Using Blockchain to Create a Decentralised Security Model for Distributed Systems

Adam David Bruce
a.bruce3@newcastle.ac.uk

April 2021

Abstract

//TODO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 COVID-19 and Cyberattacks

In the summer of 2020 during the midst of the COVID-19 pandemic, universities and research institutions worldwide were working hard to understand the structure of the virus and develop a vaccine in an attempt to return to normality. However, whereas some countries were making fast progress in understanding the virus, others were falling behind, and the virus began to put a strain on healthcare, and increasing critique on governments. In order to keep up with the nations at the forefront of vaccine development, nations turned to state-sponsored cyber-attacks in order to both hinder nations, and also obtain research and information about other countries' vaccine efforts. One such example was the threat group 'Cozy Bear', formally known as Advanced Persistent Threat (APT) 29. APT29 used a number of tools to target various organisations involved in COVID-19 vaccine development in Canada, the United States and the United Kingdom. The National Cyber Security Center (NCSC) believe that the intention was highly likely stealing information and intellectual property relating to the vaccine [5].

In addition to the mortality of COVID-19, the virus also caused a number of economic issues across a number of nations. Global stock markets lost $6 trillion in value over size days from 23 to 28 February [6]. This gave private companies no other choice than to make large volumes of staff redundant, which increased job insecurity causing many people to become redundant, and in nations without suitable support or benefits, attackers turned to cybercrime for financial gain. These attacks represented the majority of cyberattacks aimed at both universities and the general public. A study of cyber-crime throughout the COVID-19 pandemic determined that 34% of attacks directly involved financial fraud with a number of attack surfaces used, the majority being phishing, smishing and malware [7].

University attacks became a frequent headline in the UK as universities suffered attacks from different threat actors. A number of threat actors launched attacks against multiple universities in the hope to find a vulnerability in at least one. One such attack was aimed at both Newcastle University and Northumbria University, two universities in extremely close proximity [8, 9]. The attack crippled both Newcastle and Northumbria Universities, however the attackers only managed to exfiltrate data from Newcastle University. Why was the attack successful on both occasions? Why wasn't knowledge of the attack shared?

One reason is that currently, there is no reliable or automated system in place to share this information. Such a system is what this paper will aim to create.

1.2 Distributed Systems

A distributed system is defined by Tanenbaum and van Steen as a "collection of independent computers that appears to its users as a single coherent system" [1]. Such systems are commonplace in peer-to-peer computing and sensor networks where each systems contributes some

data via transactions to the system. A distributed system therefore should be autonomous and to the user, should appear as though they are interacting with a single system. Furthermore users and applications should be able to interact with the distributed system in a consistent and uniform way, regardless of where and when system interaction takes place. This requires a common interface provided by a stub which is used to bridge the gap between a programming language or protocol and the distributed system. This stub hides the differences in machine architecture and communication between the computer and the distributed system. The use of stubs creates a new software layer, known as middleware which runs on an Operating System (OS) and exposes distributed functions to higher-level applications and users.

1.3 Decentralised Systems

Reed defines a decentralised computer system as a computer system that "involves separation of the computers in the system by physical distance, by boundaries of administrative responsibility for individual computers and their applications, and by firewalls" [10]. Reed suggests that for a computer system to be decentralised, it must be separated by both physical distance and administrative responsibility, such that no single body administrates the system. One of the most well-known examples of decentralisation is cryptocurrency, a currency which takes no physical form, but instead exists entirely digitally. If cryptocurrency were to be governed by a central body, nefarious transactions could be used to launder money. Using a decentralised system ensures the transaction can only take place if all nodes within the system are in consensus that the transaction is genuine.

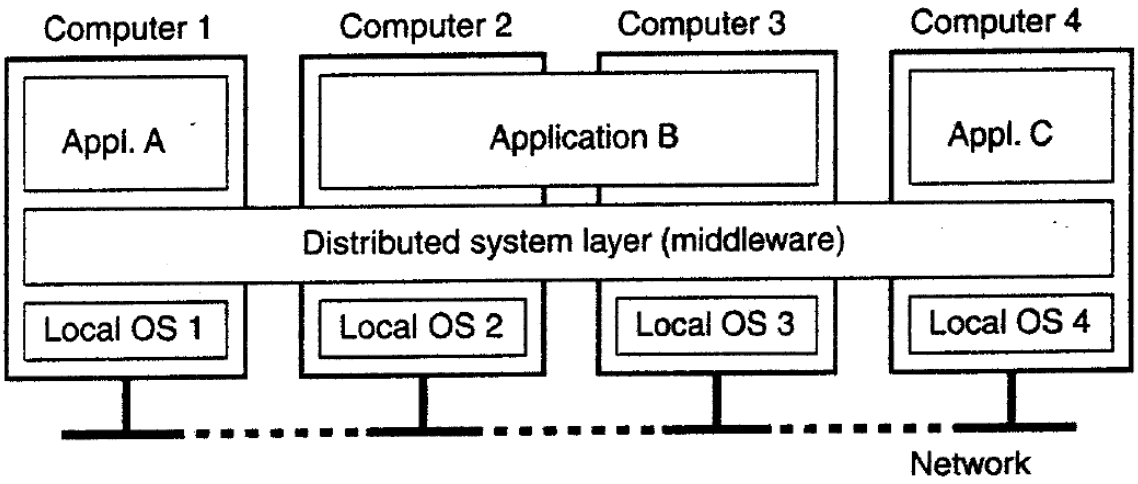

Figure 1.1: A Distributed System visualised as middleware [1]

1.4 Blockchain

Nofer et al. define blockchains as "data sets which are composed of a chain of data packages (blocks) where a block comprises multiple transactions. The blockchain is extended by each additional block and hence represents a complete ledger of the transaction history." [2]. Nofer et al. describe the basic fundamentals of a blockchain, which is that numerous blocks of transactions contribute to a larger chain. This chain is never controlled by a single body, instead a copy of the chain is stored at each node within a system, making blockchain a popular candidate for

controlling transactions over a decentralised computer system. Hence, blockchain is the foundation for the vast majority of cryptocurrencies including Bitcoin[11] and Ethereum[12]. One of the key aspects of blockchain is the use of cryptographic hashing algorithms, these algorithms represent a block as a fixed-length string. For a block to be added to the chain, it must contain the hash of the previous block.

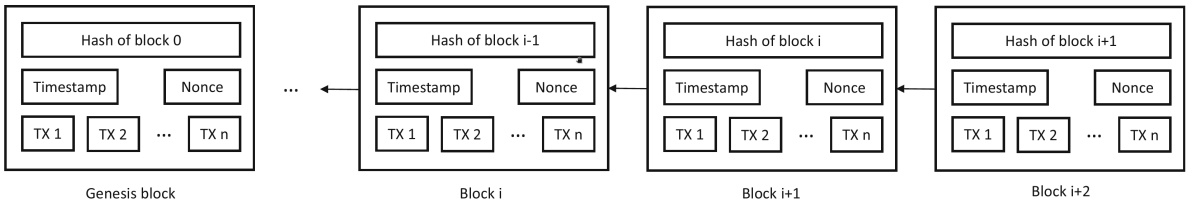

Figure 1.2: An example of a blockchain [2]

Chapter 2

Project Aims and Objectives

2.1 Aim

The original aim for this project was to design and create a decentralised firewall that could communicate knowledge of cyberattacks aimed at universities in real-time, allowing other universities to protect themselves from the same attacks. This system would be distributed, and hence must conform to the previous description of a distributed system in section 1.2. Following an extensive amount of background reading, there appeared to be no existing implementation or design of such a system which inspired me to alter my aim and instead focus entirely on designing a protocol and implementing a stub to demonstrate the protocol's effectiveness. This project will therefore not be implementing a firewall, but instead a system to coordinate firewalls. Further research determined that blockchain was the best choice for the underlying structure for such a protocol, and so this final change shaped the current aim for this project: **Using Blockchain to Create a Decentralised Security Model for Distributed Systems**.

2.2 Objectives

The following objectives provide an outline for what this project hopes to achieve:

1. Evaluate the effectiveness of existing distributed security mechanisms.
2. Investigate methods of establishing connections and synchronising computers within distributed systems.
3. Understand the structure of blockchains and adapt them for firewall transactions.
4. Implement and test relevant resilience, fault tolerance and security mechanisms.
5. Compare the use of decentralised security mechanisms.

