuring the sender's authenticity. Secure Hashing Algorithms (SHA series), Message Digest (MD series) and Hash Message Authentication Code (HMAC) are some of the hashing algorithms in popular use.

A hash algorithm (HMAC) is used in this thesis to sign data on producer side and verify its authenticity on the receiving end. It uses a secret shared key for authentication and signature verification which can be done using the same method used for transferring the AES key.

Equation 5.7 describes the signature generation process using shared secret key, cipher and Initialization vector generated from the AES encryption process and SHA256 algorithm.

$$SignatureGeneration = HMAC(K_{secret}, cipher + iv, sha256) \tag{5.7}$$

Equation 5.8 describes the verification process by comparing stored signature and generated MAC code on receiving side.

$$SignatureVerify = HMAC(K_{secret}, \{decryptedCipher\} + \{decryptedIV\}, sha256) \tag{5.8}$$

5.2.2 Private PKI

Public Key Infrastructure (PKI) is a formal structure that constitutes policies, and procedures needed to manage, and distribute digital certificates and help in public-key encryption. Its primary purpose is to facilitate secure electronic information transmission. PKI consists of a Certificate Authority, Registration authority, and Central directory.

PKI is used to authenticate users of a service and prevent man in the middle attacks in networks. A PKI involves using an external Third party which is universally trusted to sign Digital Certificates in X.509 format. SSL/ TLS use this format for digital certificates which in turn, is basis for the HTTPS standard.

Private PKI is generally used to secure communications between local devices. We have a use case in our thesis where we must connect to the IoT Consumer node which runs the geth and swarm endpoints. Outside APIs or other devices such as another Raspberry Pi which has an LCD device attached to it (called LCD node) and is used to display the current temperature and humidity readings. To prevent unauthorized access and to encrypt data between the IoT Consumer node and Display node, we set up a private PKI to sign digital certificates and route all communications through a reverse proxy. The PKI setup used in this thesis is pictured in Figure 5.2

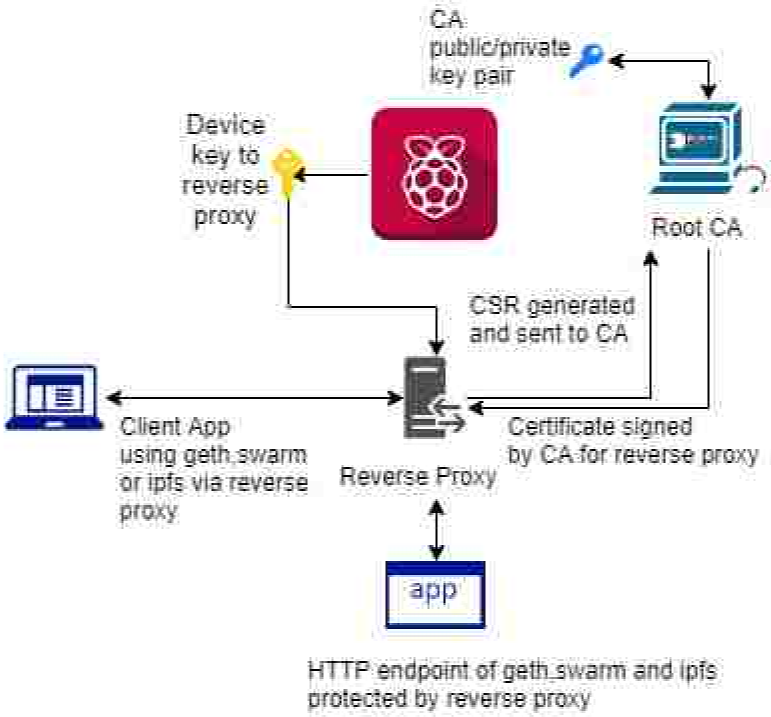

CA public/private key pair
Device key to reverse proxy
Root CA
CSR generated and sent to CA
Client App using geth,swarm or ipfs via reverse proxy
Reverse Proxy
Certificate signed by CA for reverse proxy
app
HTTP endpoint of geth,swarm and ipfs protected by reverse proxy

Figure 5.2: PKI architecture diagram

1. Generate key for private Root CA and store it securely.

2. Generate digital certificate for our Root CA and self-sign it with its own key.

3. On the IoT Consumer node, generate a device key or use TPM key.

4. Generate a Certificate Signing Request (CSR) for the device.

5. Send CSR to Root CA for signing.

6. Root CA signs and returns digital certificate using its own private key.

7. Device key and certificate are used to configure Nginx reverse proxy server to secure geth, swarm and ipfs endpoints.

Reverse Proxy

A reverse proxy is an intermediate server that intercepts all or most requests depending on their usage. These intercepted requests are then routed to the correct backend server. In our case, we run a reverse proxy on the IoT Consumer which uses certificates signed by our private PKI and allows HTTPS communication between our LCD node and the IoT Consumer. In this way, LCD node is allowed to connect to geth, swarm, and ipfs endpoints and receive data securely. Nginx is a popular server option and can be configured to be used as a reverse proxy.

5.2.3 Hardware Security with TPM

TPM is used to help enhance the relatively weak security on a base Raspberry Pi. It provides a hardware level root-of-trust in a system by using a unique element of the system which cannot be replicated by anyone trying to do so. It is generally used to protect secret keys stored in the device as well as fully encrypt the file system and decrypt only when the device is attached to the system. Zymkey 4i (TPM) communicates with the raspberry pi via the I2C interface.

TPM typically provide the following functions.

1. Random number generator
2. Generation or Locking/Unlocking of cryptographic keys
3. Full Disk Encryption

The first step in setting up TPM is to pair it with Raspberry Pi. This pairing process uses a couple of unique features from both the host raspberry pi and TPM and generates a unique Device ID that binds TPM to the host. A Linux systemd service zkifc runs on host raspberry pi when properly paired. It is used to provide access to keys stored on the TPM via I2C interface and all communications between TPM and I2C are encrypted. Once binding is completed, all applications are installed and is considered ready for deployment, this TPM can be set to a permanent binding mode which locks the TPM to the raspberry pi and cannot be bound to other raspberry pis in the future. This permanently disables the raspberry pi if it is tampered or if the TPM is removed.

Figure 5.3 shows the different features offered by a typical TPM.

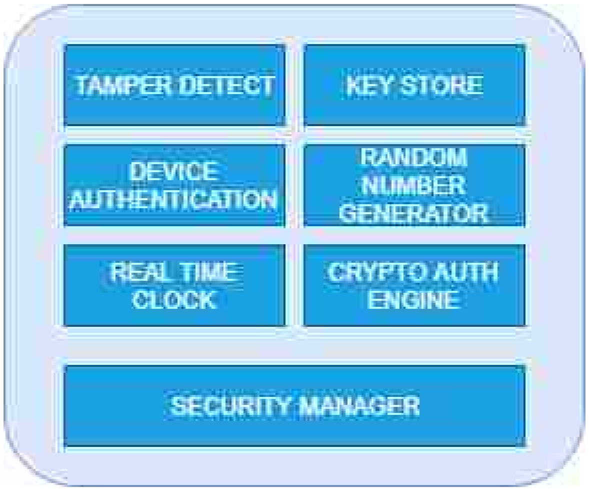


Figure 5.3: TPM Block diagram

Among other features, TPMs generally provides two important features that were extremely useful for this thesis.

5.2.4 Root File encryption with LUKS

The Root File System is not very secure in a Raspberry Pi. The SD card can be easily imaged and sensitive data like shared keys can be stolen very easily. A viable solution is to encrypt the entire Root File System using Linux Unified Key Setup and dm-crypt.

Figure 5.4 shows how easily the SD card can be read and sensitive data can be stolen.

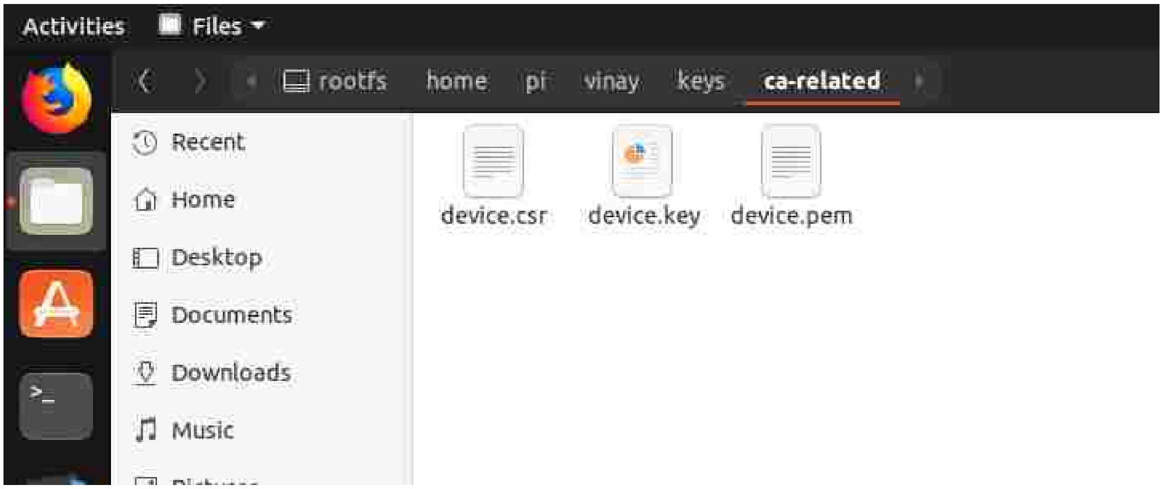


Figure 5.4: Unencrypted File System

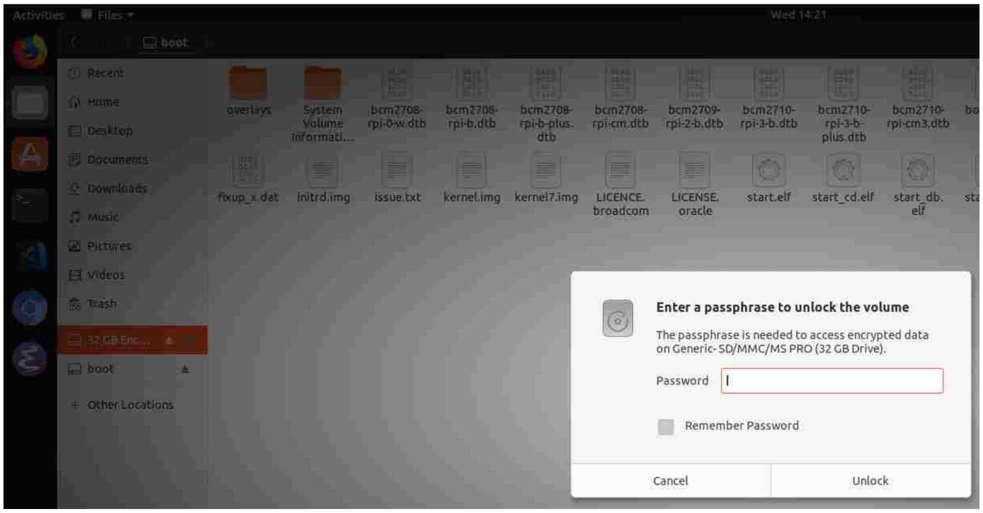
Boot Sequence of normal LUKS

1. Kernel initializes initramfs - A single archive loaded into main memory during startup

2. initramfs presents decryption key to LUKS

3. LUKS decrypts the root file system

There are a couple of issues with above procedure when Raspberry Pi is involved. First, there is no dedicated key-ring for storing cryptographic keys available in other Operating Systems. Secondly, sensitive keys should not be stored in plain text on the removable SD card. To fix such issues, a TPM is used with LUKS [Sco17] for a more secure boot sequence. Figure 5.5 shows an encrypted file system.



Enter a passphrase to unlock the volume

The passphrase is needed to access encrypted data on Generic- SD/MMC/MS PRO (32 GB Drive).

Password

□ Remember Password

Figure 5.5: Encrypted File System with LUKS

Boot Sequence of LUKS with TPM

1. Kernel initializes initramfs

2. initramfs present locked LUKS key to TPM

3. TPM verifies the key signature and decrypts the LUKS key

4. Unlocked LUKS key is used to decrypt the root file system

Figure 5.6 shows how root file system is partitioned into an encrypted block and how master keys and user keys are protected by TPM.

