Using Blockchain to Create a Decentralised Security Model for Distributed Systems

Adam David Bruce

May 2021

Computer Science (Security and Resilience)

Supervisor: Dr John Mace

Word count: 14,993

Declaration

"I declare that this dissertation represents my own work except where otherwise stated."

Abstract

Network security has, and always will be one of the greatest concerns for universities and other educational institutions. The 2020 cyberattacks throughout the COVID-19 pandemic highlighted both the importance of cybersecurity in UK universities, but also the lack of modern security infrastructure deployed across campuses. This literature aims to provide a method for universities to share information on potential, or ongoing cyberattacks to other universities via a trustworthy channel, capable of operating independently from other security measures. This is achieved by creating middleware capable of running on existing devices on each campus that uses blockchain to establish a decentralised trust mechanism. We explore a number of methods to achieve this including examining existing technologies as well as designing and implementing our own. A thorough analysis of our implementation is provided and we conclude by suggesting a number of future improvements that could be made to improve both the efficiency and effectiveness of such a system.

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 University and Northumbria University, 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.

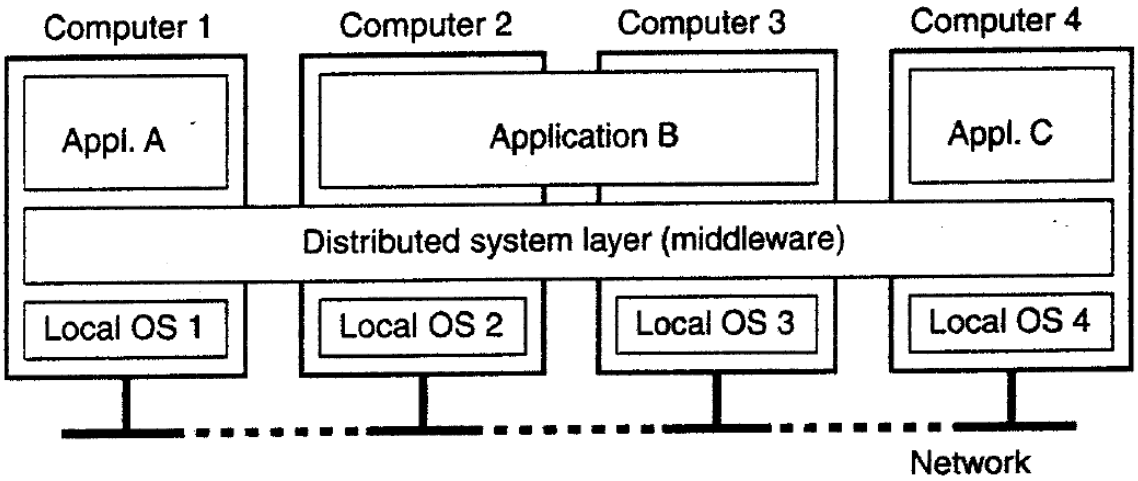

Figure 1.1: A Distributed System visualised as middleware [1]

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.

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.

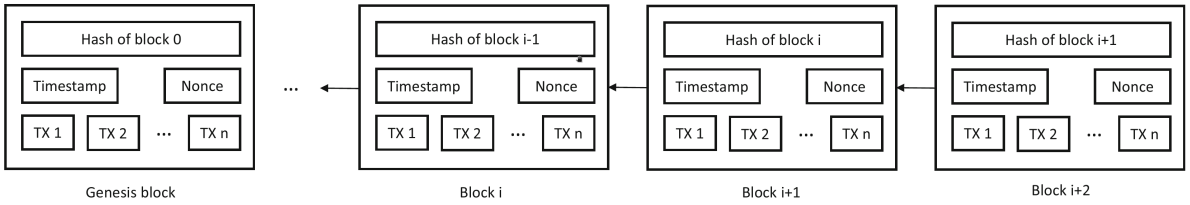

Figure 1.2: An example of a blockchain [2]

1.5 Aims and Objectives

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 the alteration of the aim to instead focus 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**.

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.

1.6 This Document

The following chapter will explain the background research conducted before starting this project. Once the necessary research has been documented, the design, implementation and testing of this project will be detailed. Finally, at the end of this document, an evaluation will be conducted followed by a conclusion to identify the success of this project and outline future work.

Chapter 2

Background

This section details the essential background research for this project. It looks at distributed systems, remote procedure calls, decentralised systems, blockchain, security, firewalls, fault tolerance, computer networks and inter-process communication.

2.1 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.

2.1.1 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 2.2. 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.

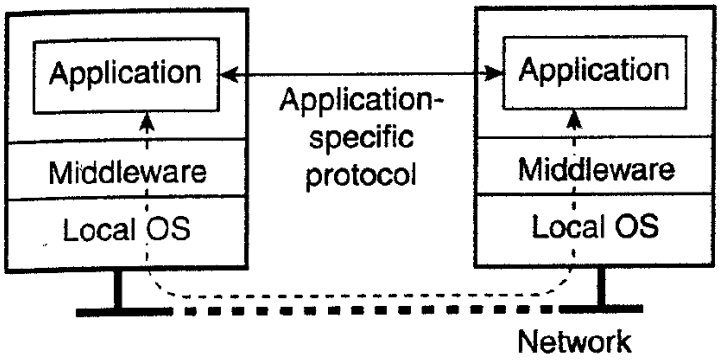

Figure 2.1: Application layer protocol running over middleware [1]

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.

2.1.2 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.