Chapter 3

Background

3.1 Distributed Systems

The primary reference used for distributed systems was Tanenbaum and van Steen's "Distributed Systems: Principles and Paradigms"[1], who's literature provides an in-depth explanation from the fundamental theory of distributed systems to the design and implementation of such systems. Key details that were taken from this publication are detailed below. In general, this book covered the essential components of creating a distributed system, however much of the detail with regards to client-server interactions was not applicable to this project due to it's decentralised nature. Furthermore, a large portion of the book was not of interest to this project as it focuses on distributed processing, which only comprises a small element of this project, hence a large volume of information regarding implementation of processing was not useful.

3.1.1 Architecture

Tanenbaum and van Steen cover many aspects of a distributed system's architecture spanning network, software and physical architecture. This project will implement a decentralised, peer-to-peer network architecture, which will be discussed in detail in section 3.2. The software used will consist primary of stubs, which are used to hide the differences in machine architecture and communication between the computer and the distributed system. The combined use of stubs creates a new software layer, known as middleware which provides a common interface between a client application, and the distributed system. Creating this layer enables applications to communicate via an application-level protocol, which is independent from the protocol spoken by the middleware.

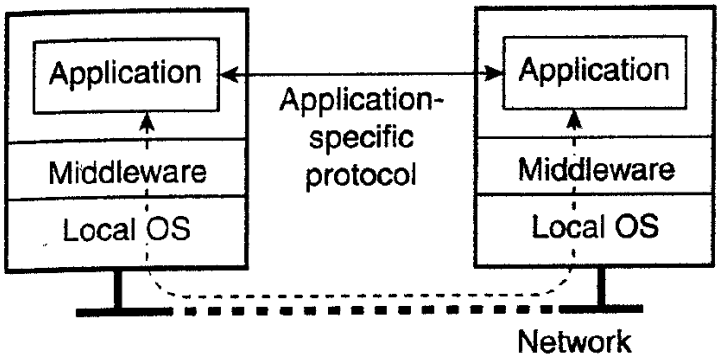

Figure 3.1: Application layer protocol running over middleware [1]

With regards to physical architecture, Tanenbaum and van Steen discuss a number of approaches to client-server architectures, however due to the decentralised nature of this project, non of Tanenbaum and van Steen's classifications apply.

3.1.2 Remote Procedure Calls (RPC)

Tanenbaum and van Steen introduce the concept of a Remote Procedure Call (RPC). RPCs are used to execute some action on a remote node within a distributed system. Tanenbaum and van Steen provide a concise breakdown of the steps required to execute an RPC:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local Operating System (OS).
3. The client OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls it's local OS.
8. The server's OS send the message to the client's acrshortOS.
9. The client's OS sends the message to the client stub.
10. The stub unpacks the result and returns to the client.

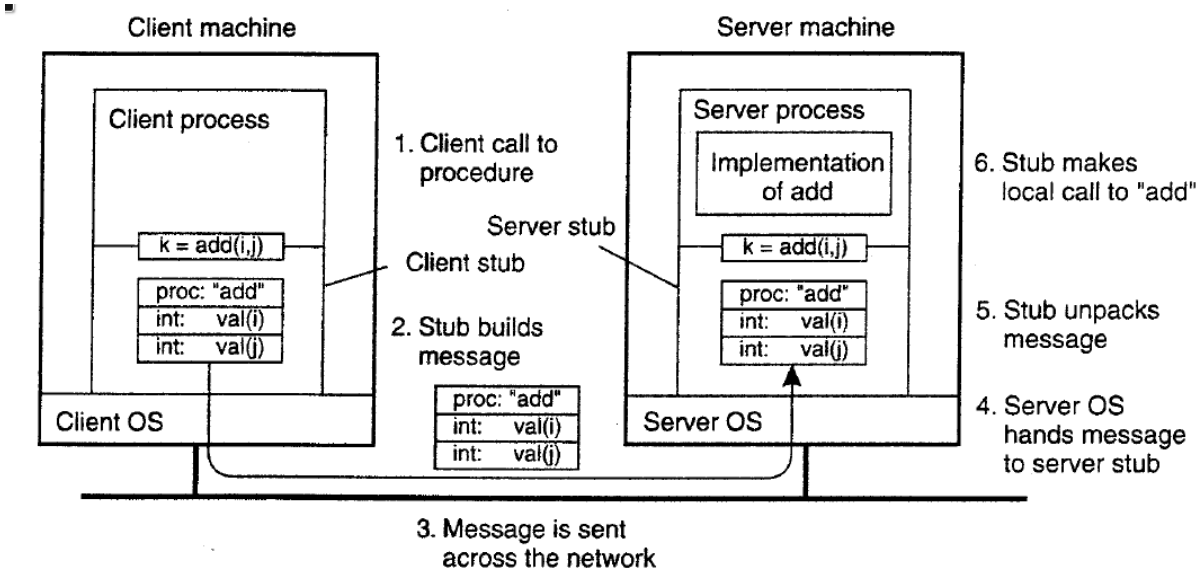

Figure 3.2: A breakdown of an RPC [1]

3.2 Decentralised Systems

Gray's "An Approach to Decentralized Computer Systems" [13] provided the basis for the decentralisation aspect of this project. Gray summarises the advantages of using decentralised systems, a number of which support the argument for using a decentralised topology in this project. The advantages which are relevant to this project are documented below.

- **Capacity**: A decentralised system can support a large number of devices.
- **Response Time**: Having devices in close proximity can reduce response times.
- **Availability**: A failure is likely to be limited to a single site, allowing the rest of the system to continue normal operation.
- **Security**: Removing the central controller in a traditional distributed system removes the risk of an attack compromising the whole system.

Gray's article also looks at how decentralised systems should be designed including data types, network protocols and transactions. There are a number of similarities between Tanenbaum and van Steen's RPCs and the structure Gray proposes for decentralised transactions. For this project however, the finer details proposed by Gray's system are not relevant as the literature uses a large number of examples base heavily on financial transactions, which contain a number of additional complexities over the transactions used within this project.

3.3 Blockchain

The primary reference used for blockchain was Nofer et al.'s "Blockchain"[2]. Nofer et al. provide a high level overview of blockchain, focusing primarily on the structure and implementation, with some consideration of the current applications of blockchain in both financial and non-financial settings. Although concise, this publication provides a valuable summary of the essential components of blocks in order to create a ledger which can accurately trace transactions. Although not essential for this project, Nofer et al. additionally discuss how blockchain can be implemented into smart contracts. In general, this literature was useful in providing a baseline for the structure of blocks within a blockchain, and clearly explained the purpose of each field within the block, which allowed informed decisions to be made in regard to the structure of blocks used in this project.

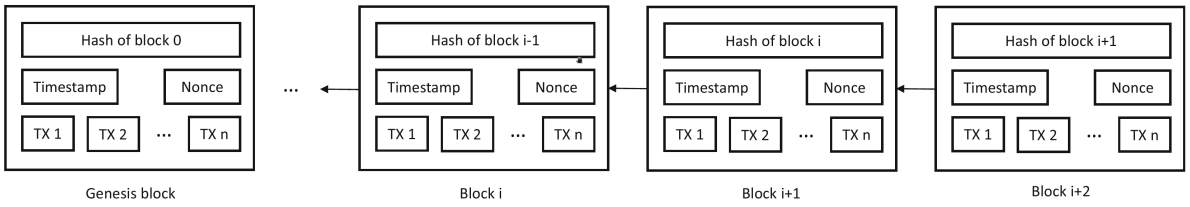

Figure 3.3: An example of a blockchain [2]

3.4 Distributed Security

The primary reference used for distributed security was Rivets and Lampson's "SDSI - A Simple Distributed Security Infrastructure" [3]. This publication provides an in-depth explanation of how a public-key infrastructure can be used in conjunction with access control lists to create a distributed security infrastructure. The majority of the literature within this publication is focussed on creating and issuing certificates, something that is not relevant for this project, however Rivets and Lampson did provide clear requirements over the data structures within such a system. Rivets and Lampson implement a message system similar to that of Tanenbaum and van Steen's in section 3.1. The message system proposed by Rivets and Lampson contains only a type and dictionary of attributes.

```
( type:
  ( Attribute1: value1 )
  ( Attribute2: value2a value2b value2c )
  ... )
```

Figure 3.4: Message format for the SDSI Model [3]

Additionally, Rivets and Lampson detail the concept of objects, which are defined by a type. This type is expressed in the form

```
protocol-name.message-type
```

3.5 Firewalls and Firewall Rules

Al-Shaer and Hazem provide a detailed explanation of how firewall policies should be modelled and managed in "Modeling and Management of Firewall Policies" [14]. This article explores methods of modelling policies and rules and provides a deep analysis of how those rules are interpreted by a firewall. The most relevant discussion within this literature is the structure of a firewall policy which is defined by Al-Shaer and Hazem as a set of rules, which act as records, with the following seven fields:

- **Order**: The priority of a rule.
- **Protocol**: Either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP).
- **Source IP**: The source IP address.
- **Source Port**: The source port.
- **Destination IP**: The destination IP address.
- **Destination Port**: The destination port.
- **Action**: The action the firewall should take (e.g. ACCEPT, DENY).