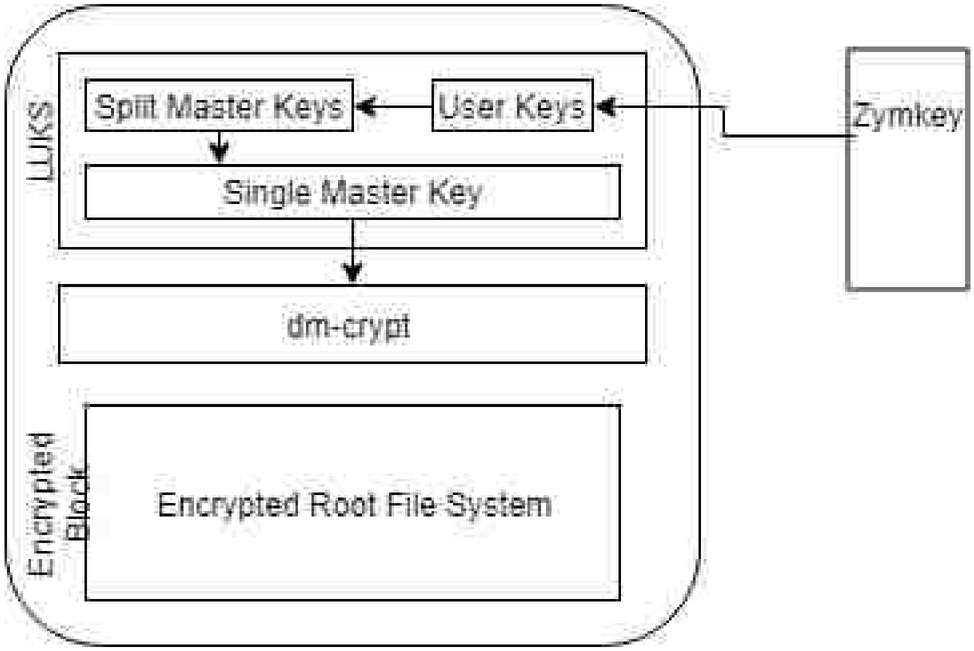


Figure 5.6: TPM enabling LUKS encryption

Data Locker for keys

This feature allows to store all shared secret keys in encrypted form. The secret keys transferred over to the raspberry pis (the process is explained in 5.2.5) are encrypted using the AES key hard-coded to TPM hardware. Zymkey 4i provides a python module called zymkey which accesses the TPM's AES key and encrypts or decrypts the secret key and locks it to the raspberry pi. The locked key is then used for transferring encrypted data over the network.

The TPM's AES key is used to encrypt and decrypt the actual AES secret key and is explained in equation 5.9 and equation 5.10.

$$\{zymkeyEncryptedKey\} = E(K_{zymkey}, K_{aesSecret}) \tag{5.9}$$

$$SecretKey = D(K_{zymkey}, K_{aesSecret}) \tag{5.10}$$

We ensure that the keys are never stored in plain sight and can never be accessed by unauthorized users even if the raspberry pi is stolen and the encrypted root file is successfully decrypted.

5.2.5 Software Security

The raspberry pi can be secured using a number of simple methods, each having an equally significant impact on improving overall security of the device.

Change default password

The default password on the raspberry pi is raspberry. According to a research [Whi18], IoT devices are easily hacked due to improper setup, often times leaving the default credentials in place. The first priority after setting up a raspberry pi must be to change the default password to something more secure.

Replace default user

To further increase security, a new user is added and the default pi is deleted. Another feature to consider is to request password when a terminal runs sudo command which is turned off in raspbian by default.

Installing Firewall

There is no firewall installed by default on the raspberry pi. Uncomplicated Firewall (ufw) can be set up very easily in any Linux based OS and the raspbian OS is no different. All ports except for those we need such those for making remote procedure calls, networking between geth, swarm or ipfs nodes are selectively enabled.

Transferring Symmetric keys

Symmetric (Secret) keys must never be transferred in plain-text format over the network or stored in the file system in plain text. The below sequence of steps is used to transfer AES keys over a network and lock them to the raspberry pi.

1. Generate RSA public-private key pair on Raspberry Pi

2. Transfer public key over to host machine generating the symmetric key using a secure channel such as Secure Copy (SCP) or Secure File Transfer Protocol (sftp)

3. Generate a signature hash for the key
4. Encrypt the symmetric key with raspberry pi's public key
5. Transfer the encrypted symmetric key back to raspberry pi
6. Decrypt the symmetric key using raspberry pi's private key
7. Verify the signature stored in decrypted text
8. Once the signature is verified, use the one-way key from TPM to encrypt and store the symmetric key, which is only decrypted and used when data collection script starts

Disable RPC on geth and swarm if not required

Remote Procedure Calls are used by remote clients to connect to geth/swarm nodes and receive or send information to them. However, our storage application runs on the same node as IoT Producer and we do not need any external client access to connect to the IoT Producer. Geth and Swarm must instead use IPC endpoint, which enforces only calls from the localhost for security purposes.

5.3 Software Design

This section describes the sequence of steps performed on sending and receiving ends of the system and also explains the process of developing a Smart Contract that will be used to register devices and the methods that will be used to store data in Blockchain as well as IPFS and Swarm.

5.3.1 Sequence of Steps

The steps to collect and retrieve the sensor data after applying encryption, verification or decryption are explained in the sections below. Figure 5.7 describes these steps at a very basic level.

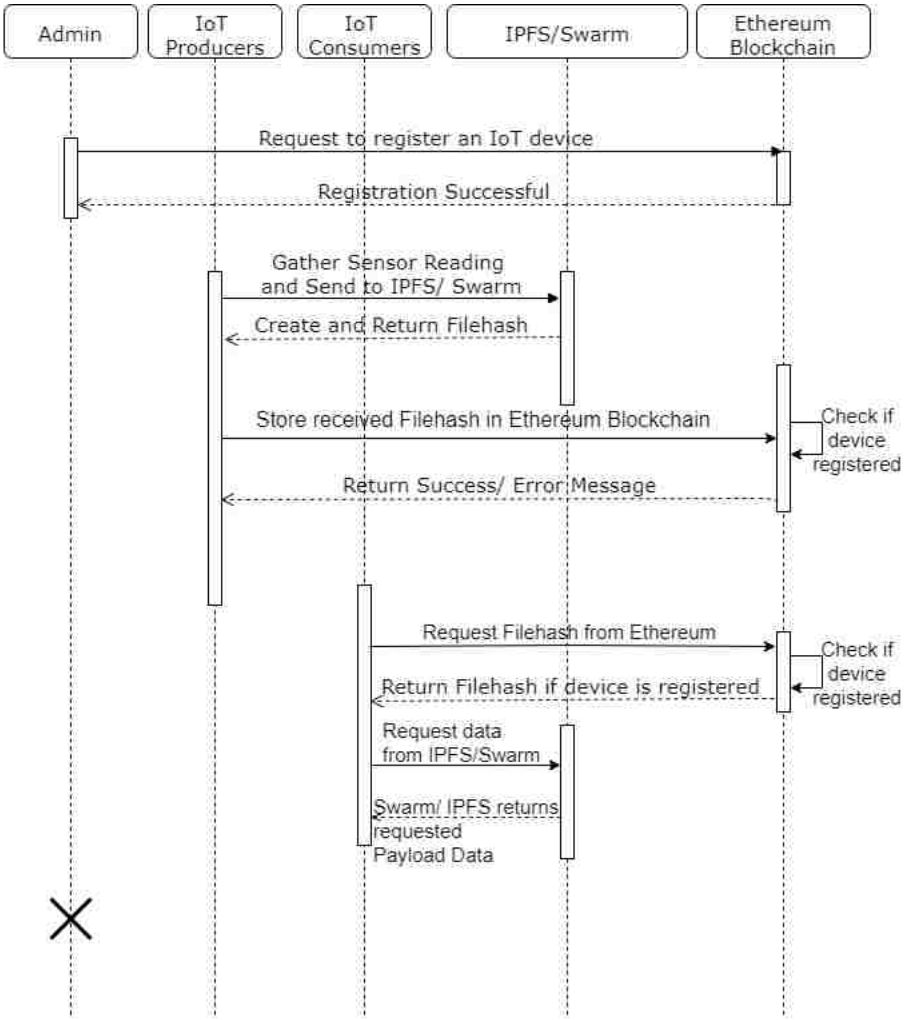

Admin
IoT Producers
IoT Consumers
IPFS/Swarm
Ethereum Blockchain
Request to register an IoT device
Registration Successful
Gather Sensor Reading and Send to IPFS/ Swarm
Create and Return Filehash
Store received Filehash in Ethereum Blockchain
Check if device registered
Return Success/ Error Message
Request Filehash from Ethereum
Check if device registered
Return Filehash if device is registered
Request data from IPFS/Swarm
Swarm/ IPFS returns requested Payload Data

Figure 5.7: Sequence Diagram

5.3.2 Sensor Data Collection

Algorithm 1 explains the overall steps followed on the Producer side to collect data from sensor, sign and encrypt the generated payload. Data is then saved to IPFS or Swarm and the resulting file hash is stored in Ethereum.

Algorithm 1: Data Collection and Payload Storage

```
Data: TPM Private key tpmPrivKey, Encrypted AES key cipherText
Result: Collect and Encrypt Payload, Store in Swarm or IPFS
secretKeyAES = decrypt(tpmPrivKey,cipherText);
//Run indefinitely
while True do
    temperature, humidity = dataCollectionFromSensor();
    timestamp = getCurrentTime();
    payload = buildPayload(temperature,humidity,timestamp);
    signature = generateSHA256(payload);
    payloadCipher = encrypt(secretKeyAES,payload,signature);
    filehash = setSwarmOrIPFSData(payloadCipher);
    callSetFileHashSensorContract(filehash);
end
```

Algorithm 2: Data Retrieval and Use

```
Data: TPM Private key tpmPrivKey, Encrypted AES key cipherText
Result: Retrieve Encrypted Payload from Swarm or IPFS
secretKeyAES = decrypt(tpmPrivKey,cipherText);
//Run indefinitely
while True do
    filehash = callGetLatestFileHash();
    payloadCipherText = getSwarmOrIPFSData(filehash);
    payloadPlainText, signature = decrypt(secretKeyAES, payloadCipherText);
    if generateSHA256(payloadPlainText) == signature then
        Send to other sources;
    else
        Throw Signature Not Verified Error;
    end
end
```

5.3.3 Sensor Data Retrieval

Algorithm 2 explains steps followed on the Consumer side to retrieve file-hashes from Ethereum, fetching encrypted payload from Swarm or IPFS and finally decrypting and verifying that payload.

5.3.4 Smart Contract Logic Functions

This section details logic used in Smart Contract to define access control, storage and retrieval if file-hashes for payload loaded to IPFS or Swarm.

struct to store the collected data

Although only file hash is part of the struct, it was left in place for future enhancements, if needed.

```
//struct to store the file-hashes which store the actual sensor data
struct SensorData{
    //filehash is pointer to data stored in swarm
    string filehash;
}
```

A very early version of the Smart Contract used the below struct data-type.

```
//struct to store the temperature, humidity and time
struct SensorData{
    uint64 temperature;
    uint64 humidity;
    string dataStorageTime;
}
```

pointer to data

An integer value, ID is used as pointer to refer to our current readings stored/ retrieved by smart contract.

```
uint public currentID;
```

Mapping from pointer to struct and addresses registered

A Mapping is defined as sensorStore from ID to SensorData struct. Another mapping stores registered addresses of IoT devices allowed to write or read data to the system.

```
//map the struct to an id
mapping(uint => SensorData) sensorStore;

//store registered addresses in mapping
mapping (address => bool) private trustedAddresses;
```

Constructor to initialize the ID

The ID must be initialized before access and so a constructor is used. 1 is chosen as the start index. Another variable, createdBy is used to store the contract owner and will be responsible for registering and de-registering IoT devices to the system.

```
//Contract created by
address private createdBy;
//Constructor for the Smart Contract
constructor() public {
    currentID = 1;
    createdBy = msg.sender;
}
```

Modifier to only allow owner of Smart Contract to register and de-register ids

Access to certain Contract methods such as registering or de-registering devices must be held only by administrator. In this case, the contract creator is designated as admin and so only the address that creates the contract is allowed to run methods which enforce the ownerOnly modifier.

```
//Modify some functions to be executed only by the Contract creator
modifier ownerOnly {
    require(msg.sender == createdBy);
    _;
}
```

Events to log Registration and Changing State

As explained earlier in Chapter 2, transactions aren't capable of returning an output value. So we use events that are emitted in response to certain conditions that occur while a transaction is being executed.

```
//event after registration/de-registration
event deviceEvent(
    address indexed _from,
    string _message
);

//event after registration/de-registration
event setFileHashEvent(
    address indexed _from,
    string _message
);
```