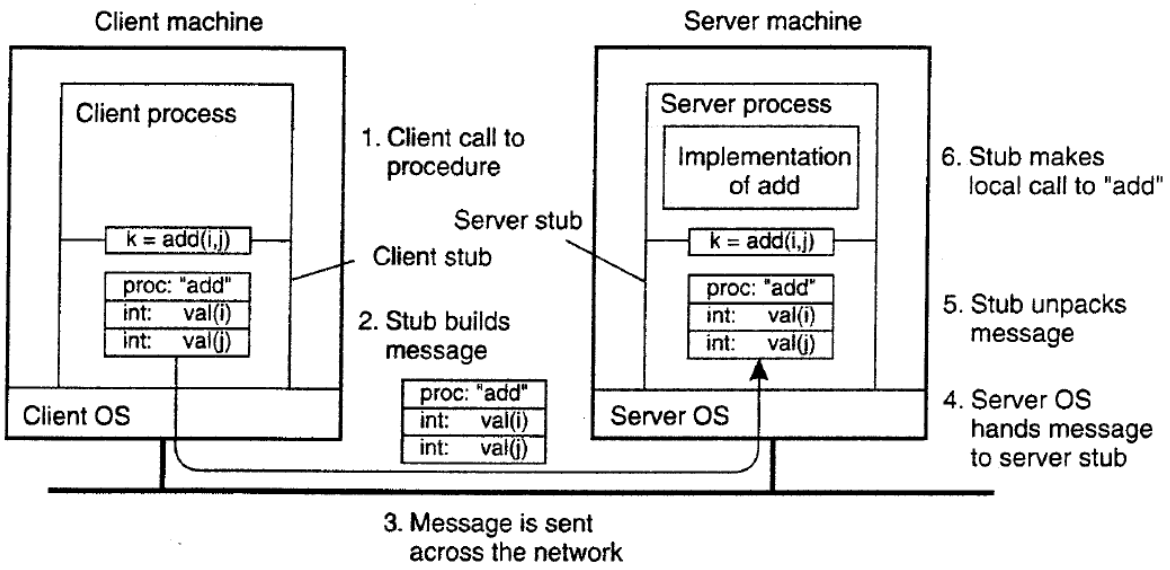

Figure 2.2: A breakdown of a RPC [1]

2.2 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.

2.3 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.

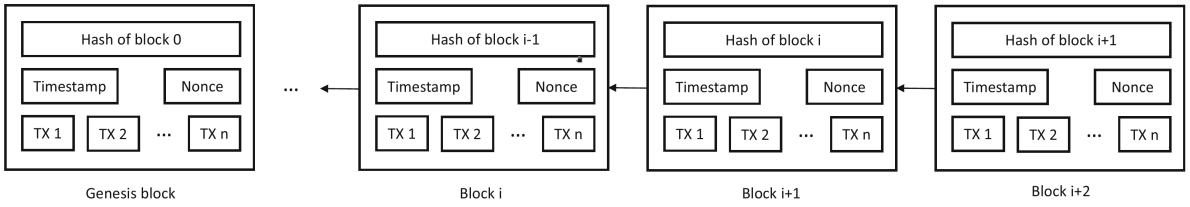

Figure 2.3: An example of a blockchain [2]

2.4 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 focused 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 2.1. The message system proposed by Rivets and Lampson contains only a type and dictionary of attributes.