In regards to this project, there was little other relevant content in the literature. Following the change in this project's aim detailed in section 2.1, this project was no longer concerned with the implementation of a firewall or the interpretation of firewall rules, which deemed the vast majority of Al-Shaer and Hazem's publication irrelevant.

Additional background information with regards to how firewalls are integrated into infrastructure came from Dulaney and Eastton's "CompTIA Security+ Study Guide: Exam SY0-501" [15]. Dulaney and Eastton's study guide covers a large number of aspects associated with cyber security, including technological, physical and psychological mitigations. With regards to firewalls, this publication details how the placement of firewalls can be used to form a Demilitarised Zone (DMZ), which is a common network layout used by universities, as it permits certain areas of the network to be accessible from outside the local network.

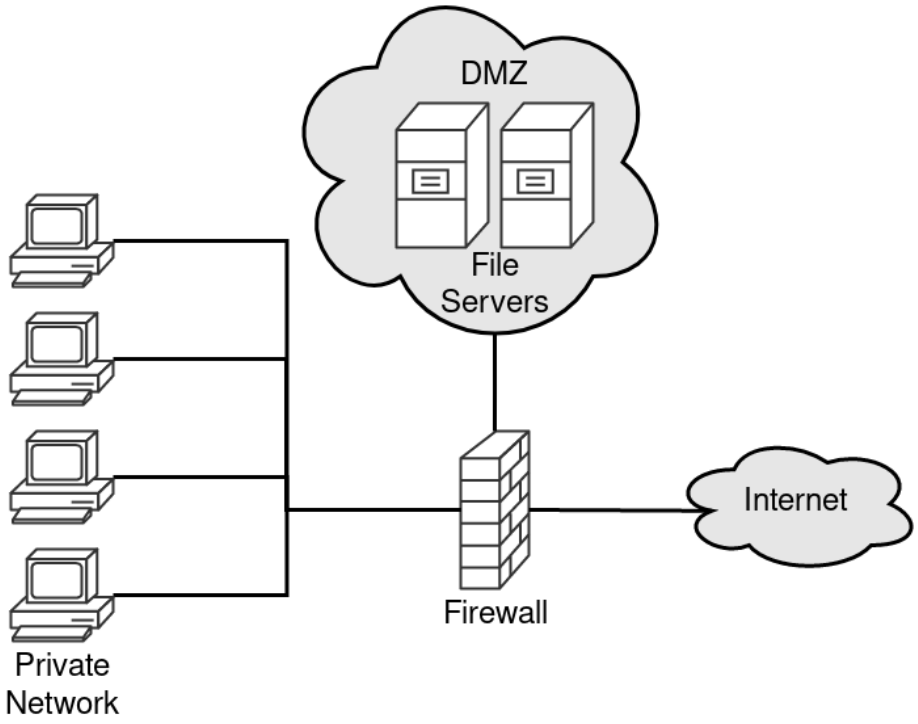

Figure 3.5: Using a firewall to create a DMZ

This information provided a strong understanding into where this project would fit in a standard network model. This publication also covered firewalls, however the information provided was not as detailed as that from Al-Shaer and Hazem, and hence no other aspects of this book were used.

3.6 Fault Tolerance

The primary reference for fault tolerance was Guerraoui and Schiper's "Fault-Tolerance by Replication in Distributed Systems"[4]. This literature details how replication can be used to provide fault tolerance in a distributed system in addition to ensuring consistency is maintained. A number of backup techniques are discussed however the technique that is best suited for application is primary backup replication. Primary backup replication consists of a client invoking an operation which is the applied to the primary data, and then cascaded to a number of additional backups.

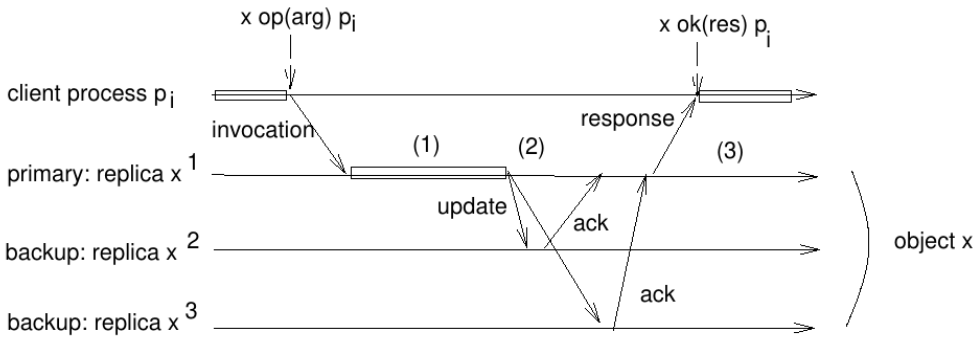

Figure 3.6: Primary backup technique [4]

Guerraoui and Schiper continue to discuss methods of detecting faults and appropriate ways to deal with them. The methods covered however require a much greater level of control than that achievable by a single application running on a standard OS, and hence are not applicable to this project.

3.7 Computer Networks

The primary reference for computer networks was Lammle's book "CompTIA Network+ Study Guide: Exam N10-007" [16]. Lammle's study guide covers a wide range of aspects associated with computer networks including physical implementations, subnets, security and protocols. Two sections which are relevant to this project are network layer protocols (TCP and UDP), and network topologies. Lammle provides a comprehensive explanation of the differences between Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), however the difference that is most relevant to this project is TCP's requirement for a connection to be established prior to transmission. The book explains how establishing a TCP connection takes additional time and blocks the port whilst attempting to establish a connection, which prevents any other connection from being made from that port. This is not ideal for a decentralised system as messages will be sent on an ad-hoc basis, with strict time constraints, and therefore UDP will be more suitable for this project.

With regards to network topologies, Lammle details seven approaches: bus, star, ring, mesh, point-to-point, point-to-multipoint and hybrid. In order to create a truly decentralised distributed system, the mesh topology is best suited to this project. In a mesh topology, each device is connected to every other device, which provides the highest level of redundancy possible, as if one device were to crash, or a cable be disconnected, communication can continue via the other devices.

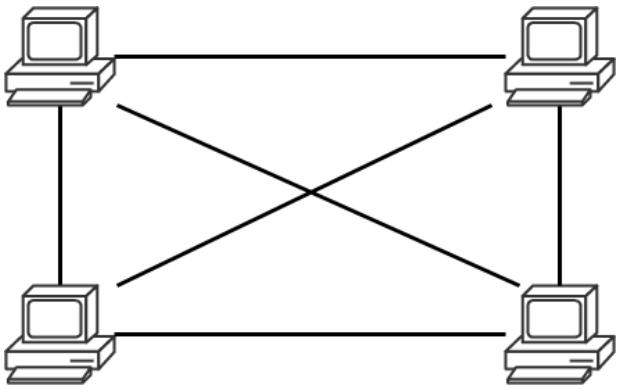

Figure 3.7: Mesh Topology

3.8 Inter-Process Communication (IPC)

Tanenbaum and Woodhull provide an extremely detailed breakdown of their UNIX based operating system in "Operating systems: Design and Implementation" [17]. This book covers all aspects of an OS, providing insightful explanations of the decisions made at every step in the design process. As this project is designed to run as an application, many of the details covered in this publication are irrelevant, however Tanenbaum and Woodhull cover one essential aspect of this project: Inter-Process Communication (IPC). IPC is a function within many Operating Systems which enables multiple processes to communicate by passing messages to each other. IPC will be the technique used to interact with the distributed system, as a client process will use IPC to issue commands to the stub.

Chapter 4

Design

4.1 System Architecture

The system will be compiled into one single executable binary, but will interface with a number of external libraries and the OS via system calls. The stub will primarily communicate with the OS in order to establish sockets and bind them to the desired ports, the framework will then communicate with the Network Interface Card (NIC) to obtain it's assigned Internet Protocol (IP) address. Whenever a new block is created or received, OpenSSL's libcrypto library [18] will be used to generate or validate the block's hash. This interaction is visualised in the following component diagram.