Function to get the current ID

This method describes the process of returning the currentID used to load data to the system.

```
//function to get the currentID
function getCurrentID() public view returns(uint){
    return currentID;
}
```

Function to increment the current ID

This method describes how to increment the currentID being used to load data to the system.

```
//function to increment the ID used internally for managing file-hashes
function incrCurrentID() public{
    currentID++;
}
```

Function to set the SensorData filehash

This method is used to initiate a transaction to add a new file hash received from the storage system and save it to the mapping we have declared earlier. If the device is not registered, an event with the appropriate message is emitted. On success, another event is triggered which relays that the transaction was successful to all the listeners.

```
//Function to store the filehash from swarm
function setSensorData(string _filehash) public {
    if(devicePresent(msg.sender)) {
        uint idToStore = getCurrentID();
        SensorData storage sensorReadings = sensorStore[idToStore];

        sensorReadings.filehash = _filehash;

        incrCurrentID();

        emit setFileHashEvent(msg.sender, ”FILEHASH TXN CALLED”);
    } else {
        emit setFileHashEvent(msg.sender, ”DEVICE NOT REGISTERED”);
    }
}
```

Function to get latest SensorData filehash

This function returns latest filehash stored in blockchain. Using currentID variable declared earlier, we retrieve value of currentID from sensorStore mapping.

```
//Function to get latest stored data
function getSensorDataLatest() public view returns (string){
if(devicePresent(msg.sender)) {
    SensorData storage sensorReadings = sensorStore[getCurrentID()-1];
    return sensorReadings.filehash;
}
return ”DEVICE NOT REGISTERED”;
}
```

Function to get SensorData file hash from any ID

This method works in a similar fashion to the above method but retrieves value associated with ID passed to the function.

```
//Function to get data stored under some ID
function getSensorDataByID(uint ID) public view returns (string){
    if(devicePresent(msg.sender)) {
        SensorData storage sensorReadings = sensorStore[ID];

        return sensorReadings.filehash;
    }
    return ”DEVICE NOT REGISTERED”;
}
```

Methods to register and deregister IoT devices

The following methods explain how to check if the device is registered and call register or deregister methods for any device and can be only accessed by the contract owner. The method ”devicePresent” returns if device is registered with the system. The methods ”registerDevice” and ”deregisterDevice” are used to register and de-register the IoT device using the Smart Contract respectively.

```
//function to check if device is registered
function devicePresent(address addressToAdd)
public view returns (bool) {
        return trustedAddresses[addressToAdd];
}

//register IoT device
function registerDevice(address addressToAdd) ownerOnly public {
    if(devicePresent(addressToAdd)) {
        emit deviceEvent(addressToAdd, ”DEVICE ALREADY REGISTERED”);
    }
        trustedAddresses[addressToAdd] = true;
        emit deviceEvent(addressToAdd, ”SUCESSFULLY REGISTERED”);
}


//deregister IoT device
function deregisterDevice(address addressToRemove) ownerOnly public {
    if(!devicePresent(addressToRemove)) {
        emit deviceEvent(addressToRemove, ”DEVICE NOT REGISTERED”);
    }
        trustedAddresses[addressToRemove] = false;
        emit deviceEvent(addressToRemove, ”SUCESSFULLY DEREGISTERED”);
}
```

Algorithm 3: Payload Storage in IPFS or Swarm

```
**Data:** Encrypted and Signed Payload $encryptedPayload$
**Result:** Return File hash $filehash$ from IPFS or Swarm
result = sendHTTPPostRequest(”HTTPProxyEndpoint”, data=$encryptedPayload$,
  headers=’Content-Type’: ’text/plain’);
filehash = result.text;
```

Algorithm 4: Payload Retrieval from IPFS or Swarm

```
**Data:** File hash of Payload $filehash$
**Result:** Return Encrypted and Signed Payload $encryptedPayload$
result = sendHttpGETRequest(”HTTPProxyEndpoint” + $filehash$);
$encryptedPayload$ = result.text;
```

5.3.5 IPFS or Swarm Logic

The provided HTTP endpoints are used in both IPFS and Swarm to store and retrieve the payload. The requests module in python is used to send POST and GET requests from the storage systems. Storing the data then boils down to a simple POST request as explained in Algorithm 3.

A GET request retrieves the payload from the system using the file hash as input which is explained in Algorithm 4.

5.3.6 Payload Format

The payload stored in the storage systems can be modified at will and is independent of the data format enforced by blockchain. It also allows data to be encrypted before storage. The payload is generated and stored in JSON format as referenced in Table 5.1. All values are stored as strings and then formatted into either integer, floating-point numbers or timestamp when needed.

Table 5.1: Payload Format

S.No	Payload Item	Brief Explanation	Example Data
1	Temperature	Temperature collected from sensor in C	23
2	Humidity	Humidity collection from the sensor in %	44
3	Temperature Units	Units used for temperature (C or F)	F
4	Humidity Units	Units used for humidity	%
6	Timestamp	Time stamp of data collection	2019-03-13 22:06:09
6	Device ID	IoT device ID	IoTProducer1
7	Device Type	IoT device type	RaspberryPi3 B
8	Device IP	IP of IoT device	192.168.0.16
9	Sensor Type	Type of the Sensor Used	DHT11

5.4 Implementation

This section describes environment setup, binaries, frameworks and modules installed to develop the proposed system.

5.4.1 Environment Setup

Python 3 is used to write scripts that interface with IoT devices, Ethereum blockchain, IPFS, and Swarm. Its package manager pip3 is installed in all the devices we use. The environment used in our test devices is tabulated in 5.2. Truffle (Node.js app) is used to build, compile and deploy Solidity based Smart Contracts.

Table 5.2: Environment Setup

Device	OS	Architecture	Python Version	Node.js	Solidity
LocalMachine	Ubuntu 18.10	amd64	Python 3.5	Yes	Yes, 0.4.24
IoTProducer	Raspbian Stretch Lite	armv7	Python 3.5	No	No
IoTConsumer	Raspbian S tretch Lite	armv7	Python 3.5	No	No

5.4.2 Software binaries

Pre-built Geth binaries are downloaded for local machine(amd64) as well as the IoT nodes(arm). To ensure uniformity, same versions are used across all devices. Ethereum-Swarm packages are readily available for Ubuntu but not for the Raspberry Pi 3.

Raspberry pi 3B doesn't have the hardware to build these binaries from Golang source code. Fortunately, the Go programming language provides excellent cross-compilation tools. The local machine is used to build the swarm binary and were installed on the raspberry pis.

Table 5.3 provides information on different versions of geth, swarm, and IPFS used in this thesis.

Table 5.3: Software binary versions

Device	**Geth**	**Swarm**	**IPFS**
LocalMachine	1.8.23-stable	0.3.11-stable	0.4.19
IoTProducer	1.8.23-stable	1.6.7-stable	0.4.19
IoTConsumer	1.8.23-stable	1.6.7-stable	0.4.19

5.4.3 Python modules

Since python 3 is used extensively, a couple of third-party python modules are used to help access the blockchain, write data to ipfs or encrypt and decrypt data. Table 5.4 describes all the python modules used in this thesis.

Table 5.4: Third party Python modules

Module Category	**Module Name**	**Usage**
Storage	web3	To access smart contract functions or get balances
Storage	ipfsapi	To store and retrieve data from IPFS
IoT	AdafruitDHT	Get temperature, humidity from DHT
IoT	Rpi.GPIO	Control GPIO pins connected to RGB LED
IoT	I2C LCD driver	Display data on LCD via I2C interface
Cryptography	pycryptodomex	Implementations of Cryptography algorithms
Cryptography	zymkey	Encrypt, Decrypt, Sign and Verify data using TPM

5.4.4 Evaluation Methods

Once the network is set up, performance of the system will be tested in terms of Transactions per Second (TPS), cost per transaction, CPU and memory usage. To test TPS, a data set of 5,000 records formatted in the Payload data format described in 5.3.6 is generated.

Three categories of tests are performed with both Ethereum and Swarm/ IPFS.

1. Data is stored to Swarm/ IPFS and write timings are measured. File hashes are stored in a text file and used while testing read performance.

2. Data is stored to Swarm/ IPFS and the received file hash is stored to Ethereum via the Smart contract function. Read and Write times are measured.

3. Data is stored to Swarm/IPFS in an encrypted format. The retrieved file hash is then stored and retrieved when needed. Encryption Algorithm used for all tests is AES-CBC with PKCS padding. The key used is of 256-bit size.

Setup

A private network consisting of at least 3 nodes needs to be set up - a ”full” miner node with IoT producer and IoT consumer ”light” nodes.

These steps outline the process of setting up a private Ethereum network.

1. Initialize a data directory which will contain chain data with our genesis.json file.

2. Create a new account for all the nodes.

3. Assign the miner node to be a static node, which is trusted and any other node can connect to it.

4. Save the node info (admin.nodeInfo.enode) of the static node on all nodes using a static-nodes.json file.

5. Start all nodes with same Network ID. The startup script must contain the nodiscover flag to prevent other peers from trying to connect to the network.

6. Start mining on the miner node and verify other nodes are able to synchronize. The mining process does not start until the Directed Acyclic Graph is generated.

7. Wait for a few minutes after mining starts so that the miner earns some Ether.

8. Transfer some of the mined ether to the admin node.

9. Compile smart contract and deploy it from the admin.

10. Register the nodes that are authorized to send and receive data from the network using the admin.

Testing

To test Ethereum, the primary parameters can be measured for both file hash and payload. The primary accounts in all the nodes of the system are transferred 100 Ethereum each to perform these experiments.

1. Cost incurred per transaction

2. Time taken to transact Set Data method

3. Time taken to retrieve per record

4. CPU and RAM usage

5. Average block creation time

6. Average Network Hashrate

The network can be fine-tuned to include more transactions per block by adjusting the gas limit per transaction.

5.4.5 IPFS and Swarm

Testing IPFS and Swarm can be done using similar methods. The below parameters are considered during testing their individual performance on the proposed system.

Setup Swarm Private Network

Swarm is closely related to Ethereum and must use the same network ID that was used in Ethereum to set up a private network.

The following steps can be followed to setup Swarm for our use.

1. Export an environment variable, BZZKEY containing wallet address of the geth node on which swarm is being set up.

2. Create a trusted boot node to which other nodes called peers can connect. One of the miner nodes is entrusted to function as the boot node.

3. Start the swarm node using the same network ID that was used for Ethereum and the geth.ipc file to connect to the Ethereum Name Space (ENS service for distinguishing peers). The startup script must contain the nodiscover flag to prevent other peers from trying to connect to the network.

4. Verify whether other peers are able to connect using swarm console. (admin.peers lists all the peers that are currently connected to the node)

Setup IPFS Private Network

A private network can be set up in IPFS by removing all the existing bootstrap nodes which help peers to find each other. An authentication key is generated and distributed among all the nodes in the network.

These steps describe how to set up a private IPFS network.

1. Initialize all nodes with custom data directories. This process generates a 2048-bit RSA key pair and a peer identity (useful to connect to other peers)

2. Generate an ipfs-swarm.key which will be used for authentication and allow access to the network.

3. Remove all existing bootstrap nodes.

4. Add only generated peer identities to nodes as bootstrap nodes.

5. Start all IPFS daemons and test if files stored in one node are able to be read from other nodes.

Testing

1. Time taken to insert one payload record

2. Time taken to retrieve one payload record

3. CPU and RAM usage

Chapter 6

Performance and Cost Analysis

We will test both costs incurred by the account submitting the transactions and time taken to load and read these records with and without encryption.

6.1 Cost Incurred by Smart Contract Deployment

The cost incurred by the Smart contracts depends on a number of factors. There is a flat fee involved and fees for computation as well as storage must be paid. The fee we pay for deployment depends on the optimization done to a certain degree.

Truffle requires deployment of two contracts, a migration contract, and the actual SensorContract. Migration contract identifies the owner of Smart Contract to be deployed and the timestamp of last successful deployment. SensorContract consists of the code related to registration, writing and reading data from blockchain.

20 gwei per unit of spent gas must be paid for deploying the Smart Contract in private blockchains. This cost is only tested in case we want to deploy the same contract to the main Ethereum network.