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

Figure 2.4: 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
```

2.5 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 1.5, 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.

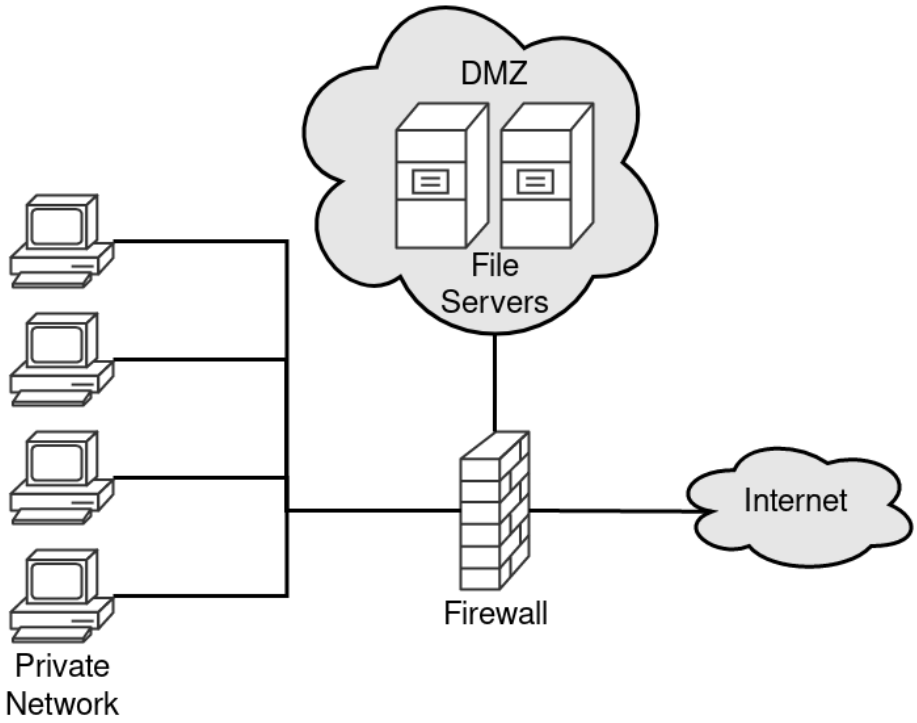

Figure 2.5: Using a firewall to create a DMZ

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.

2.6 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 this project 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.

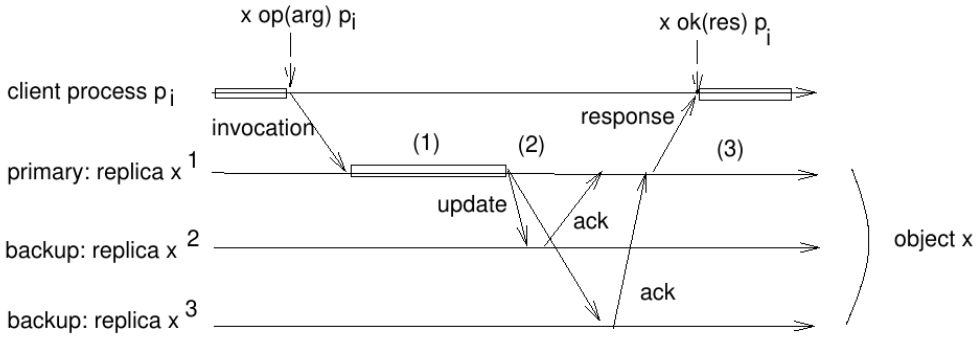

Figure 2.6: Primary backup technique [4]

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.

2.7 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.

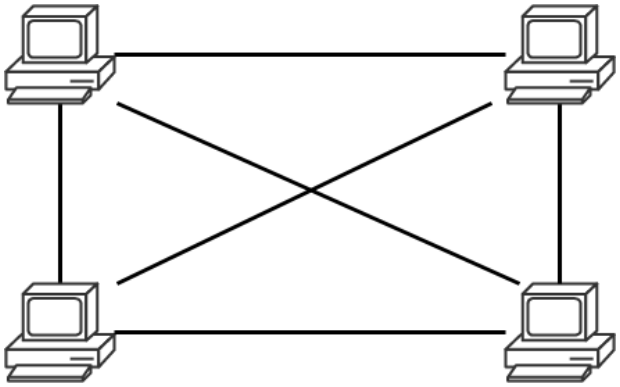

Figure 2.7: Mesh Topology

2.8 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.

The background research has identified how this project can utilise all of the mentioned technologies, along with a brief analysis regarding the relevance of each technology to this project.

Chapter 3

Design

This section will look at how the project was designed from the ground up to include the relevant features to meet the aim and objectives. The section covers the logical design of the system, protocols and structures used within the system.

3.1 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.

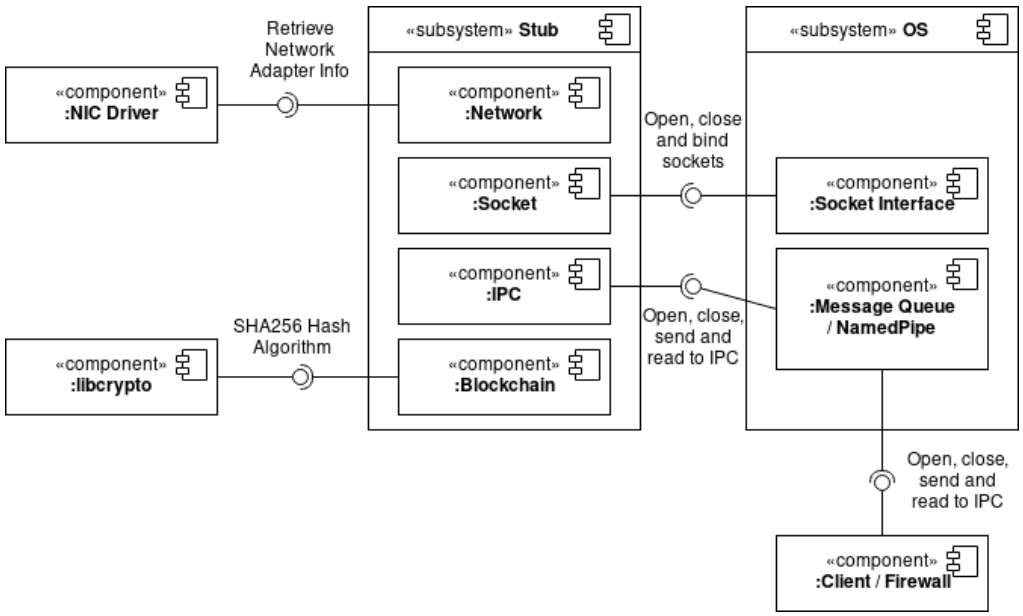

Figure 3.1: System Component Diagram

3.2 Data Structures

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

3.2.1 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 3.1: 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.

3.2.2 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 3.2: 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 3.2.1. 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.

3.3 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.