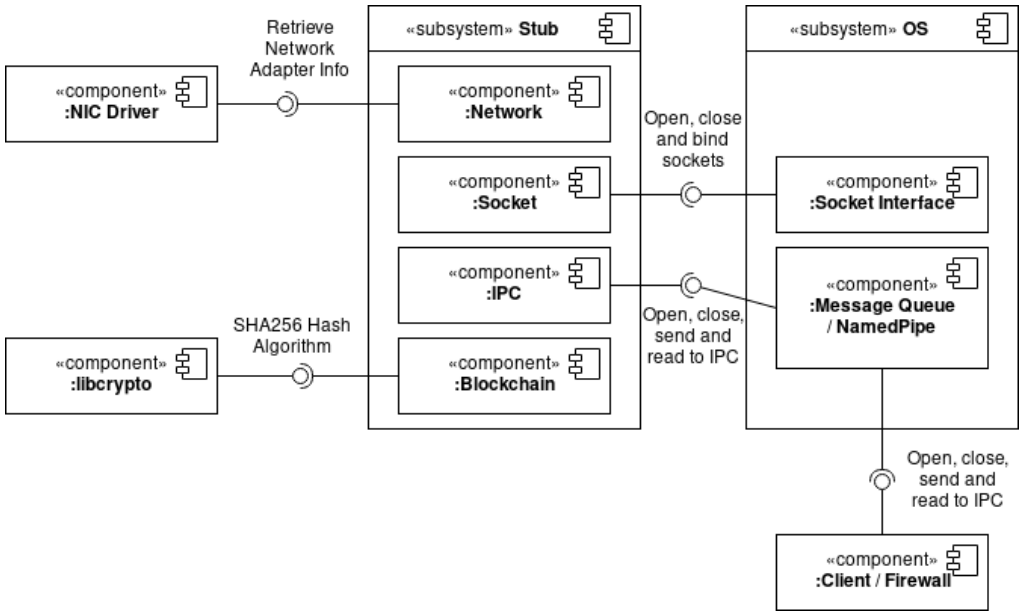

Figure 4.1: System Component Diagram

4.2 Data Structures

The system uses two primary structures for storing data within the system, these are used for storing firewall rules and blocks.

4.2.1 Firewall Rule

The system supports five different actions that can be applied to a firewall rule: allow, deny, bypass, force allow and log. These actions are supported by the distributed system but firewall software remains free to interpret these actions freely, which ensures consistency can be achieved despite different firewall vendors. The firewall rule structure comprises of the following:

Field Name	Data Type	Size (Bytes)
source_addr	Character Array	15
source_port	Unsigned Integer	2
dest_addr	Character Array	15
dest_port	Unsigned Integer	2
action	FirewallAction	1

Table 4.1: The Structure of a Firewall Rule

This structure supports the standard model for firewall policies, supporting categorisation by the source address, source port, destination address and destination port. Each rule is then given an associated action.

4.2.2 Firewall Block

Firewall blocks are used to create the chain of firewall rules within the system. These are comprised of the following fields:

Field Name	Data Type	Size (Bytes)
last_hash	Unsigned Char Array	Dependent on Hash Algorithm
author	Character Array	15
rule	FirewallRule	Dependent on machine architecture
next	FirewallBlock Pointer	Dependent on machine architecture

Table 4.2: The Structure of a Firewall Block

The structure holds the last block hash in order to verify that the block has been validated and fits onto the chain. In addition to the last block hash, the structure includes the author of the block, which enables an audit to verify that the block was submitted by a reputable source. The structure contains a firewall rule which has been described previously in section 4.2.1. Finally, the structure contains a pointer to another block, this field is used to join the blocks into a chain via a linked list. When a block is appended to the chain, this field will be null, however once the next block is added to the chain, this field will point to the new block.

4.3 Message Structures

The system uses two different message passing mechanisms, these being via the network for distributed transactions, and via IPC for client - stub communication. Each message type follows a particular structure which is discussed below.

4.3.1 Network

For communications within the distributed system, three different message types are used: advertisement, consensus and rule. Furthermore, each message contains a subtype which is either a broadcast or an acknowledgement. All message types contain a set of common fields, which are detailed below.

Field Name	Data Type	Size (Bytes)
type	MessageType	1
subtype	MessageSubType	1
hops	Unsigned Integer	1
source_addr	Character Array	15
target_addr	Character Array	15
next_addr	Character Array	15

Table 4.3: Common Fields in all Network Messages

The type field holds an enum which is called MessageType and contains three values: advertisement, consensus and rule. Similarly the subtype field holds an enum that contains two values: broadcast and acknowledgement. Broadcast messages are used to advertise a new device, request consensus and distribute new a firewall transaction, whereas acknowledgements are used to provide a response, such as acknowledging the new device and providing consensus for a proposed transaction. As the system is decentralised, messages are not sent directly to remote hosts as there may be devices on the network that another device is unaware of, which would result in unsynchronised blockchains. The use of a hop count allows us to limit the number of times a message can be sent around the network, essential to prevent older messages from clogging up the system. The last three fields contain addresses of devices. It may appear unnecessary to provide these values as the messages are wrapped in an IP packet which already contains these values. These are necessary as when messages are relayed, the IP packet contains the address of the most recent sender, overwriting the original information. The source address contains the address of the origin device, the target address contains the address of the final destination, and the next address contains the address of the next device to propagate the message to.

Advertisement Messages

The advertisement message is used to advertise the presence of a new device within the distributed system. This message contains no additional fields than those described in table 4.3. When the advertisement message subtype is broadcast, the source address is set to the new device's IP address. At least one host must be known to advertise on the network, and this host's address will be placed into the next address field. The target address field is not used for broadcasts as we have no specific target that we wish to send our message to. Once a node receives an advertisement broadcast, it will check whether that host is already known, if so, the message is ignored, and propagated, otherwise if the host is not yet known, the node first adds the new device to its list of known hosts, and then propagates the message. When the acknowledgement is sent, the target address is set to the address of the new device, which ensures that any other nodes which may have advertised do not interpret the acknowledgement as regarding their broadcast. Once the acknowledgement arrives at the new device, the node adds the host which sent the ack, but does not propagate the message.

Consensus Messages

The consensus message is used when a client or firewall submits a new firewall rule to the stub via IPC. This message contains one additional field which holds the hash of the last block. When a node proposes a new firewall rule, it must first submit the hash of it's last block, which is sent to all known hosts. On receipt of a consensus broadcast, the node calculates the hash of the last block on it's chain, and if it matches the hash sent in the message it sends an ack, and appends the origin node's address to a list of pending rules. The node then forwards the message to all known hosts. After broadcasting the message, the origin host resets a counter,

which is incremented every time it receives an ack from one of it's known hosts. If this counter reaches atleast half of it's known hosts in some bounded time, it is deemed to have obtained consensus and may now distributed the new firewall rule.

Field Name	Data Type	Size (Bytes)
last_block_hash	Unsigned Char Array	Dependent on Hash Algorithm

Table 4.4: Additional Fields in a Consensus Message

Rule Messages

The rule message is used to distribute a new firewall rule once a node has achieved consensus. Only the broadcast subtype is used for rule messages, as waiting for acknowledgements could halt the system at the time of an attack, which would prevent the node from responding to other attacks. Upon receiving a rule broadcast the node will check to see if has previously approved the transaction by looking up the rule's origin in its list of pending rules. If the origin exists in the pending rules then the new block will be added to the chain, including its author and the last hash, then propagated to all of the node's known hosts.

Field Name	Data Type	Size (Bytes)
rule	FirewallRule	Dependent on machine architecture

Table 4.5: Additional Fields in a Rule Message

4.3.2 IPC

There are four different types of IPC message: rule, enable, disable, shutdown. The rule type is intended to be sent by a firewall, and contains a new firewall rule to be submitted to the distributed system. The enable and disable message types simply control the status of the stub, if it is believed that a host on the network may be malfunctioning (i.e. a firewall is sending bogus rules), it can be disabled and then re-enabled once the fault is fixed. Finally the shutdown message is used to properly terminate the stub. Upon receiving the shutdown message, all sockets will be closed, the IPC tunnel will be destroyed and the application will wait for all threads to terminate before exiting. To ensure messages can be parsed correctly, all of these message types follow the same structure and simply do not use the redundant fields.