Total cost to deploy the smart contracts was 0.0232 Eth. At the time of writing, Eth to dollar conversion rate was 182.37 which is an approximate dollar cost of $4.2

```
Gas Spent for MigrationsContract =  277462
Gas Price = 20 gwei
Total Cost for MigrationsContract = 277462 * 20
                                  = 5549240 gwei
                                  = 0.0055 Eth
```

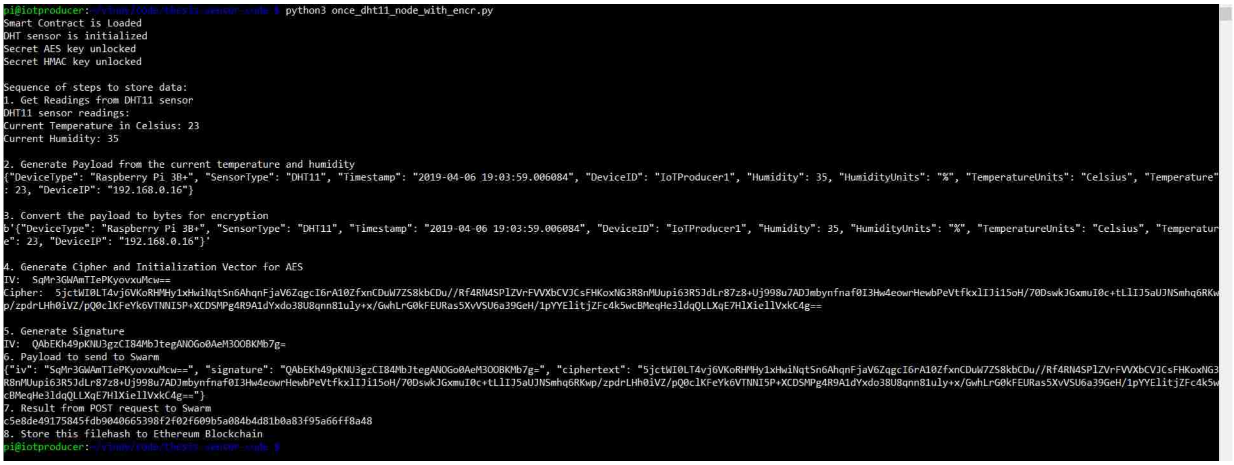
$$
\begin{aligned}
&\text{Gas Spent for SensorContract} = 884293\\
&\text{Gas Price} = 20 \text{ gwei}\\
&\text{Total Cost for MigrationsContract} = 884293 * 20\\
&\qquad = 17685860 \text{ gwei}\\
&\qquad = 0.0176 \text{ Eth}
\end{aligned}
$$

$$
\begin{aligned}
\text{Total Cost for deployment} &= 0.0055 + 0.0176\\
&= 0.0232 \text{ Eth}
\end{aligned}
$$

6.2 Storing and Retrieving data from Proposed System

Once the Contract is deployed and devices using the system are registered in the Smart Contract, we are able to store data in the system.

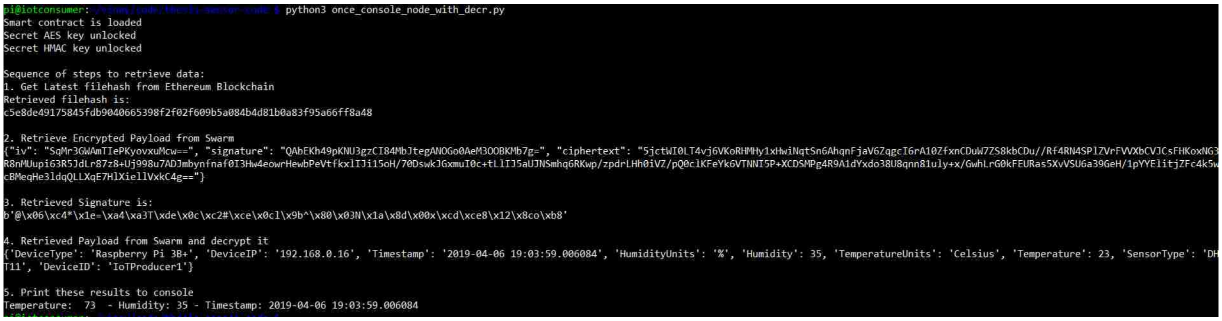
Figure 6.1 describes implementation of the proposed system which gets the sensor data from the connected DHT11 sensor, generates a payload and encrypts it and then stores the payload in Swarm. The request to store payload in Swarm returns a filehash which is submitted to the Smart Contract to be mined and stored in the blockchain.



```
pi@iotproducer:~/vinny/code/thesis_sensor-code $ python3 once_dht11_node_with_encr.py
Smart Contract is Loaded
DHT sensor is initialized
Secret AES key unlocked
Secret HMAC key unlocked

Sequence of steps to store data:
1. Get Readings from DHT11 sensor
DHT11 sensor readings:
Current Temperature in Celsius: 23
Current Humidity: 35

2. Generate Payload from the current temperature and humidity
{"DeviceType": "Raspberry Pi 3B+", "SensorType": "DHT11", "Timestamp": "2019-04-06 19:03:59.006084", "DeviceID": "IoTProducer1", "Humidity": 35, "HumidityUnits": "%", "TemperatureUnits": "Celsius", "Temperature"
: 23, "DeviceIP": "192.168.0.16"}

3. Convert the payload to bytes for encryption
b'{"DeviceType": "Raspberry Pi 3B+", "SensorType": "DHT11", "Timestamp": "2019-04-06 19:03:59.006084", "DeviceID": "IoTProducer1", "Humidity": 35, "HumidityUnits": "%", "TemperatureUnits": "Celsius", "Temperatur
e": 23, "DeviceIP": "192.168.0.16"}'

4. Generate Cipher and Initialization Vector for AES
IV:  SqMr3GWAmTIePKyovxuMcw==
Cipher:  5jctWI0LT4vj6VKoRHMHy1xHwiNqtSn6AhqnFjaV6ZqgcI6rA10ZfxnCDuW7ZS8kbCDu//Rf4RN4SPlZVrFVVXbCVJCsFHKoxNG3R8nMUupi63R5JdLr87z8+Uj998u7ADJmbynfnaf0I3Hw4eowrHewbPeVtfkxlIJi15oH/70DswkJGxmuI0c+tLlIJ5aUJNSmhq6RKw
p/zpdrLHh0iVZ/pQ0clKFeYk6VTNNI5P+XCDSMPg4R9A1dYxdo38U8qnn81uly+x/GwhLrG0kFEURas5XvVSU6a39GeH/1pYYElitjZFc4k5wcBMeqHe3ldqQLLXqE7HlXiellVxkC4g==

5. Generate Signature
IV:  QAbEKh49pKNU3gzCI84MbJtegANOGo0AeM3OOBKMb7g=
6. Payload to send to Swarm
{"iv": "SqMr3GWAmTIePKyovxuMcw==", "signature": "QAbEKh49pKNU3gzCI84MbJtegANOGo0AeM3OOBKMb7g=", "ciphertext": "5jctWI0LT4vj6VKoRHMHy1xHwiNqtSn6AhqnFjaV6ZqgcI6rA10ZfxnCDuW7ZS8kbCDu//Rf4RN4SPlZVrFVVXbCVJCsFHKoxNG3
R8nMUupi63R5JdLr87z8+Uj998u7ADJmbynfnaf0I3Hw4eowrHewbPeVtfkxlIJi15oH/70DswkJGxmuI0c+tLlIJ5aUJNSmhq6RKwp/zpdrLHh0iVZ/pQ0clKFeYk6VTNNI5P+XCDSMPg4R9A1dYxdo38U8qnn81uly+x/GwhLrG0kFEURas5XvVSU6a39GeH/1pYYElitjZFc4k5w
cBMeqHe3ldqQLLXqE7HlXiellVxkC4g=="}
7. Result from POST request to Swarm
c5e8de49175845fdb9040665398f2f02f609b5a084b4d81b0a83f95a66ff8a48
8. Store this filehash to Ethereum Blockchain
pi@iotproducer:~/vinny/code/thesis_sensor-code $
```

Figure 6.1: Storing Data to Proposed System

Figure 6.2 describes implementation of the proposed system which retrieves the filehash from Ethereum using the Smart Contract and then gets the payload associated with that filehash, decrypts the payload and displays it on the connected LCD device.



```
pi@iotconsumer:[redacted] python3 once_console_node_with_decr.py
Smart contract is loaded
Secret AES key unlocked
Secret HMAC key unlocked

Sequence of steps to retrieve data:
1. Get Latest filehash from Ethereum Blockchain
Retrieved filehash is:
c5e8de49175845fdb9040665398f2f02f609b5a084b4d81b0a83f95a66ff8a48

2. Retrieve Encrypted Payload from Swarm
{"iv": "SqMr3GWAmTIePKyovxuMcw==", "signature": "QAbEKh49pKNU3gzCI84MbJtegANOGo0AeM3OOBKMb7g=", "ciphertext": "5jctWI0LT4vj6VKoRHMHy1xHwiNqtSn6AhqnFjaV6ZqgcI6rA10ZfxnCDuW7ZS8kbCDu//Rf4RN4SPlZVrFVVXbCVJCsFHKoxNG3
R8nMUupi63R5JdLr87z8+Uj998u7ADJmbynfnaf0I3Hw4eowrHewbPeVtfkxlIJi15oH/70DswkJGxmuI0c+tLlIJ5aUJNSmhq6RKwp/zpdrLHh0iVZ/pQ0clKFeYk6VTNNI5P+XCDSMPg4R9A1dYxdo38U8qnn81uly+x/GwhLrG0kFEURas5XvVSU6a39GeH/1pYYElitjZFc4k5w
cBMeqHe3ldqQLLXqE7HlXiellVxkC4g=="}

3. Retrieved Signature is:
b'@\x06\xc4*\x1e=\xa4\xa3T\xde\x0c\xc2#\xce\x0cl\x9b^\x80\x03N\x1a\x8d\x00x\xcd\xce8\x12\x8co\xb8'

4. Retrieved Payload from Swarm and decrypt it
{'DeviceType': 'Raspberry Pi 3B+', 'DeviceIP': '192.168.0.16', 'Timestamp': '2019-04-06 19:03:59.006084', 'HumidityUnits': '%', 'Humidity': 35, 'TemperatureUnits': 'Celsius', 'Temperature': 23, 'SensorType': 'DH
T11', 'DeviceID': 'IoTProducer1'}

5. Print these results to console
Temperature:  73  - Humidity: 35 - Timestamp: 2019-04-06 19:03:59.006084
```

Figure 6.2: Retrieving data from Proposed System

6.3 Performance Comparison

Transactions Per Second is usually a bad performance metric for blockchains and decentralized applications. However, we have implemented a storage solution based on private blockchain with decentralized storage and it makes sense to measure load on CPU, memory as well as time taken to process a given set of transactions.

6.3.1 Data set for Tests

A data set consisting of 10,000 records was used to test the performance of the system and see how many devices could theoretically read or write data to the system at a time.

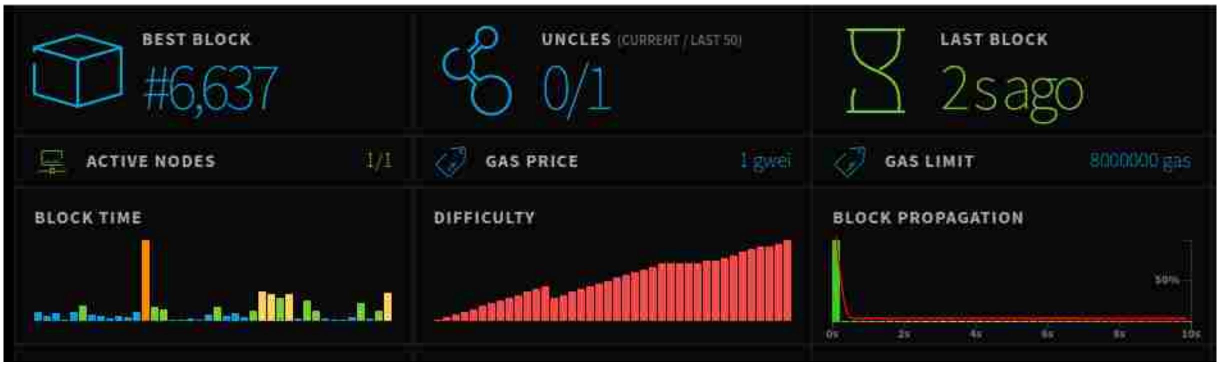
Sample set

```
{
    "Temperature": 30,
    "Humidity": 66,
    "TemperatureUnits": "Celsius",
    ”HumidityUnits”: ”%”,
    ”Timestamp”: ”2019−03−27 23:32:52.193888”,
    ”DeviceType”: ”Raspberry Pi 3B+”,
    ”DeviceID”: ”IoTProducer1”,
    ”DeviceIP”: ”192.168.0.16”,
    ”SensorType”: ”DHT11”
}
```

6.3.2 Ethereum Solution

A new block was generated every 4.5 seconds on average and only 1 transaction was submitted to be mined per block. The mining rate (Network Hash Rate) was around 31 KH/s (Kilo Hash per second) when only the miner1 was running and approximately 51.4 KH/s when both miner1 and miner2 were running on 1 thread each. Using only Ethereum to store the plain text data, a TPS of 13.5 (5000/368) was achieved on the Mining machine and 1.13 Ether was spent to load 5000 records. Average Network Hash Rate rate was improved to 132 KH/s by increasing the number of threads on all miners to 4.