3.3.1 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 3.3: 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 the number of times a message can be sent around the network to be limited, 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 3.3. 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 there is no specific target that the message should be sent 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 also 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 distribute the new firewall rule.

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

Table 3.4: 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 3.5: Additional Fields in a Rule Message

3.3.2 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 the rule field is simply disregarded for all non-rule messages.

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

Table 3.6: Structure of IPC Messages

3.4 ACR Protocol

As described in section 3.3.1, the system supports three message types: advertisement, consensus and rule, which form the ACR protocol.

3.4.1 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. Upon receiving an advertisement message, the host checks if the host is known, if not, it is added, and an acknowledgement is returned.

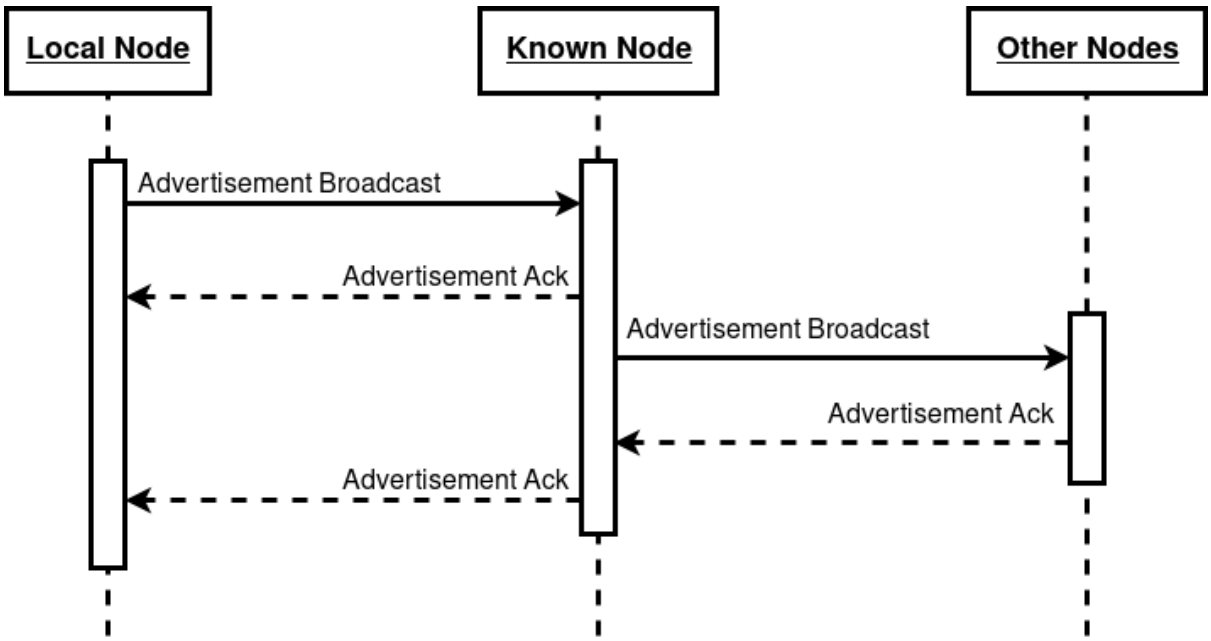

Figure 3.2: Advertisement Sequence Diagram

3.4.2 Consensus-Rule

The consensus and rule messages are sent when a client or firewall sends 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.

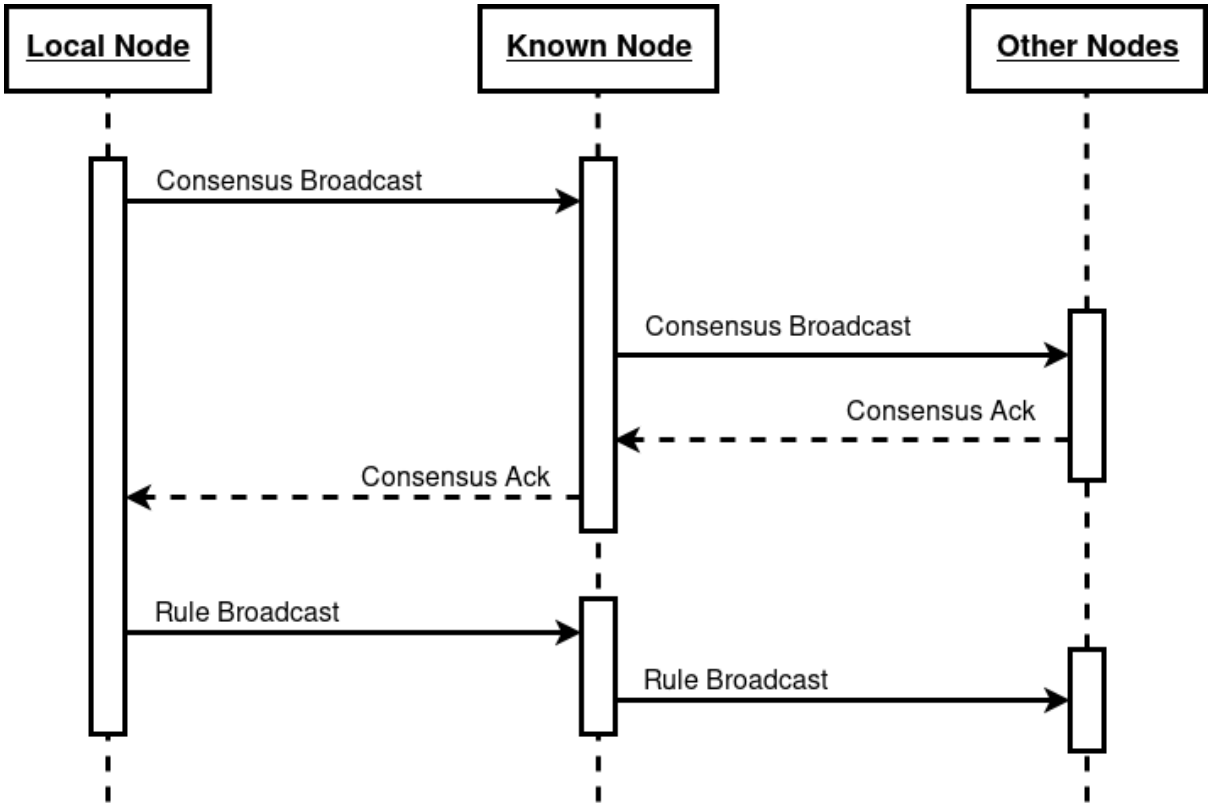

Figure 3.3: Successful Consensus-Rule Sequence Diagram

This section has provided the foundations for the software and protocol allowing an proof of concept to be implemented in the next chapter.

Chapter 4

Implementation

This section will discuss the different technologies used within the system and how they were incorporated into this project. A detailed analysis is provided whenever a decision was made in regards to using different technologies, tools or languages.

4.1 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.

4.2 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 in the following sections.

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

Listing 4.1: 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);
```