Field Name	Data Type	Size (Bytes)
message_type	IPCMessageType	1
rule	FirewallRule	Dependent on machine architecture

Table 4.6: Structure of IPC Messages

4.4 ACR Protocol

As described in section 4.3.1, the system supports three message types: advertisement, consensus and rule, which form the ACR protocol.

4.4.1 Advertisement

The advertisement message is sent at startup and is mutually exclusive from the other messages, as it is executed automatically and does not affect the consensus or rule transaction stages. The advertisement message is sent when the framework is executed. The framework will send an advertisement broadcast to all known hosts, each of which will relay the message to each of

their known hosts, provided that the hop count has not yet been exceeded. The known host will then forward the message to all of its known hosts, propagating the message throughout the network. Upon receiving an advertisement message, the host checks if the host is known, if not, it is added, and an acknowledgement is returned.

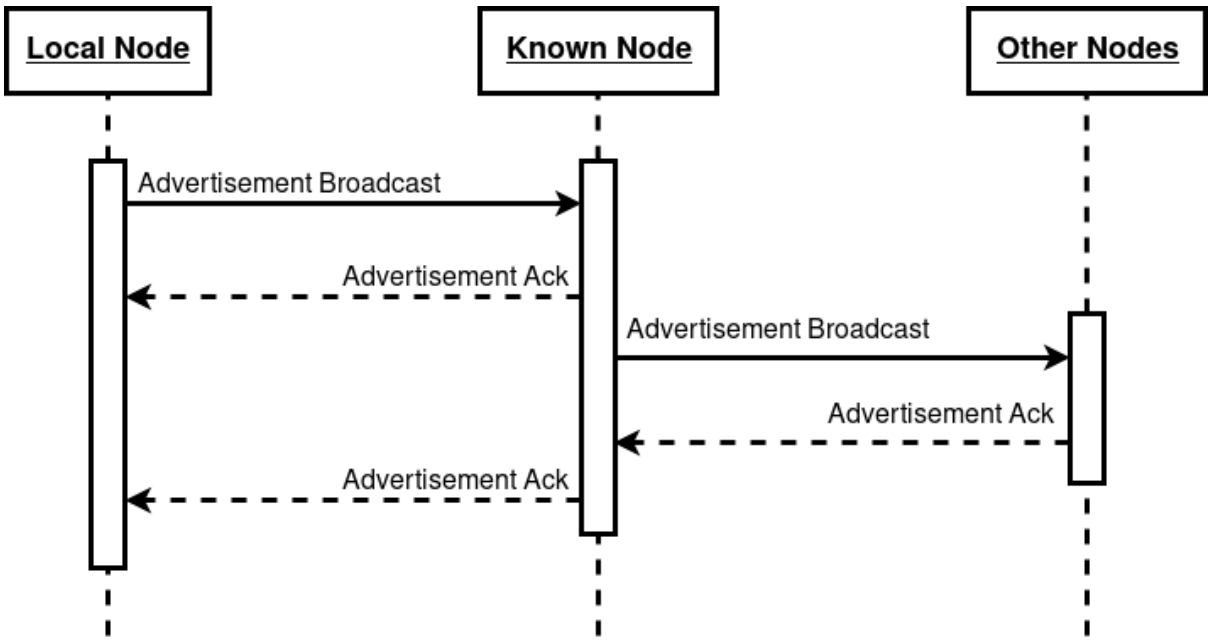

Figure 4.2: Advertisement Sequence Diagram

4.4.2 Consensus-Rule

The consensus and rule messages are sent when a client or firewall send an IPC message to the local stub. Upon receiving the rule, the stub will marshal the parameters into a network message, which is then sent to all known hosts. When a node receives the consensus message, it will compare the hash in the message to the hash of the previous block on its chain, if the hashes match, the block is considered valid and an acknowledgement is returned. The broadcast is then forwarded to all of the hosts known by that node. When the origin node receives an acknowledgement, it will check if that host is known. If it is not, then it will not increment the ack count, as it would interfere with the consensus calculation, however it will still receive the rule, if consensus is achieved. The origin node will then wait for some timeout, after which it will test if a sufficient number of acknowledgements have been received in comparison to the number of known hosts. If this test passes (e.g. at least half of the known hosts must acknowledge), then a rule message is sent. No acknowledgements are sent upon receipt of a rule message, but nodes will continue to propagate the message until the hop limit is reached.

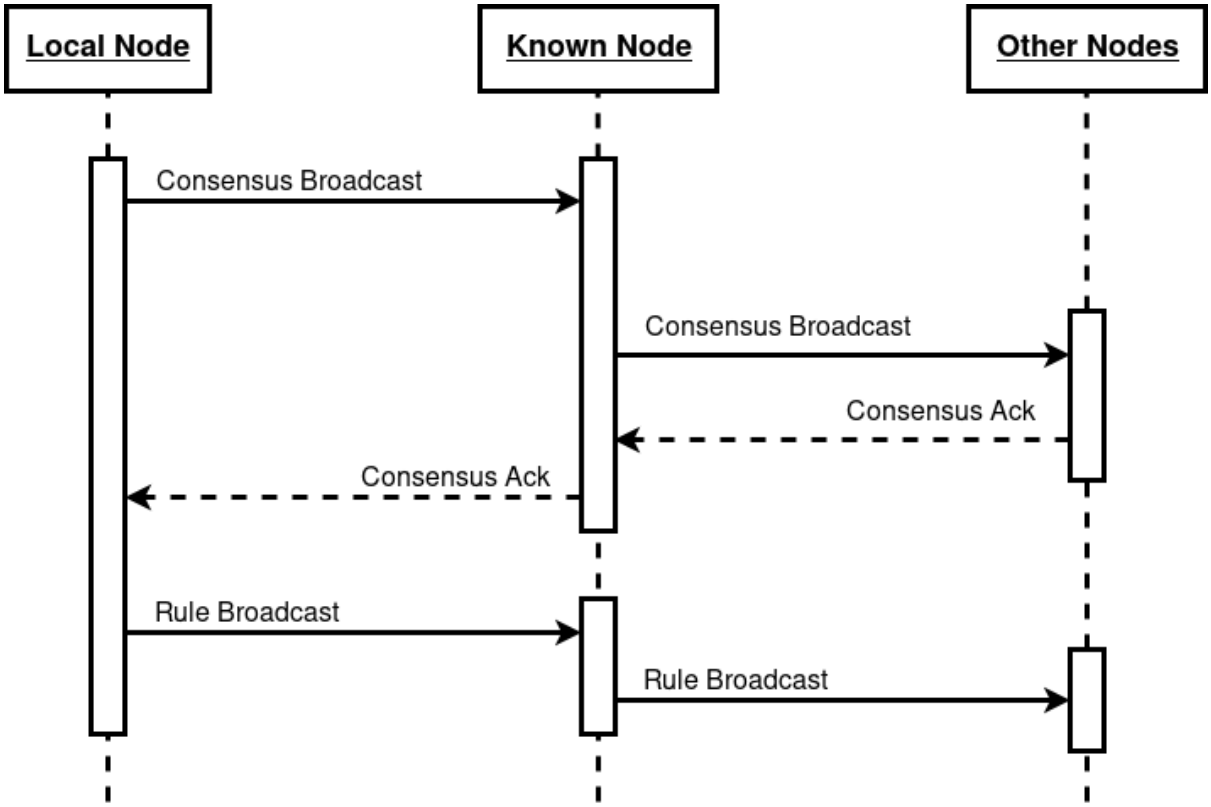

Figure 4.3: Successful Consensus-Rule Sequence Diagram

Chapter 5

Implementation

5.1 Language

The logic within this project could be achieved using a range of programming languages, however such a system needs to be reliable and fast, which narrows down the applicable languages. As cross compatibility is desirable, Java[19] or Python[20] may seem like the best options as Java executes entirely within the Java Virtual Machine (JVM)[21] and Python is executed using an interpreter. Each of these technologies provide a suitable abstraction such that if the VM or interpreter is supported by an OS, it should work. The drawbacks associated with this are that a VM or interpreter introduce a large memory overhead, something that is not desirable given this framework is intended to be implemented into edge devices, additionally, with such abstractions we lose both speed and control. Rust[22] is a modern language focused on performance and reliability, however its immaturity leaves a number of features missing or poorly implemented as it currently relies heavily on community submitted modules. One advantage of Rust however is its low memory footprint, and memory management, which enables small binary executables to be built, ideal for this project.