Figure 6.3, 6.4 and 6.5 show different metrics calculated from miners in the Ethereum network.



BEST BLOCK #6,637

UNCLES (CURRENT / LAST 50) 0/1

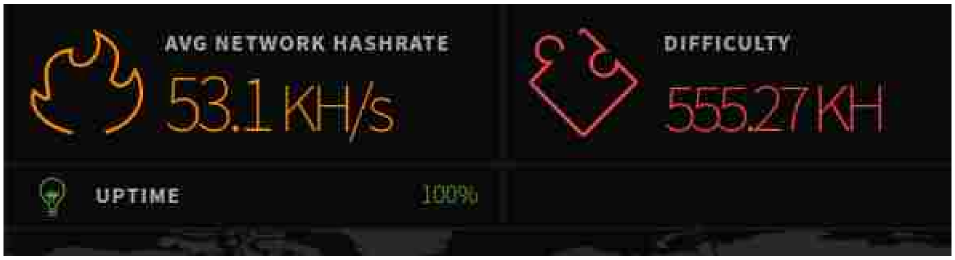
LAST BLOCK 2s ago

ACTIVE NODES 1/1

GAS PRICE 1 gwei

GAS LIMIT 8000000 gas

Figure 6.3: Ethereum Dashboard

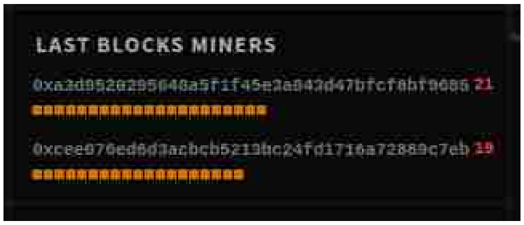


AVG NETWORK HASHRATE 53.1 KH/s

DIFFICULTY 555.27 KH

UPTIME 100%

Figure 6.4: Hashrate of Ethereum Network



LAST BLOCKS MINERS

0xa2d9520295648a5f1f45e2a043d47bfcf8bf9686 21

0xcee676ed6d3acbcb5213bc24fd1716a72889c7eb 19

Figure 6.5: Miners on the Network

6.3.3 Data stored in Swarm

When compared with geth and ipfs, swarm utilized the least amount of CPU power and an equivalent amount of RAM in every test. Performance of Swarm was really fast as it only depends on an HTTP POST request to receive and send data. The data is sent and retrieved from a device set up to serve as the IoT Producer (Raspberry Pi)

Figure 6.6 shows read vs write timings for 10,000 records. Barring a few anomalies, read and write times stayed approximately the same throughout the tests. Using data with encryption caused the performance to degrade by a very small amount as shown in Figure 6.7

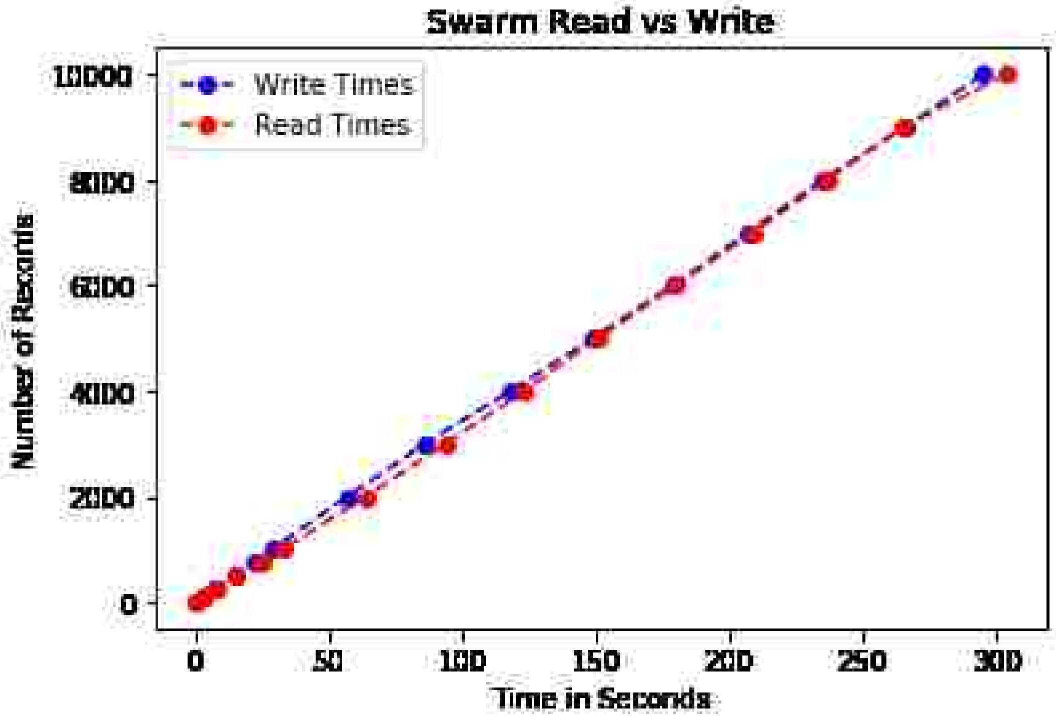


Figure 6.6: Swarm Read vs Write Performance for 10k records

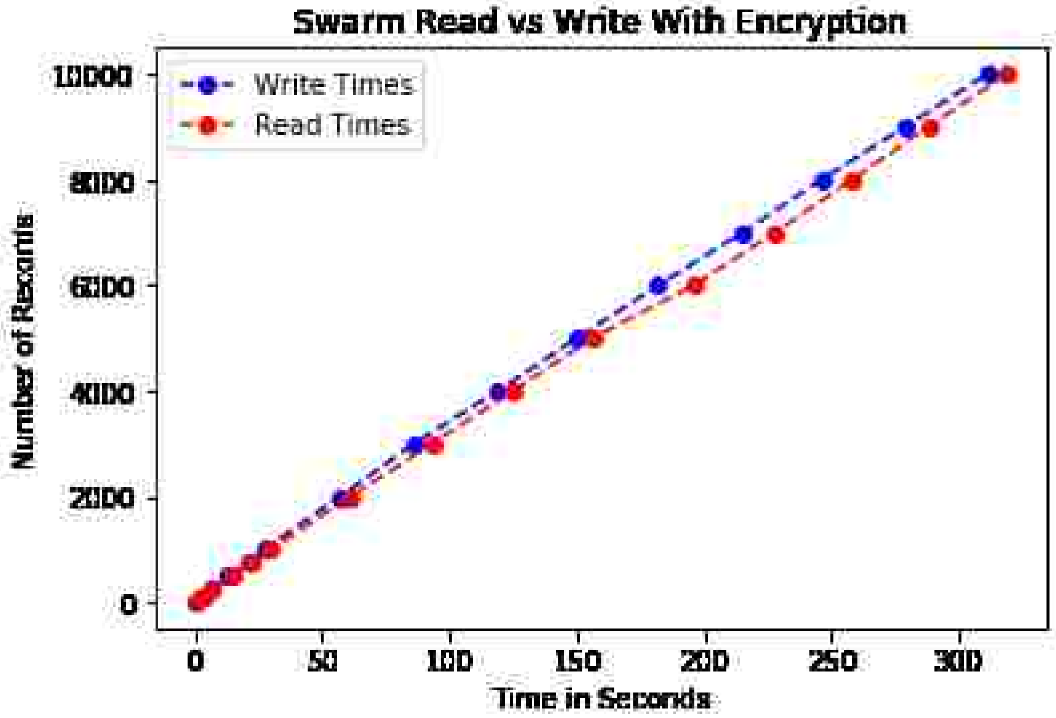


Figure 6.7: Swarm Read vs Write Performance with Encryption for 10k records

6.3.4 Data stored in Ethereum and Swarm

The geth node for storing information from the IoTProducer (Raspberry Pi) is a light node and depends on a full node for getting its information. However, this node is responsible for submitting the transactions generated by calling the Smart Contract function to store file hashes.

Without Encryption

Payload is generated and stored in swarm. The returned file hash is then stored in Ethereum via the smart contract. Figure 6.8 shows the performance difference between reading and write without any encryption.

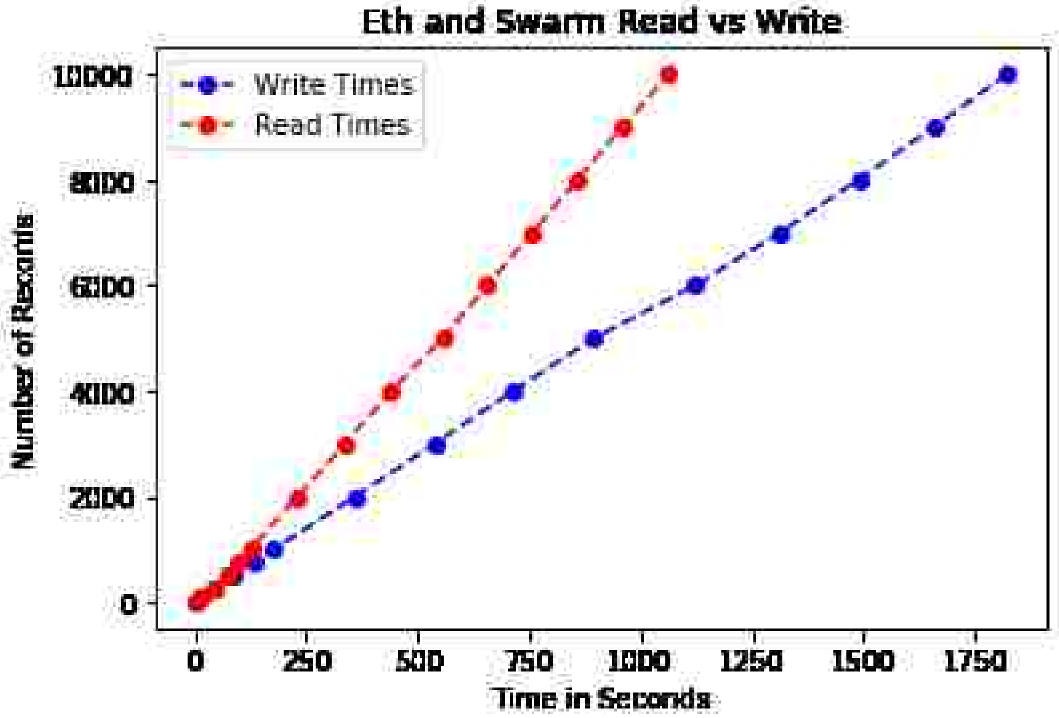


Figure 6.8: Ethereum with Swarm Read vs Write Performance for 10k records

With Encryption

The process of encrypting the payload did not affect the performance adversely as previously feared. Figure 6.9 shows performance difference between read and write with encryption.