4.2.1 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 process 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 4.2: Creating the message queue on a POSIX OS

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

4.2.2 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[29] 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 4.2.1.

Listing 4.3: 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);
```

4.3 Sockets

Network sockets are the primary method allowing processes to communicate over a network, however similarly to IPC, POSIX and Windows systems 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[30] 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 modifications 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 4.4: Socket API

```
1 int init_sockets(void);
2 int cleanup_sockets(void);
3 socket_t create_socket(void);
4 void close_socket(socket_t sock);
5 int bind_socket(socket_t sock, int port);
6 int send_to_socket(socket_t sock, void *message, size_t length, int flags,
7                     struct sockaddr_in dest_addr);
8 int recv_from_socket(socket_t sock, void *buffer, size_t length, int flags);
```

4.4 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 4.5: Network API

```
1 int get_local_address(char *buffer);
2 int get_acks(void);
3 int reset_acks(void);
4 int set_ack(char *addr);
5 int load_hosts_from_file(const char *fname);
6 int save_hosts_to_file(const char *fname);
7 int add_host(char* addr);
8 int check_host_exists(char *addr);
9 int get_host_count(void);
10 int init_net(void);
11 int cleanup_net(void);
12 int send_to_host(char *ip_address, void *message, size_t length);
13 int send_advertisement_message(AdvertisementMessage *message);
14 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);
```

4.5 Multithreading

In order for the stub to be capable of consecutively sending and receiving 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[31]. 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).

4.6 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 3.2.2, this section will focus on how the functionality of this was implemented, primarily the hashing of blocks.

4.6.1 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 have 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[32]
	BLAKE2b	512	No	None
BLAKE3		Variable	No	None

Table 4.1: Comparison of Hash Algorithms

4.6.2 Hash Implementation

When implementing the desired hash algorithm, there were two 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 require 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 reason it was decided that an external library would be implemented, and a breakdown of the considered libraries is provided below [33].

Cryptography Library	Algorithm Families	Language	Compatible OSs	First Created
OpenSSL (libcrypto)[18]	9	C	28	1998
wolfSSL (wolfCrypt)[34]	6	C	26	2006
cryptlib[35]	6	C	37	1995
GPG (Libgcrypt)[36]	11	C	Unknown	1999

Table 4.2: Comparison of Cryptographic Libraries

From the previous table, it is apparent that libgcrypt supports the most algorithms from the comparison given in [33], 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 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.

4.7 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 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, each node maintains a log of all complete transactions that have taken place on that node. Once a node receives 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.

This section has provided a high-level explanation of how this project was implemented, along with the reasons particular technologies were used. Now a functional implementation has been created, the system can be tested.

Chapter 5

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 can 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 an attempts to test these using 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.

5.1 Local Testing

As previously mentioned, local tests were carried out with unit tests. There are a number of unit 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[37] 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 test to check how demanding the system would be. The operating systems were chosen as they all have strong market shares in networking components and technologies, in addition to being freely available and supporting a wide range of hardware. 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 5.1: Devices used for Testing

A summary of the tests used is provided below.

Name
Get block hash (invalid, buffer is null)
Get block hash (invalid, buffer too small)
Get block hash (invalid, block is null)
Get block hash (valid)
Get hash string (invalid, hash is null)
Get hash string (invalid, buffer too small)
Get hash string (invalid, buffer is null)
Get hash string (valid)
Initialise network stack (valid)
Clean up network stack (valid)
Send and receive message (valid)
Get Local IP Address (valid)
Load hosts from file (valid)
Add host (valid)
Initialise socket API (valid)
Clean up socket API (valid)
Create socket (valid)
Bind sending socket (valid)
Bind receiving socket (valid)
Bind socket (invalid, null socket)
Bind socket (invalid, port already in use)
Send and receive over socket (valid)

Table 5.2: Unit Tests

5.2 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 5.1. A diagram of this topology is pictured below. Due to the nature of this project, these network tests were carried out at various stages throughout the development processes. There was no formal agenda to these tests, instead a large number of scenarios were created using the pictured network. All of the tested scenarios were handled correctly, which included testing advertisements, both valid and invalid consensus messages and verifying if rules were successfully communicated between nodes.

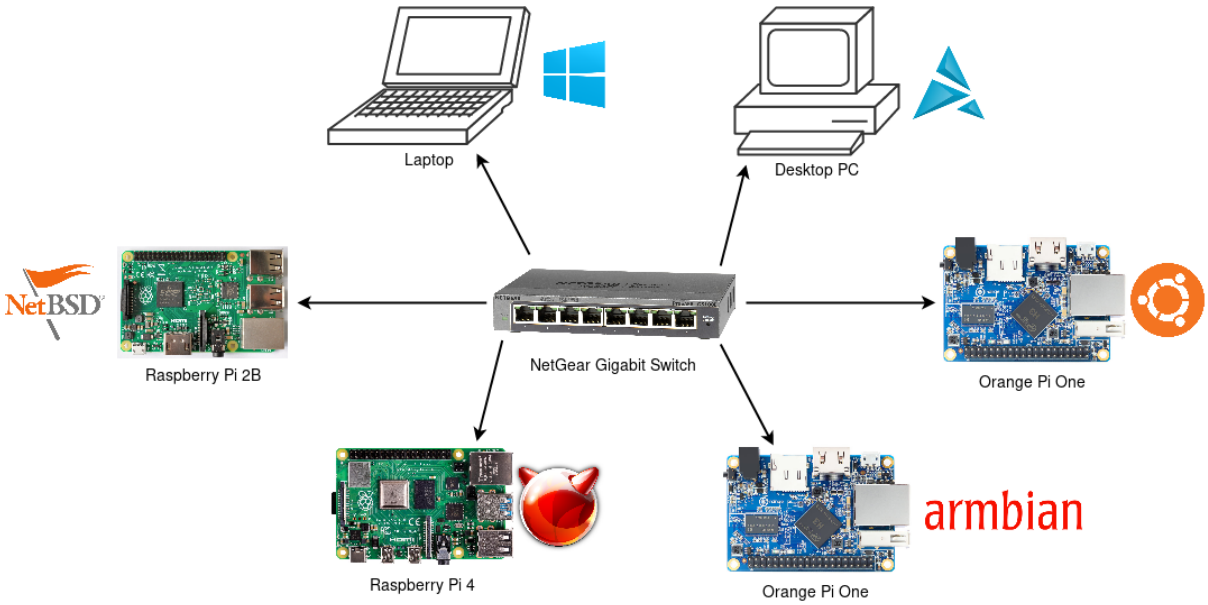

Figure 5.1: The network layout used for testing

5.3 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 5.3: 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.

5.4 Checking for Memory Leaks

Not only is it essential that the system is functionally and logically correct, but it must also correctly utilise system resources. The most important of these resources is memory, in particular, the heap where numerous dynamic allocations are made throughout the lifetime of the stub. As the system is designed to function for extended periods of time, it is essential that any memory allocated is correct freed back to the operating system otherwise the system would slowly begin