The final two languages considered were C[23] and C++[24]. It may seem an obvious choice to use C++ over C, however for this project very few features of C++ would be useful, and would simply add additional steps in the program to bridge between Portable Operating System Interface (POSIX) functions (implemented in C) and C++ types and syntax. One advantage that C has over C++ is its existing dominance in operating systems. Deemed the *system programming language*, the Linux[25] and NetBSD[26] kernels are written in pure C, along with the vast majority of the FreeBSD[27] and Windows[28] kernels. This means that using C enables this project to be compiled into an operating system, such that the distributed system is automatically instantiated on boot in kernel space and cannot be stopped or interrupted by a malicious application or user. Furthermore, writing this project in C allows wrappers to be written for all other languages, C++ can be integrated with no changes, and modules can be written for most other languages including Java and Python.

5.2 IPC

In order to achieve cross-platform IPC, different approaches were required for Portable Operating System Interface (POSIX) compliant operating systems and Windows[28] operating systems. Each OS type offers a number of IPC methods, which are briefly discussed the following sections.

The stub provides the following abstractions for interfacing with native IPC. The implementation methods are discussed in their relative sections below.

Listing 5.1: IPC API

```
int init_ipc_server(void);
int init_ipc_client(void);
int connect_ipc(void);
int cleanup_ipc(void);
int send_ipc_message(IPCMessage *message);
int recv_ipc_message(IPCMessage *message);
```

5.2.1 POSIX

POSIX systems provide four interfaces for IPC: pipes, FIFO, message queues and sockets. Although pipes do allow processes to communicate, the primary purpose of pipes is to obtain the output of one command and pass this output into a second command, which is not suitable for our application. FIFO and message queues operate very similarly, whereby each method creates some file in the filesystem which can then be referenced by any process. FIFO is the predecessor to message queues, and hence different operating systems have varying implementations for FIFO, however as message queues are a POSIX standard, all implementations provide the same interface, greatly simplifying the processes of making this project cross compatible. For this reason, message queues were used for POSIX IPC. The final method is via UNIX sockets. The socket interface is primarily designed for networked communication, but also supports IPC. Unfortunately, as sockets are intended for use in networks, some implementations still use the NIC to communicate even locally, which slows down communication, and adds additional points of failure.

Listing 5.2: Creating the message queue on a POSIX OS

```
mqueue = mq_open("/dfw", O_CREAT | O_RDWR, 0644, &attr);
```

Whilst testing this application it was discovered that nobody had yet created an interface for POSIX message queues in Java, so a secondary project was created, called Jmq [29] which provides an interface for POSIX message queues in Java using the Java Native Interface (JNI). This enables a stub to be created in Java by passing messages using the following syntax:

Listing 5.3: Example of Jmq

```
Jmq_attr attrs = new Jmq_attr(0, 2, 1, 0);
Jmq my_queue = new Jmq("/dfw", Jmq.O_CREAT | Jmq.O_RDWR, 0644, attrs);
my_queue.unlink();
```

5.2.2 Windows

Windows[28] offers three methods of IPC: Anonymous Pipes, Named Pipes and sockets. Anonymous pipes allow half-duplex communication, which is not ideal for this project, as it is necessary for the client / firewall to send instructions to the stub, and for the stub to send messages back to the client. Furthermore, Hart[30] explains the fundamental issue with anonymous pipes: they have no identifier associated with them, which creates a new problem of communicating the pipe handle to any clients wishing to connect. Named pipes are full-duplex, message oriented pipes than can operate locally or over a network. The difference between anonymous and named pipes is that a named pipe can be identified using a string which represents a file path, this enables both client and stub to connect directly to the pipe. The final method is via sockets which act similarly to UNIX sockets, and hence have the same pitfalls as UNIX sockets mentioned in section 5.2.1.

Listing 5.4: Creating the message queue on Windows

```
mqueue = CreateNamedPipe(TEXT("\\\\.\\pipe\\dfw"),
                         PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
                         PIPE_TYPE_MESSAGE, 1, 0, 0, 0, NULL);
```

5.3 Sockets

Network sockets are the primary method allowing processes to communicate over a network, however similarly to IPC, POSIX and Windows system provide slightly different socket interfaces. A numeric port is bound to the socket which enables the network stack to communicate with the socket, and hence the process. The main difference is that Windows requires the initialisation of the WinSock[31] library. Thankfully other than the initialisation of WinSock, the rest of the interface provided by WinSock is very similar to the basis of POSIX sockets, which are known as Berkeley sockets. This similarity means that only small modification need to be made to the source code. After creating the socket, it is then bound to the numeric port. The default ports used for this project are 8070 and 8071.

The stub provides the following abstractions for interfacing with native sockets.

Listing 5.5: Socket API

```
int init_sockets(void);
int cleanup_sockets(void);
socket_t create_socket(void);
void close_socket(socket_t sock);
int bind_socket(socket_t sock, int port);
int send_to_socket(socket_t sock, void *message, size_t length, int flags,
                   struct sockaddr_in dest_addr);
int recv_from_socket(socket_t sock, void *buffer, size_t length, int flags);
```

5.4 Network

As the project is focussed on creating a distributed system, the vast majority of the implementation is based on networking. Thankfully, due to the previous abstractions created regarding sockets, the network implementation consists primarily of the business logic. The one major difference between the UNIX network stack, and the Windows network stack, is querying the OS to retrieve information from NICs. Implementing this required an entirely different system for each operating system. On UNIX, the `getifaddrs` function was used, whereas on Windows, the `GetAdaptersAddresses` function was used.

The stub provides the following abstractions for interfacing with the native network stack.

Listing 5.6: Network API

```
int get_local_address(char *buffer);
int get_acks(void);
int reset_acks(void);
int set_ack(char *addr);
int load_hosts_from_file(const char *fname);
int save_hosts_to_file(const char *fname);
int add_host(char* addr);
int check_host_exists(char *addr);
int get_host_count(void);
int init_net(void);
int cleanup_net(void);
int send_to_host(char *ip_address, void *message, size_t length);
int send_advertisement_message(AdvertisementMessage *message);
int send_to_all_advertisement_message(AdvertisementMessage *message);
```

```
15 int recv_advertisement_message(void *buffer);
16 int recv_advertisement_broadcast(AdvertisementMessage *message);
17 int recv_advertisement_ack(AdvertisementMessage *message);
18 int send_consensus_message(ConsensusMessage *message);
19 int send_to_all_consensus_message(ConsensusMessage *message);
20 int recv_consensus_message(void *buffer);
21 int recv_consensus_broadcast(ConsensusMessage *message);
22 int recv_consensus_ack(ConsensusMessage *message);
23 int send_rule_message(RuleMessage *message);
24 int send_to_all_rule_message(RuleMessage *message);
25 int recv_rule_message(void *buffer);
26 int recv_rule_broadcast(RuleMessage *message);
27 int poll_message(void *buffer, size_t length);
```

5.5 Multithreading

In order for the stub to be capable of consecutively send and receive messages over both the network, and via IPC, the framework had to be multithreaded. Unsurprisingly, POSIX and Windows use different thread models, which means there is no common threading interface. Thankfully, the POSIX thread model has been ported to Windows in the form of the POSIX Threads for Windows project[32]. This allows the source code to be written identically for all operating systems, the only difference being that Windows will be linked with the necessary Dynamic-Link Library (DLL).

5.6 Blockchain

Once the previous APIs had been created, the blockchain aspect of the project could be implemented. As the structure of these blocks have already been discussed in section 4.2.2, this section will focus on how the functionality of this was implemented, primarily the hashing of blocks.

5.6.1 Hash Algorithms

The first decision that had to be made was which hash algorithm to implement. A wide range of hash algorithms are available, however the most popular three are Secure Hash Algorithm (SHA), MD5 and BLAKE. The SHA hash suite boasts a number of algorithms which are classed into four generations: SHA-0, SHA-1, SHA-2 and SHA-3. The SHA-0 and SHA-1 algorithms are the least secure, and are now deemed insecure as collisions been found, which would enable a malicious block to contain the hash of a valid block. At the time of writing this, there are no known collisions for the SHA-2 or SHA-3 families. The MD5 hash algorithm is also insecure for the same reason as SHA-1 and is hence not appropriate for this project. The final algorithm family is BLAKE, which consists of three families: BLAKE, BLAKE2 and BLAKE3. There are no known collisions within any of the BLAKE hash families, however the BLAKE algorithm has not been tested for blockchain applications, whereas the SHA family has. For this reason, this project will utilise the same hash algorithm as Bitcoin [11], SHA-256.