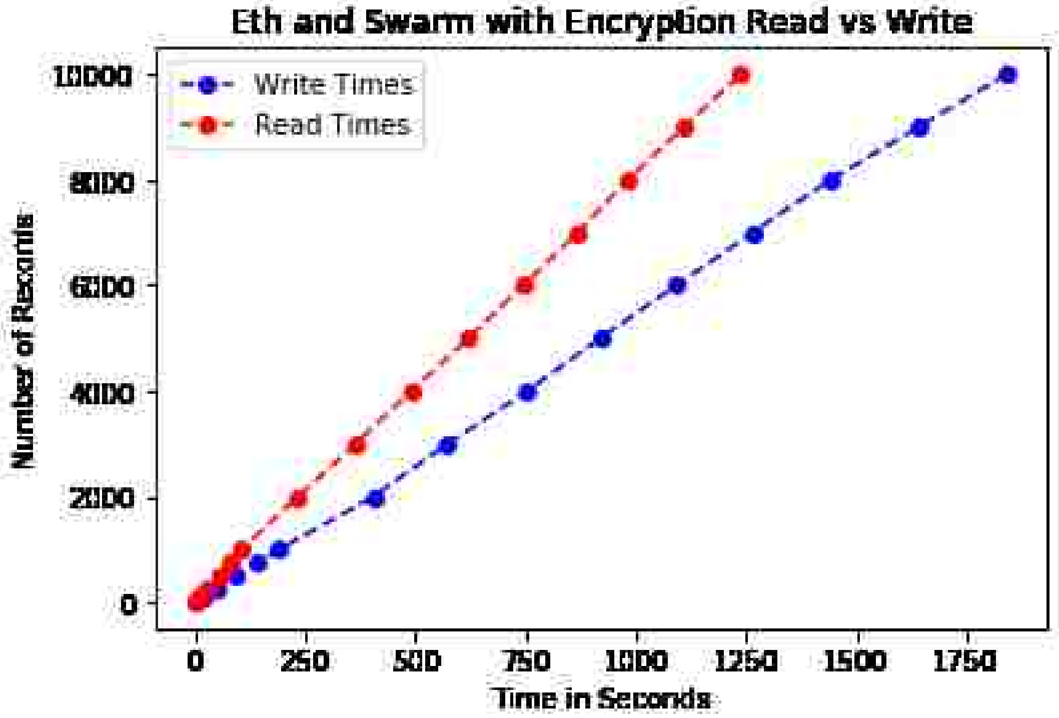


Figure 6.9: Ethereum with Swarm Read vs Write Performance for 10k records

6.3.5 Data stored in IPFS

From the very beginning, IPFS performance in write was significantly worse than Swarm. The data set was reduced to 5000 records and then tested with encrypted data as well as without encryption. However, read performance in IPFS was excellent though it was nowhere near the performance observed when using Swarm.

Figure 6.10 shows performance difference between reading and writing in IPFS.

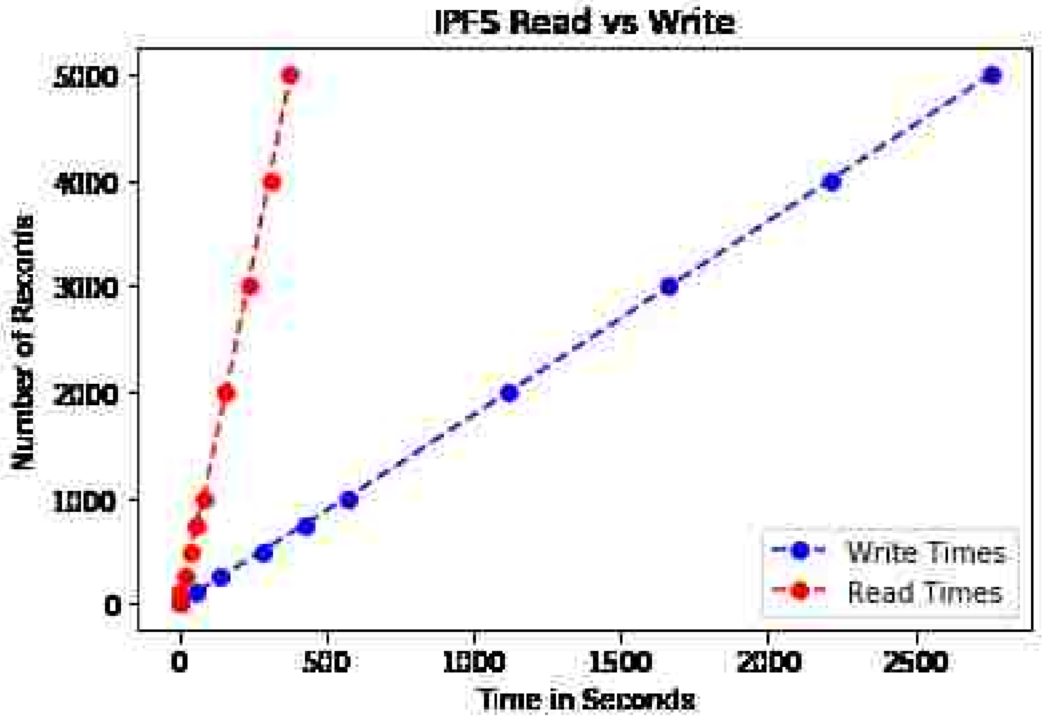


Figure 6.10: IPFS Read vs Write Performance for 5k records

Figure 6.11 shows performance difference between read and write in IPFS with encryption.

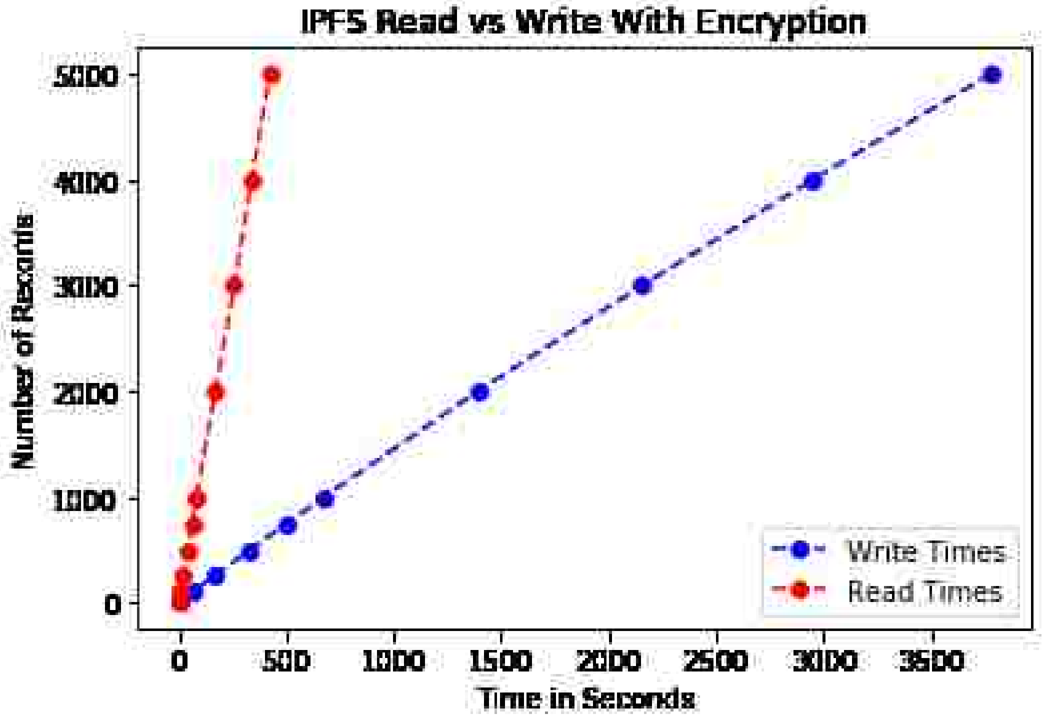


Figure 6.11: IPFS Read vs Write Performance with Encryption for 5k records

6.3.6 Data stored in Ethereum and IPFS

Read vs Write faired similarly when compared to just IPFS, confirming that IPFS is indeed being bottle-necked by the CPU on the raspberry pi.

Figure 6.12 shows performance difference between reading and write using a combination of Ethereum and IPFS.

Figure 6.13 shows performance difference between reading and write using a combination of Ethereum and IPFS.

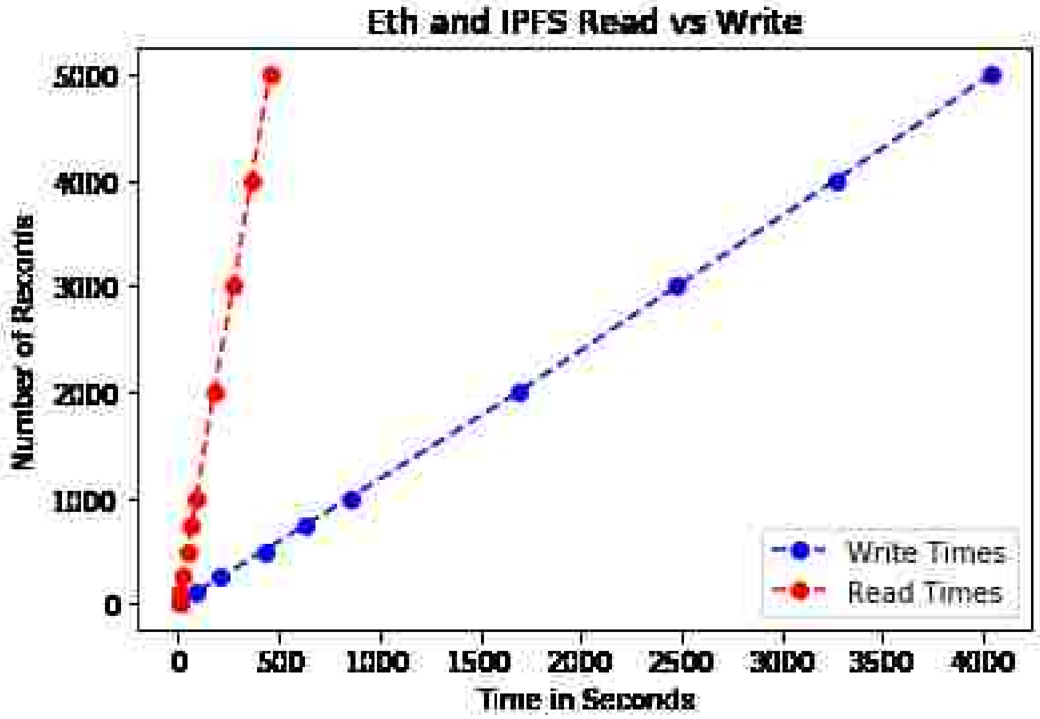


Figure 6.12: Ethereum with IPFS Read vs Write Performance for 5k records

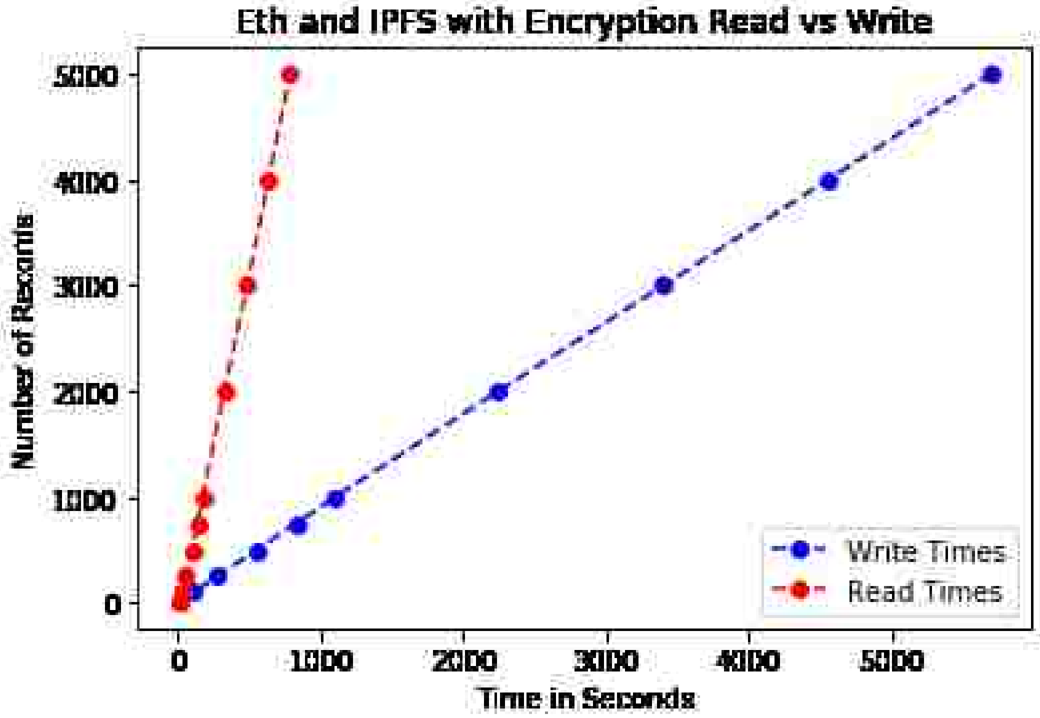


Figure 6.13: Ethereum with IPFS Read vs Write Performance with Encryption for 5k records

6.3.7 Performance Comparisons

Reading or Writing data to the system scales linearly over time due to the fact that both Swarm and IPFS use a form of hashing to store data. Overall, reading and writing to Swarm or IPFS took roughly the same time. Generally, write speeds are slower compared to read speeds. These tests were conducted over a network and latencies from network read and write overshadowed any difference between actual read and write to these storage systems.

In this section, we compare the read and write times for different types of storage systems we have used including those with and without encryption, decryption and signature verification.