consuming all memory on the system, leading to a crash. The chosen way to test for memory leaks was to use the tool Valgrind [38]. Valgrind provides an extensive range of memory and thread checking tools along with profiling tools. For this test, the memory leak test was run with the `--leak-check=full` option enabled to provide a thorough check. In order to provide a fair and realistic test, a number of hosts on the network first advertised, and then several firewall transactions took place from different nodes.

The test was carried out by connecting all six nodes to the system, followed by two firewall rule transactions.

This chapter has provided an overview of how the system will be tested. The next chapter will detail the results of these testing methods, along with a general evaluation of the success of the project

Chapter 6

Evaluation

This chapter will detail the successful aspects of the project in addition to identifying where the project could have been improved. Additionally, the results from the previously detailed tests will be used to evaluate the effectiveness of the provided system and implementation.

6.1 Project Results

6.1.1 Decentralised Distributed System Model

The first output of this project is a new model for creating decentralised distributed systems. As discussed at the start of this publication, distributed systems are traditionally centralised systems, and those that are decentralised usually require a large amount of overhead to ensure consistency across the system. This project has produced a much more lightweight protocol, called the ACR (Advertisement, Consensus, Rule) protocol which enables quick transactions to be propagated throughout any network, with no requirements for specific hardware or network topology. The ACR protocol was designed as a high level abstraction for the underlying transaction stages, and acts as a very high level interface for additional implementations. This allows a huge amount of variation, for example any combination of advertisement, consensus and rule transaction methods can be used, which can then be further customised, for example by using different hash algorithms, or adding additional fields to the blocks.

The primary advantage of this model is speed, simplicity and redundancy, however encasing an entire protocol into just three message types vastly reduces the amount of control available, which means that this model currently does not have a method for nodes to catch up to a chain, instead it must simply start a new chain and is unable not obtain any previous knowledge within the system. There are a number of additional issues with the proposed implementation, the first of which is the propagation mechanism. Currently, each time a node receives a message from any other node, if that node is not the message's destination, it is forwarded to all known hosts. This creates an extremely high number of redundant packets on the network, which is exponential with regards to the number of hops permitted for each message. When the simulation in section 5.3 was analysed, over 16,000 messages were sent across the network for only five advertisements and two consensus-rule transactions. For smaller networks, this can be avoided by reducing the hop count to the minimum required to reach all nodes, however the problem only gets worse when applying this protocol to larger networks.

Security is another major flaw within the proposed implementation. While the ACR protocol does permit security, it is not discussed in this publication due to the difficulty of reliably securing transactions over UDP. To implement security would require some form of key negotiation at the advertisement stage, however if a malicious node has gained access to the network, then it can easily intercept the negotiation, allowing it to decrypt any message involving that node. This would also introduce a large overhead in regards to computation, as each node would

be required to store the keys of every other node, and as seen by the inefficiency of the message system, each node would be required to carry out a large volume of decryption and encryption.

6.1.2 Firewall Blockchains

The second output of this project is a new application of blockchain to firewall rules. Blockchain is a prominent technology used within decentralised cryptography for validating cryptocurrency transactions, but is yet to see many uses in cybersecurity. This project has explored one of many potential uses, by enabling nodes to share information about attacks in the form of firewall rules, which allow other nodes to quickly configure their security to protect against the coming attacks. To accommodate these new transactions, a new form of consensus mechanism had to be created in order to provide near real-time decisions across a vast network.

The proposed consensus mechanism, although it is some form of consensus, is not necessarily a secure nor trustworthy method. Traditionally, in blockchain, transactions are not validated in real-time, and instead the hash is solved via a process called mining. Only once this hash has been mined, a process achieved by a number of high power Graphics Processing Unit (GPU)s, is the transaction deemed as valid and appended onto the blockchain. This consensus mechanism ensures the legitimacy of the block, whereas the proposed consensus method does not necessarily validate this property. Instead, the current consensus mechanism simply determines whether the sending node is up-to-date with the reset of the system, whereby if it's last block is the same as the majority of other nodes, the new block is deemed as valid, as the hashes will correctly match. This technique does mitigate the risks of a malicious node joining the network and immediately sending malicious firewall rules, however such a mechanism can easily be defeated by intercepting a genuine rule message, and using its hash to then validate its own block.

Similarly to the model mentioned above, the firewall blockchain is also designed with flexibility. The examples shown in this project demonstrate how the chain is used to store traditional stateless rules, which either allow or block transactions based entirely off the IP information of each packet. These rules could be easily modified to contain rules which are evaluated depending on the contents of each packet, or other control mechanisms such as a Access Control List (ACL), which can be integrated into organisational security mechanisms.

6.1.3 Proof of Concept