Hash Algorithm		Output Size	Collisions found?	Used By
MD5		128	Yes	None
SHA-0		160	Yes	None
SHA-1		160	Yes	None
SHA-2	SHA-224	224	No	None
	SHA-256	256	No	Bitcoin[11], Bitcoin Cash
	SHA-384	384	No	None
	SHA-512	512	No	None
	SHA-512/224	224	No	None
	SHA-512/256	256	No	None
SHA-3	SHA3-224	224	No	None
	SHA3-256	256	No	None
	SHA3-384	384	No	None
	SHA3-512	512	No	None
BLAKE	BLAKE-224	224	No	None
	BLAKE-256	256	No	None
	BLAKE-384	384	No	None
	BLAKE-512	512	No	None
BLAKE2	BLAKE2s	256	No	Nano[33]
	BLAKE2b	512	No	None
BLAKE3		Variable	No	None

Table 5.1: Comparison of Hash Algorithms

5.6.2 Hash Implementation

When implementing the desired hash algorithm, there were to primary options, to implement the algorithm from scratch, or to use an existing library. If the algorithm was to be implemented from scratch, then the resulting application would requires less dependencies and would overall result in a more self-contained binary. Using a library would result in one additional dependency, however the vast number of algorithms provided by most libraries would allow the entire blockchain and protocol to be easily updated should a more secure algorithm be preferable. Furthermore, most libraries are thoroughly tested and well engineered to guarantee the most efficient operation and make sure the algorithms can handle every edge case. For this it was decided that an external library would be implemented, and a breakdown of the considered libraries is provided below [34].

Cryptography Library	Algorithm Families	Language	Compatible OSs	First Created
OpenSSL (libcrypto)[18]	9	C	28	1998
wolfSSL (wolfCrypt)[35]	6	C	26	2006
cryptlib[36]	6	C	37	1995
GPG (Libgcrypt)[37]	11	C	Unknown	1999

Table 5.2: Comparison of Cryptographic Libraries

From the previous table, it is apparent that libgcrypt supports the most algorithms from the comparison given in [34], however for cross compatibility, cryptlib boasts 37 supported operating systems. OpenSSL's libcrypto sits in the middle of the two, supporting nine hash algorithm families and 28 different operating systems. Additionally, libcrypto is has been the default cryptography library distributed with most Linux, BSD and Windows operating systems, meaning it has received a large amount of support, and faced tight scrutiny by the likes of Microsoft. This made libcrypto the optimum library for use in this project.

5.7 Fault Tolerance

In regards to fault tolerance, this project was extremely limited, as it ran as a high-level application on general operating systems, meaning the operating systems themselves provided no guarantees of performance or reliability. With regards to fault tolerance during operation, the only reasonable measure was to ensure no malformed data could be propagated throughout the system. Such malicious data could cause the local stub, or even worse, remote stubs to crash, rendering the entire system non-functional whilst the malicious data is removed from each system. This measure was achieved by using the fail-early principle in all functions within the system. When each function is executed, specific checks take place, such as ensuring buffers are large enough to store specified data, and that the data provided is valid, and of the required type. Many of these checks are provided by the standard library within the C language, such as verifying the string representation of an IP address is correctly formatted when parsing to a binary representation of the network address. The vast majority of these functions return an integer to represent the whether the function exited normally, or if it exited with an error, which was replicated in many of the functions provided by this application. Returning the exit status enables any calling functions to know that the function did not exit normally, which means the output of that function will contain incorrect data, and so the path of execution can no longer follow the normal path, but must instead fail early, and return to the top-level calling function.

In addition to the fail-early principle, the each node maintains a log of all complete transactions that have taken place on that node. Once a node receives a either an advertisement, or a new rule, it is logged immediately. This means that should the operating system fail, or the system suffers from an unrecoverable fault, the operating system can immediately restart the system (which can run as a daemon) which will then load all known hosts into memory, and rebuild the chain from the transaction log.

Chapter 6

Testing

Testing the system required tests to be categorised into two types: local and network. Local tests consist of any functions which take place on the local node, such as hashing of blocks and controlling network sockets. These local tests could be easily tested using unit tests. Network tests consist of protocol tests that can only be tested over a physical network. Network tests are much harder to strategically test, which means unit tests would require careful synchronisation, such that when one node was testing it's send feature, another node would need to be ready to accept a message, for this reason, networks tests did not use unit tests.

6.1 Local Testing

As previously mentioned, local tests were carried out with unit tests. There are a number of uni test frameworks for C, which all offer very similar functionality, and for this reason the decision process will not be discussed here, however in the end the cmocka[38] framework was identified as the best framework for this project. All tests were carried out on six different devices, which contained different architectures, hardware, operating systems and distributions. Furthermore, four out of the five devices tested were low-power single board computers including Raspberry Pi's and Orange Pi's, two of which only had 512MB of memory, providing an additional an test to check how demanding the system would be. A summary of these devices is given below.

Architecture	Network Hardware	Operating System	Distribution
Intel x86	Intel	Linux 5.11.12	Artix
Intel x86	Realtek	Windows 10	Home
ARMv7	Allwinner	Linux 5.3.5	Ubuntu
ARMv7	Allwinner	Linux 5.10.12	Armbian
ARMv7	Broadcom	NetBSD 8.0	N/A
ARMv8	Broadcom	FreeBSD 13	N/A

Table 6.1: Devices used for Testing

A summary of the tests used is provided below.

Cryptography Library	Algorithm Families	Language	Compatible OSs	First Created
OpenSSL (libcrypto)[18]	9	C	28	1998
wolfSSL (wolfCrypt)[35]	6	C	26	2006
cryptlib[36]	6	C	37	1995
GPG (Libgcrypt)[37]	11	C	Unknown	1999

Table 6.2: Unit Test Results

As can be seen, all unit tests passed on all tested operating systems.

6.2 Network Testing

In order to test the operation over the network, a physical network was created using a switch and all of the devices described in table 6.1. A diagram of this topology is pictured below.

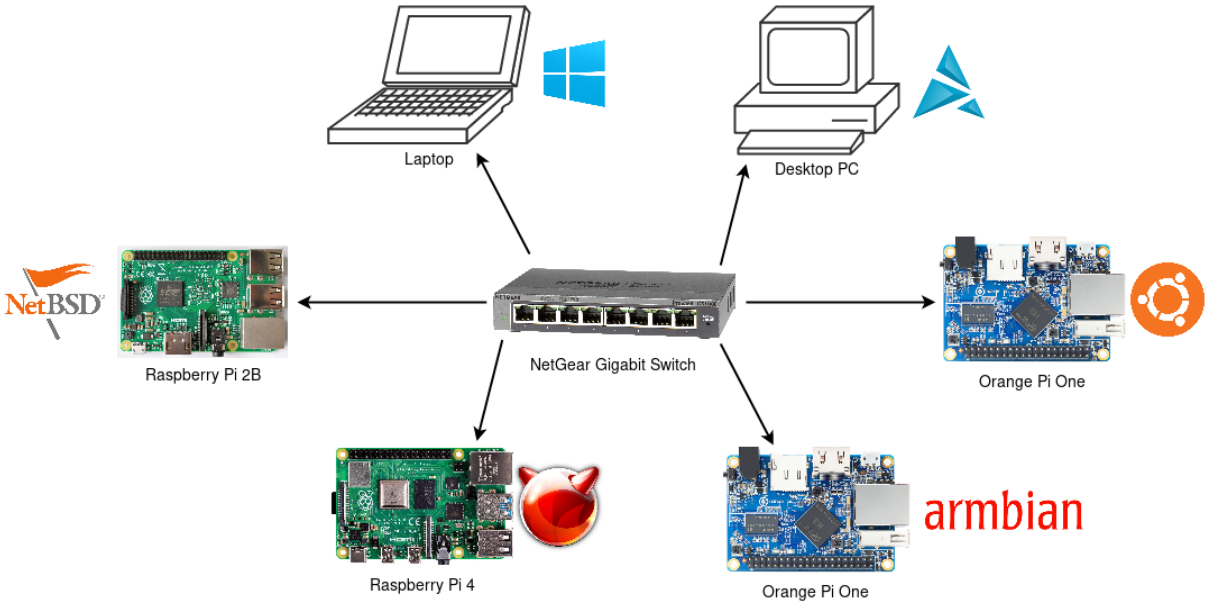

Figure 6.1: The network layout used for testing

A summary of the tests user is provided below.