Table 6.1 shows different metrics captured during the course of Write tests for 5000 transactions on proposed system running on the IoT Producer (Raspberry Pi). Time Taken is measured in seconds while the Cost is measured in Ether.

Table 6.1: Results for Write tests for 5000 txns

Test	**Time Taken**	**TPS**	**Cost**	**Cost per txn**	**Avg CPU Usage (%)**		**Avg RAM Usage (%)**	
	(seconds)		**(Eth)**		**Geth**	**Swarm/IPFS**	**Geth**	**Swarm/IPFS**
Swarm	149	33.55	NA	NA	0.3	4.3	7.2	52
Swarm + Encryption	150	33.33	NA	NA	0.6	5.7	15.2	53.7
IPFS	2755	1.81	NA	NA	NA	90.1	NA	5.6
IPFS + Encryption	3777	1.32	NA	NA	NA	91.4	NA	5.8
Ethereum + Swarm	895	5.58	0.46	0.000093975	43.7	5	14	51.6
Ethereum + IPFS	4039	1.23	0.46	0.000093975	14.2	76.2	20.7	6.6
Ethereum + Swarm + Encryption	920	5.43	0.46	0.000093975	50.2	6.6	22.2	28.1
Ethereum + IPFS + Encryption	5695	0.87	0.46	0.000093975	13.2	86.8	20.3	6.7

Table 6.2 shows different metrics captured during the course of Read tests for 5000 transactions on proposed system running on IoT Producer (Raspberry Pi).

Table 6.2: Results for Read tests for 5000 txns

Test	**Time Taken**	**TPS**	**Avg CPU Usage (%**		**Avg RAM Usage (%**	
	(seconds)		**Geth**	**Swarm/IPFS**	**Geth**	**Swarm/IPFS**
Swarm	151	33.11	14.1	26	10.5	11
Swarm + Decryption	156	32.05	15.1	30.3	13.9	31.8
IPFS	379	13.19	NA	23.9	NA	5.7
IPFS + Decryption	426	11.73	NA	26.7	NA	5.8
Ethereum + Swarm	556	8.99	26.3	6.6	16.7	10.1
Ethereum + IPFS	456	10.96	31.1	5.9	19.1	6.8
Ethereum + Swarm + Decryption	622	8.03	27.7	6.8	18.3	10.7
Ethereum + IPFS + Decryption	783	6.38	34.2	5.7	20.2	6.5

Performance difference between Swarm and IPFS

Using full version of our proposed system (Ethereum + Swarm), we have achieved approximately 5 transactions per second (TPS) for write and 9 for the read test. This is well within reach of our intended use of 1 transaction per minute per device. We will be able to use this system comfortably for systems where transactions are sent in low to medium frequency.

Using IPFS instead of Swarm lead to significantly lower performance due to the fact that IPFS CPU usage on raspberry pi during write test was incredibly high compared to Swarm. When same tests are performed on the reference machine (Mining machine), we have achieved a TPS of 16 (5000/306) for write and 71 (5000/70) for read.

Table 6.3 shows the TPS comparison for Ethereum, Swarm and IPFS on both the IoT Producer (Raspberry Pi) and Reference Machine (Mining Machine).

Table 6.3: Results for Read vs Write on Raspberry Pi and Mining Machine

Test for 5000 records	**Reference Machine**		**Raspberry Pi**	
Write	**Time Taken**	**TPS**	**Time Taken**	**TPS**
Ethereum + Swarm + Encryption	306	16.33	920	5.43
Ethereum + IPFS + Encryption	1103	4.53	5695	0.87
Read				
Ethereum + Swarm + Decryption	70	71.42	622	8.03
Ethereum + IPFS + Decryption	51	98.03	783	6.38

We include cost incurred for transactions in Ethereum in the tables 6.16 and 6.14. However, this metric doesn't matter directly as we are using a private ethereum network with no intrinsic monetary value. However, these metrics help us indirectly. First, since we do not use Arrays, either static or dynamic in this thesis, the cost incurred doesn't increase over time and stays the same throughout our tests. Secondly, we can use the cost measures to easily identify if any unauthorized transactions are being run on the network.

Performance Cost for Write

Storing only Swarm data was very fast compared to using both the Ethereum Blockchain and Swarm. Adding Encryption did not affect the performance very much. Figure 6.14 shows the performance cost of using just Swarm, IPFS, both Ethereum and Swarm and both Ethereum and IPFS to store the test data set.

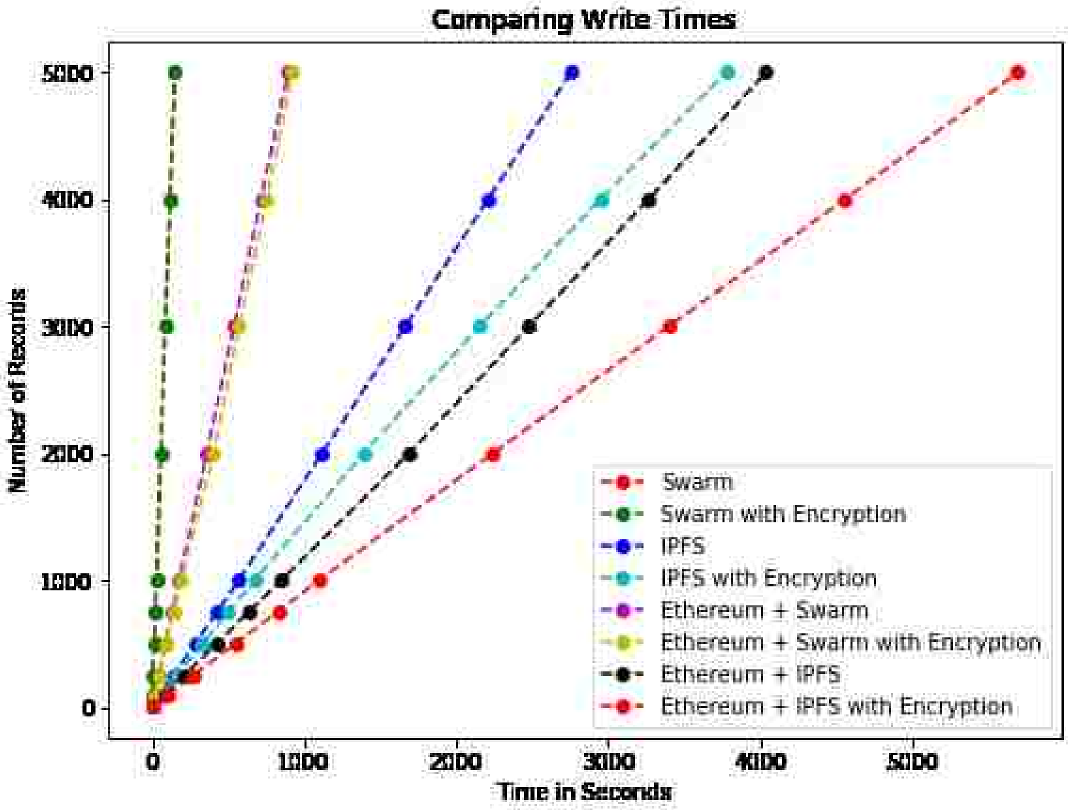


Figure 6.14: Performance Cost for Writes with different methods

The slow CPU and limited RAM on the raspberry pi had a significant impact on the performance of the proposed system. The Mining machine is used a reference to test the Write performance difference between the IoT device and a fully powered machine in Figure 6.15. The Write performance, as mentioned previously is significantly impacted by the CPU on Raspberry Pi when run on a combination of Ethereum and IPFS.

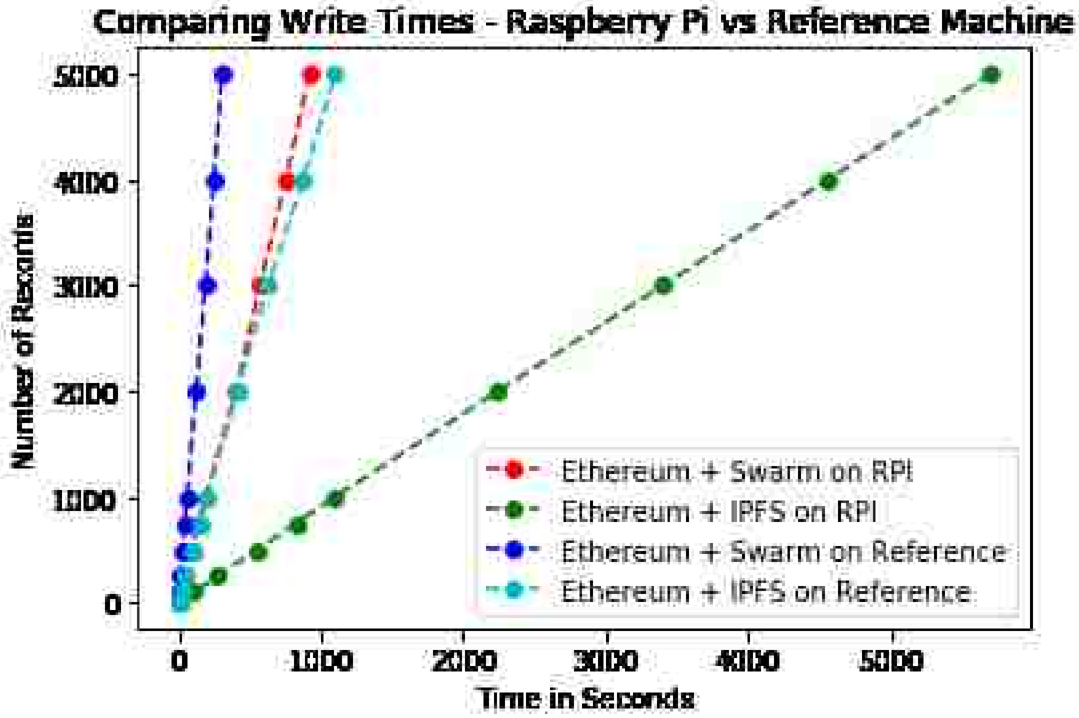


Figure 6.15: Performance Cost for Write on RPI vs Reference Machine

Performance Cost for Read

Reading only Swarm data was very extremely fast compared to using the combination of the Ethereum Blockchain and Swarm method. Adding Encryption did drop the performance by a little amount as the signature must be generated and verified on the receiving end. However, there's much more performance degradation when Ethereum Blockchain and IPFS is used. Figure 6.16 shows the performance cost of using just Swarm, IPFS, both Ethereum and Swarm and both Ethereum and IPFS to read the test data set.

The Mining machine is used a reference to test Read performance difference between IoT device and a fully powered machine in Figure 6.17. Read performed similarly and was comparable when run on the RPI and reference machine.