The final output of this project is a proof of concept implementation of the system. This proof of concept is provided in the form of a C program which is written in such a way to be cross-compatible with the majority of UNIX and Windows operating systems. Measured on the primary development computer, this C program which acts as a stub in the distributed system is 22.9 KB in size, which makes the system extremely portable (you could store 62 copies on a single floppy disk). This proof of concept is designed to run in the background (as a daemon) and receives communications via IPC and over the network, and works extremely effectively as seem by the tests carried out. For sending messages to the system via IPC, a small driver program was written in C, which was used to construct messages and then send then to the stub in the relevant format.

In addition to communicating via IPC, as the chosen communication channel is message queues and named pipes, interfaces can easily be built in other programming languages to relay messages to the stub. Whilst it was established that this was possible early in the project, it wasn't until the end that a driver program was implemented in a different language. It was then decided that a Java wrapper would be created as Java is a widely used language, and also presents an additional challenge of sending messages out of the JVM and through the native IPC mechanism. Whilst developing this it was discovered that nobody had yet created a library for passing messages from Java to a POSIX message queue, and hence a secondary project was created, called JPMQ [39]. Although the details will not be discussed here, JPMQ provides

a method of interfacing with POSIX message queues. The development of this Java based program reinforces the advantages previously mentioned of developing the program in C, as it allows nearly all higher languages to communicate via relevant wrappers and APIs.

6.2 Evaluating the Development Process

In regards to software engineering, this project has created a working, reliable piece of software which is cross compatible and has been extensively tested. With regards to the development methodology used, the waterfall method seemed like the most appropriate method, as the project was developed by a single person, and there would be no input from clients so there would be no substantial changes along the way. Using a method such as agile would vastly overcomplicate the development process, as if a particular element of the project was delayed, the rest of development would need to be restructured which would add unnecessary overhead to the project administration. This method worked very well for this project as at the start of the project there were a lot of new technologies and a number of hurdles to overcome, and using the waterfall method allowed each technology to be implemented one step at a time without concern for the other aspects of the project.

The original designs for the system were intentionally vague due to the project using unfamiliar technologies. Unfortunately this did lead to a number of oversights throughout the project, for instance although the rough outline of the protocol was designed upfront, there were a number of issues such as how routing would be carried out, and how exactly consensus would be negotiated. Ultimately however, the original timetable permitted sufficient time for researching and working on these additional issues, but they could have been avoided with more careful designing.

With regards to implementation, this project used a loose implementation of the Test Driven Development (TDD) methodology. The term 'loose' is used here as it was implemented alongside the previously mentioned waterfall method. TDD was used to help keep the project on track, as if tests were not created up front, there was a large risk of feature creep, and the addition of unnecessary features into a system which is designed to be reliable is not a good idea. Unlike traditional TDD, this project did not define all tests upfront, but instead tests were created prior to each stage / component of the system. The reason for this was due to the unfamiliarity with the libraries used, and so it seemed logical to create the tests once a strong understanding was grasped of that particular technology. This system worked well and ensured that tests were suitable and tailored to both the problem and the technologies used.

6.3 Test Results

The methodology for testing was described in detail in section 5, however this section provides a more detailed analysis of those results.

The local tests all passed for the tested operating systems and architectures. The local tests confirmed that all of the tested hardware and operating systems supported the software, and that the software could correctly interact with other components on the OS. It would be bewildering to show all test outputs here, so the tests for one particular operating system are provided below.

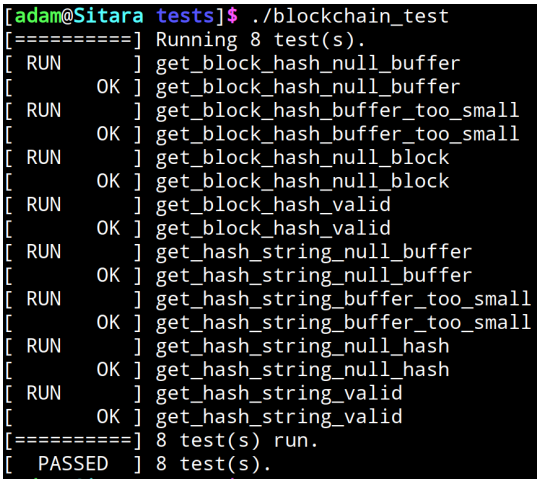

Figure 6.1: The output of the blockchain unit tests

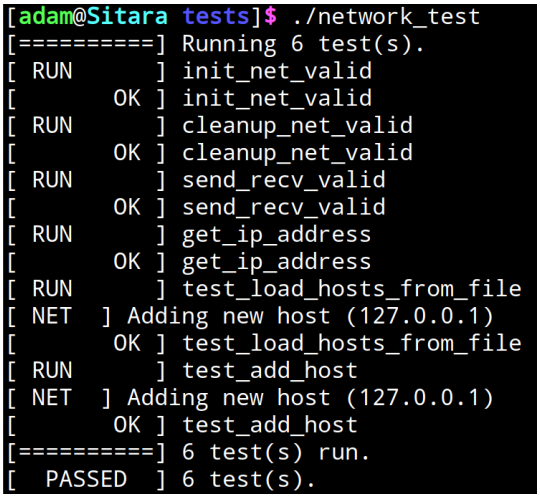

Figure 6.2: The output of the network unit tests

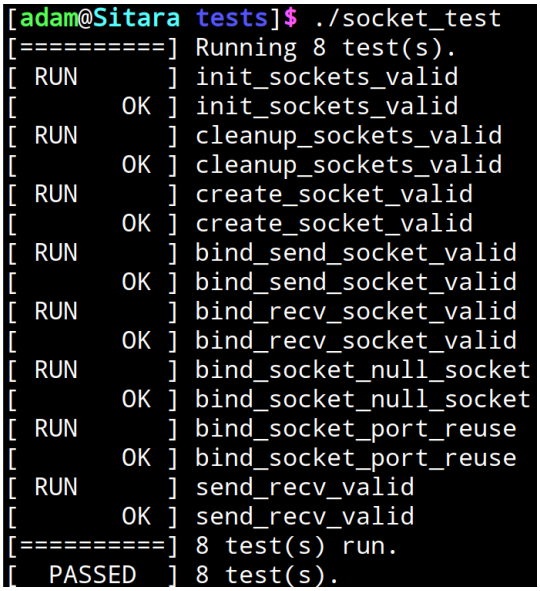

Figure 6.3: The output of the socket unit tests

Unfortunately the number of tests that could be carried out locally was constricted by the networked nature of the system. Although these tests are limited, they did provide valuable information and guidance when implementing these functions, however their effectiveness began to decline as the system became more complex. As can be seen from the tables in 5.1, there were a large number of additional functions implemented but not tested, and this is where using both waterfall and TDD began to lose effectiveness. Before starting development on a particular component, the tests were written up, however it became increasingly difficult to maintain both the tests and the functional code, which resulted in TDD becoming a lower priority and functions were instead tested as they were implemented, which is why there is a lack of tests in the unit tests.

Additionally, as mentioned previously, the software underwent testing for memory leaks to ensure it was capable of operating without hogging system resources. The image below shows the output of Valgrind, indicating that there were a total of 43 heap allocations throughout the lifetime of the program, and that there were no allocated bytes still in use at the point of program termination. Additionally, Valgrind provides a convenient summary stating that “All heap blocks were freed - no leaks are possible”.