Cryptography Library	Algorithm Families	Language	Compatible OSs	First Created
OpenSSL (libcrypto)[18]	9	C	28	1998
wolfSSL (wolfCrypt)[35]	6	C	26	2006
cryptlib[36]	6	C	37	1995
GPG (Libgcrypt)[37]	11	C	Unknown	1999

Table 6.3: Network Test Results

6.3 Attack Simulation and Performance Testing

Following successful unit and network tests, an attack was simulated on the system to measure how long it took for a rule to be created, advertised, consensus gained and then transmit to all other nodes. The attack would be reported to the stub of one node, and the local timestamp would be recorded at each step of the process. Ideally, timestamps would be collected at all nodes, however due to variations in clocks, synchronisations and architectures, the resulting data was inaccurate, and hence it was decided that timestamps would only be taken from the local node.

Stage	Time (s)
Received firewall rule via IPC	1618330473.99529
Sent all consensus broadcasts	1618330473.99558
Received all consensus acks	1618330474.49588
Sent new firewall rule to all nodes	1618330474.49599
New block added to chain	1618330474.49631

Table 6.4: Attack Simulation Results

The times provided in the above table are given in seconds since the epoch. Running on a standard operating system, not designed for real-time, the entire transaction took place in 501.02 ms, and out of that duration, 500 ms is a hard-coded delay to wait for acknowledgements for the consensus. This indicates that with regards to processing time, only 1.02 ms is required to carry out the required calculations, hashing and assembling the chain.

6.4 Checking Memory Leaks

valgrind

Chapter 7

Evaluation

Chapter 8

Conclusion

Glossary

blockchain A growing list of records, called blocks, that are linked using cryptography. 3, 6–8, 11, 17, 24, 25

cryptocurrency A digital currency produces by a public network. 6

hashing The practise of taking data and representing that data as a fixed-length string. 7

ledger A record of all transactions executed on a particular cryptocurrency. 6

malware Malicious computer software that interferes with normal computer function or sends personal data about the user to unauthorised parties. 5

middleware Software that functions at an intermediate layer between applications and the operating system to provide distributed functions. 3, 6, 9

phishing Sending an email that falsely claims to be from a legitimate organisation, usually combined with a threat or request for information. 5

smishing Sending a text message via SMS that falsely claims to be from a legitimate organisation, usually containing a link to a malicious website. 5

stub A piece of code that is used to marshal parameters for transmission across the network. 6, 8–10, 14, 17–19, 21–24, 26, 28

Acronyms

API Application Programming Interface. 24

APT Advanced Persistent Threat. 5

DMZ Demilitarised Zone. 3, 12, 13

IP Internet Protocol. 15, 17, 26

IPC Inter-Process Communication. 4, 14, 16–19, 21–24

NCSC National Cyber Security Center. 5

NIC Network Interface Card. 15, 22, 23

OS Operating System. 6, 10, 14, 15, 21, 23

POSIX Portable Operating System Interface. 21–24

RPC Remote Procedure Call. 3, 10, 11

TCP Transmission Control Protocol. 12, 14

UDP User Datagram Protocol. 12, 14

VM Virtual Machine. 21

Bibliography

[1] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.

[2] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *Business & Information Systems Engineering*, vol. 59, pp. 183–187, 2017.

[3] R. L. Rivest and B. Lampson, "SDSI-a simple distributed security infrastructure," Crypto, 1996.

[4] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *International conference on reliable software technologies*, pp. 38–57, Springer, 1996.

[5] NCSC and CSE, "Advisory: APT29 targets COVID-19 vaccine development." `https://www.ncsc.gov.uk/files/Advisory-APT29-targets-COVID-19-vaccine-development-V1-1.pdf`, 2020.

[6] P. K. Ozili and T. Arun, "Spillover of covid-19: impact on the global economy," *SSRN 3562570*, 2020.

[7] H. S. Lallie, L. A. Shepherd, J. R. Nurse, A. Erola, G. Epiphaniou, C. Maple, and X. Bellekens, "Cyber security in the age of COVID-19: A timeline and analysis of cybercrime and cyber-attacks during the pandemic," *Computers & Security*, vol. 105, 2021.

[8] BBC, "Newcastle university cyber attack 'to take weeks to fix'." `https://www.bbc.co.uk/news/uk-england-tyne-54047179`, 2020. Accessed: 01/04/2020.

[9] BBC, "Northumbria university hit by cyber attack." `https://www.bbc.co.uk/news/uk-england-tyne-53989404`, 2020. Accessed: 01/04/2020.

[10] D. P. Reed, *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." `https://bitcoin.org/bitcoin.pdf`, 2008.

[12] V. Buterin, "A next generation smart contract & decentralized application platform." `https://whitepaper.io/coin/ethereum`, 2013.

[13] J. N. Gray, "An approach to decentralized computer systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6, pp. 684–692, 1986.

[14] E. S. Al-Shaer and H. H. Hamed, "Modeling and management of firewall policies," *IEEE Transactions on network and service management*, vol. 1, no. 1, pp. 2–10, 2004.

[15] E. Dulaney and C. Easttom, *CompTIA Security+ Study Guide: Exam SY0-501.* John Wiley & Sons, 7 ed., 2018.

[16] T. Lammle, *CompTIA Network+ Study Guide: Exam N10-007*. John Wiley & Sons, 4 ed., 2018.

[17] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*. Pearson, 3 ed., 2015.

[18] OpenSSL Software Foundation, "Libcrypto API - OpenSSLWiki." `https://wiki.openssl.org/index.php/Libcrypto_API`, 2014. Accessed: 07/04/2020.

[19] Oracle, "Java — oracle." `https://www.java.com`, 2021. Accessed: 08/04/2021.

[20] Python Software Foundation, "Welcome to python.org." `https://www.python.org/`, 2021. Accessed: 08/04/2021.

[21] Oracle, "Java virtual machine technology overview." `https://docs.oracle.com/en/java/javase/16/vm/java-virtual-machine-technology-overview.html`, 2021. Accessed: 08/04/2021.

[22] Rust Team, "Rust programming language." `https://www.rust-lang.org/`, 2021. Accessed: 08/04/2021.

[23] B. W. Kernighan and D. M. Ritchie, *The C programming language*. 1988.

[24] B. Stroustrup, *The C++ programming language*. Pearson Education India, 2000.

[25] L. Torvalds, "Github - torvalds/linux: Linux kernel source tree." `https://github.com/torvalds/linux`, 2021. Accessed: 08/04/2021.

[26] NetBSD Foundation, "The NetBSD project." `https://http://netbsd.org/`, 2021. Accessed: 08/04/2021.

[27] FreeBSD Foundation, "The FreeBSD project." `https://www.freebsd.org/`, 2021. Accessed: 08/04/2021.

[28] Microsoft, "Explore windows 10 os, computers, apps & more — microsoft." `https://www.microsoft.com/en-gb/windows`, 2021. Accessed: 08/04/2021.

[29] A. D. Bruce, "Jmq - Java POSIX Message Queue interface." `https://gitlab.com/adamdb/jmq`, 2021. Accessed: 08/04/2021.

[30] J. M. Hart, *Windows system programming*. Pearson Education, 2010.

[31] Microsoft, "Windows sockets 2 - win32 apps — microsoft docs." `https://docs.microsoft.com/en-us/windows/win32/winsock`, 2021. Accessed: 08/04/2021.

[32] R. P. Johnson, "POSIX threads for windows / wiki home." `https://sourceforge.net/p/pthreads4w/wiki/Home/`, 2021. Accessed: 08/04/2021.

[33] C. LeMahieu, "Nano: A feeless distributed cryptocurrency network." `https://content.nano.org/whitepaper/Nano_Whitepaper_en.pdf`, 2015.

[34] "Comparison of cryptography libraries - wikipedia." `https://en.wikipedia.org/wiki/Comparison_of_cryptography_libraries`, 2021. Accessed: 09/04/2020.

[35] wolfSSL inc., "wolfCrypt Embedded Cryptography Engine — wolfSSL Products." `https://www.wolfssl.com/products/wolfcrypt-2/`, 2020. Accessed: 09/04/2020.

[36] cryptlib, "Cryptlib - Encryption Security Software Development Toolkit." `https://www.cryptlib.com/`, 2015. Accessed: 09/04/2020.

[37] The GnuPG Project, "The GNU Privacy Guard." https://www.gnupg.org/software/libgcrypt/index.html, 2020. Accessed: 09/04/2020.

[38] "cmocka - uni testing framework for c." https://cmocka.org/, 2020. Accessed: 12/04/2020.