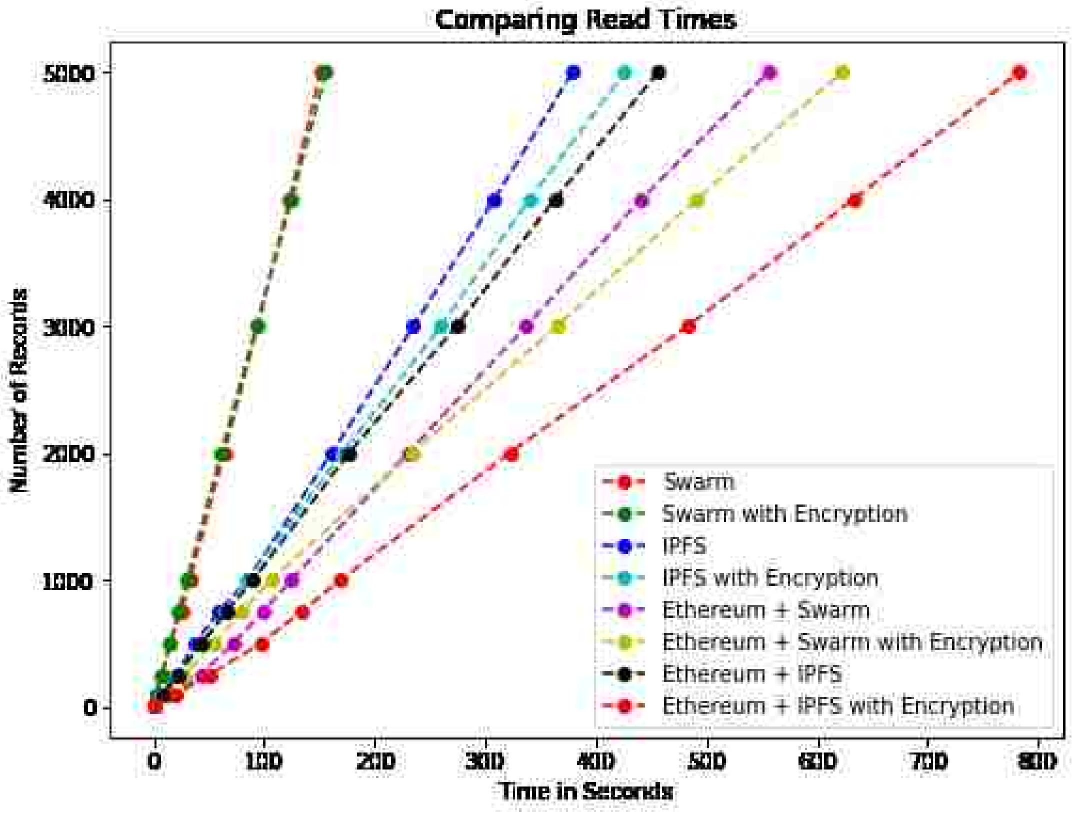


Figure 6.16: Performance Cost for Read with different methods

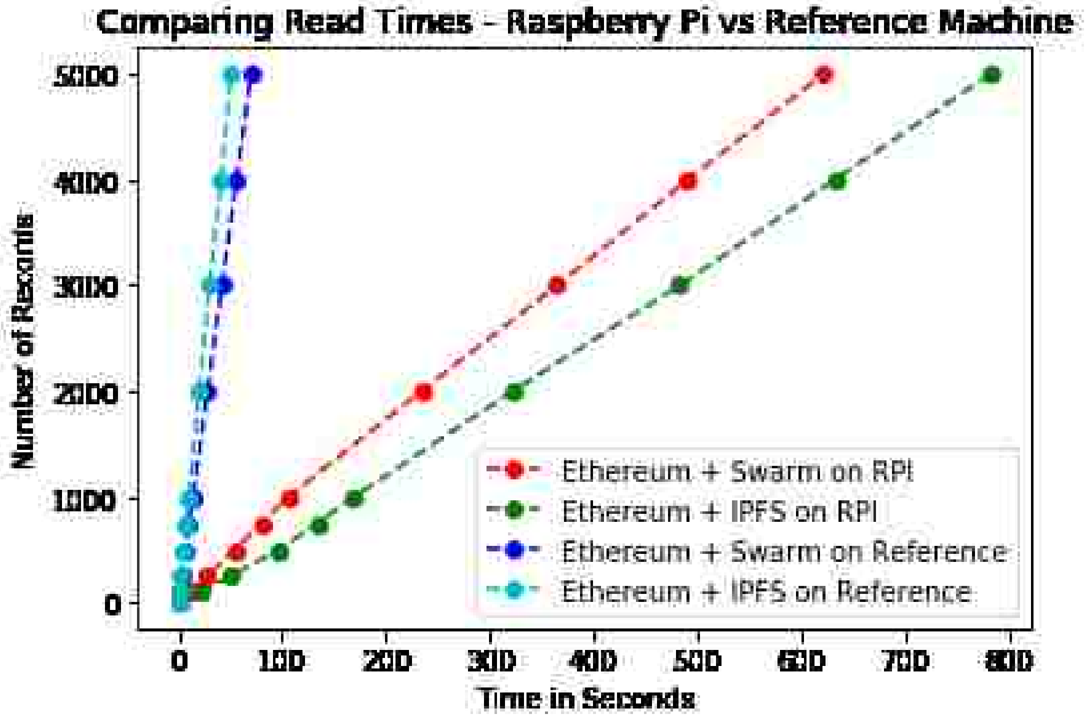


Figure 6.17: Performance Cost for Write on RPI vs Reference Machine

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have successfully set up a system where Ethereum is used to register devices and store references to data stored in IPFS or Swarm. Since Ethereum requires registration before data can be saved to it and Swarm/IPFS data is stored in an encrypted format, we do not even need a private network to store data and can instead use the main networks of Ethereum, IPFS or Swarm.

Performance wise, a combination of Ethereum (to maintain the sequence of records) and Swarm (for actual storage) has seen the best performance when compared to IPFS. In both cases, performance can be increased by using the Raspberry pis only to collect data and instead use a powerful machine to actually submit the transactions for writing data. The Geth, swarm, and ipfs endpoints can be secured using the private PKI we have set up earlier in 5.2.2.

The same method of data storage can be extended to many use-cases such as storing records in fields such as Government, Education, Health care, etc securely and also increase storage capacity as needed without having to stop the services even momentarily for maintenance.

Currently, though, both Swarm and IPFS are not ready for real-time use yet and are still in the process of developing their respective incentivization layers [TFA+16] like Ethereum. Additionally, methods to implement Encryption for data stored in the respective systems are being explored and would further make the process of setting up and using them easy and secure. Swarm is yet to implement a monitoring method to view networking traffic, status of peers, etc while Ethereum and IPFS have good monitoring services to keep track of network performance.

7.2 Future Work

The next areas of focus would be to enhance the quality and types of Sensor data stored and retrieved, secure the IoT devices even further and develop a faster Proof of Work algorithm to validate transactions on the Ethereum blockchain.

A major area of improvement would be to allow multiple producers to send their sensor data to a single time-stamped Swarm link between time intervals. This would allow for much easier processing of data collected from different sources.

Another area of improvement from a blockchain perspective is to use a permissioned blockchain like Quorum[Tea] instead of an open blockchain like Ethereum run in private mode. This allows for much better Access Control and permission for users.

A fast Proof of Work algorithm could be developed as an aim with the express purpose of developing an algorithm for achieving Consensus which is more suitable for IoT data than that is used currently to approve crypto-currency transactions. This optimization would make the algorithm much faster and improve our transaction rate. Aside from using Quorum, tests can be repeated on Ethereum itself when the Proof of Stake protocol goes live with the Casper protocol.

IoT devices are usually underpowered due to power usage limitations. Using resource heavy encryption like RSA and AES (even with hardware acceleration) on these edge devices is counter-intuitive to the idea of using low power devices dedicated for sensing and collecting data. Modern encryption methods like Adiantum [CB18] are tailored for low powered edge nodes and are slowly gaining momentum. Early tests with this algorithm have been promising in terms of resource utilization and speeds.

Bibliography

[AAv18] A. Aldweesh, M. Alharby, and A. van Moorsel. Performance benchmarking for ethereum opcodes. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–2, Oct 2018.

[ABB$^+$18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[ACG$^+$18] M. Alessi, A. Camillo, E. Giangreco, M. Matera, S. Pino, and D. Storelli. Make users own their data: A decentralized personal data store prototype based on ethereum and ipfs. In *2018 3rd International Conference on Smart and Sustainable Technologies (SpliTech)*, pages 1–7, June 2018.

[ADDPSHJ14] Joan Antoni Donet Donet, Cristina Prez-Sol, and Jordi Herrera-Joancomart. The bitcoin p2p network. volume 8438, 03 2014.

[But13] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[But15] V. Buterin. On public and private blockchains. 2015.

[BV16] Simone Bossi and Andrea Visconti. What users should know about full disk encryption based on luks. Cryptology ePrint Archive, Report 2016/274, 2016. https://eprint.iacr.org/2016/274.

[CB18] Paul Crowley and Eric Biggers. Adiantum: length-preserving encryption for entry-level processors. *IACR Trans. Symmetric Cryptol.*, 2018(4):39–61, 2018.

[CD16] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[Dai98] Wei Dai. bmoney, 1998. Accessed: 2016-04-31.

[EHH+16] Wesley Egbertsen, Gerdinand Hardeman, Maarten Hoven, Gert Kolk, and Arthur Rijsewijk. Replacing paper contracts with ethereum smart contracts. 2016. Accessed: 2017-10-24.

[ES] Ethereum-Swarm. Ethereum-swarm. https://docs.swarm.fund/swarm-whitepaper-eng.pdf.

[GBE+18] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018.

[Heg18] P. Hegedus. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 35–39, May 2018.

[IPF14] IPFS. Interplanetary file system. https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf, 2014.

[Kim19] Ron Kim. Trusted platform module and privacy. 04 2019.

[KK18] E. F. Kfoury and D. J. Khoury. Secure end-to-end volte based on ethereum blockchain. In *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, pages 1–5, July 2018.

[KKKH18] D. Khoury, E. F. Kfoury, A. Kassem, and H. Harb. Decentralized voting platform based on ethereum blockchain. In *2018 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*, pages 1–6, Nov 2018.

[Ksh17] N. Kshetri. Can blockchain strengthen the internet of things? *IT Professional*, 19(4):68–72, 2017.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[NHE17] A. Nag, R. J. Haddad, and A. El-Shahat. Analysis of centralized vs decentralized storage systems in pv distributed generation. In *SoutheastCon 2017*, pages 1–2, March 2017.

[OY19] K. R. Ozyilmaz and A. Yurdakul. Designing a blockchain-based iot with ethereum, swarm, and lora: The software solution to create high availability with minimal security risks. *IEEE Consumer Electronics Magazine*, 8(2):28–34, March 2019.

[PE18] J. D. Preece and J. M. Easton. Towards encrypting industrial data on public distributed networks. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4540–4544, Dec 2018.

[PPM+18] C. Pop, C. Pop, A. Marcel, A. Vesa, T. Petrican, T. Cioara, I. Anghel, and I. Salomie. Decentralizing the stock exchange using blockchain an ethereum-based implementation of the bucharest stock exchange. In *2018 IEEE 14th International*

Conference on Intelligent Computer Communication and Processing (ICCP), pages 459–466, Sep. 2018.

[RD17] S. Rouhani and R. Deters. Performance analysis of ethereum transactions in private blockchain. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 70–74, Nov 2017.

[Sco17] Scott. Encrypting your root file system on raspberry pi - using luks and dm-crypt. https://community.zymbit.com/t/encrypting-your-root-file-system-on-raspberry-pi-using-luks-dm-crypt/150, 2017.

[Tea] Quorum Team. Quorum. https://github.com/jpmorganchase/quorum/blob/master/docs/QuorumWhitepaperv0.2.pdf.

[TFA+16] Viktor Tron, Aron Fischer, Daniel Nagy A, Zsolt Felfldi, and Nick Johnson. swap, swear and swindle: incentive system for swarm. Technical report, Ethersphere, 2016. Ethersphere Orange Papers 1.

[VC14] David Vorick and Luke Champine. Sia: Simple decentralized storage. Technical report, Sia, 2014.

[WBBB14] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj: A peer-to-peer cloud storage network. Technical report, storj, 2014. v1.01.

[Whi18] J. Whitter-Jones. Security review on the internet of things. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 163–168, April 2018.

[Woo14] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.

[WZY+18] X. Wang, X. Zha, G. Yu, W. Ni, R. P. Liu, Y. J. Guo, X. Niu, and K. Zheng. Attack and defence of ethereum remote apis. In *2018 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, Dec 2018.

[WZZ18] S. Wang, Y. Zhang, and Y. Zhang. A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access*, 6:38437–38450, 2018.

[XHLX18] Q. Xu, Z. He, Z. Li, and M. Xiao. Building an ethereum-based decentralized smart home system. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1004–1009, Dec 2018.

[XSML18] Q. Xu, Z. Song, R. S. Mong Goh, and Y. Li. Building an ethereum and ipfs-based decentralized social network system. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–6, Dec 2018.

[ZKS+18] Yuanyu Zhang, Shoji Kasahara, Yulong Shen, Xiaohong Jiang, and Jianxiong Wan. Smart contract-based access control for the internet of things. arXiv:1802.04410, 2018. Accessed:2018-02-16.

[DY18] K. R. zyilmaz, M. Doan, and A. Yurdakul. Idmob: Iot data marketplace on blockchain. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 11–19, June 2018.

[Y17] K. R. zylmaz and A. Yurdakul. Work-in-progress: integrating low-power iot devices to a blockchain-based infrastructure. In *2017 International Conference on Embedded Software (EMSOFT)*, pages 1–2, Oct 2017.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Vinay Kumar Calastry Ramesh
vinay.calastry@gmail.com

Degrees:

Bachelor of Technology in Computer Science 2014
Jawaharlal Nehru Technological University, Hyderabad, India

Thesis Title: Storing IoT Data Securely In A Private Ethereum Blockchain

Thesis Examination Committee:

Chairperson, Dr. Yoohwan Kim, Ph.D.
Committee Member, Dr. Ju-Yeon Jo, Ph.D.
Committee Member, Dr. Fatma Nasoz, Ph.D.
Graduate Faculty Representative, Dr. Satish Bhatnagar, Ph.D.