```
[adam@Sitara src]$ valgrind --leak-check=full ./dfw
==6479== Memcheck, a memory error detector
==6479== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6479== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==6479== Command: ./dfw
==6479==
**************************************************************
*         Decentralised Distributed Firewall Framework       *
*                        by Adam Bruce                       *
**************************************************************

[ IPC  ] Initialised IPC
[ NET  ] Initialised network stack
[ NET  ] No hostfile found
[ BLOC ] No chain file found
[ INFO ] Initialised receiving thread
[ INFO ] Ready
[ NET  ] Adding new host (192.168.2.2)
[ NET  ] Adding new host (192.168.2.128)
[ NET  ] Adding new host (192.168.2.83)
[ NET  ] Adding new host (192.168.2.126)
[ IPC  ] Received IPC Message: New Firewall Rule
[ CONS ] Sent consensus message to 4 known host(s)
[ CONS ] Consensus achieved (4/2 required hosts)
[ RULE ] Sent new rule message to 4 known hosts(s)
[ BLOC ] Added new block with hash c7db4d454...38a6f6d3ee
[ IPC  ] Received IPC Message: New Firewall Rule
[ CONS ] Sent consensus message to 4 known host(s)
[ CONS ] Consensus achieved (4/2 required hosts)
[ RULE ] Sent new rule message to 4 known hosts(s)
[ BLOC ] Added new block with hash 869d7e5f6...42c0727d22
[ IPC  ] Received IPC Message: Shutting down
[ INFO ] Waiting for receiving thread to terminate
==6479==
==6479== HEAP SUMMARY:
==6479==     in use at exit: 0 bytes in 0 blocks
==6479==   total heap usage: 43 allocs, 43 frees, 49,912 bytes allocated
==6479==
==6479== All heap blocks were freed -- no leaks are possible
==6479==
==6479== For lists of detected and suppressed errors, rerun with: -s
==6479== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 6.4: The output of Valgrind

Valgrind is an effective tool and is well recognised for its accuracy and detailed output when a leak is found. It can therefore be trusted that the output pictured above is accurate, and that the software does not contain any memory leaks.

Now the system has been evaluated and the results of the testing process have been collected it is clear that the system operates correctly and a successful and functional proof of concept has been created.

Chapter 7

Conclusion

This section will conclude the project by identifying whether the original aims and objectives were met, and suggesting future work to improve the proposed system.

7.1 Original Aims and Objectives

This project satisfied the original aim, creating a security model which comprises of protocols and data structures that utilises blockchain in order to create a decentralised security mechanism that can be operated over a distributed system. However, it has not succeeded in meeting all of the proposed objectives.

Objective 1 was partially met as initial research did provide background information with regards to existing distributed security mechanisms and their uses, however this literature has not drawn any evaluation in regard to their effectiveness. Unfortunately this objective was not met due a lack of publicly accessible information on these systems and due to time constraints.

Objective 2 was met as initial research and existing knowledge allowed a number of possible methods of both connection establishment and synchronisation techniques to be investigated. As detailed it was decided that the best method of establishing connections was actually not to use any connection oriented protocols at all, and UDP which is a connectionless protocol was identified as the best protocol for this project. As for synchronisation, this aspect became largely irrelevant as the implementation of blockchain provided a mechanism for keeping all nodes up-to-date with the correct information.

Objective 3 was met as a structure for encapsulating firewall transactions within blockchain was designed and successfully implemented into the system.

Objective 4 was partially met as additional research indicated that a number of fault tolerance mechanisms would not be suitable for this project. As the project runs on an underlying operating system, the amount of control the software has is not sufficient for traditional fault tolerance mechanisms. Instead, the software utilises a fail early design, such that should a situation arise that is unrecoverable, the system will simply ignore the malicious data, or if needed, terminate the program. Additionally, backups are created whenever a transaction takes place enabling the system to load its previous state should it crash or terminate.

Objective 5 was not met as despite thorough research, no existing decentralised security mechanisms were found, which implies that this project could be the first publicly available implementation of such a system.

7.2 Personal Development

Throughout this project I have developed a number of skills and vastly expanded my knowledge of certain areas of computing. One of the most important skills I have developed is my research ability as before undertaking this project I had never carried out such thorough research nor analysed the literature I was reading in such depth. These skills will allow be to continue to both learn and scrutinise research in the future, and should I be the author of any research I feel much more confident in writing clear documents, with reputable references.

Developing this project in a low level language such as C has taught me a number of practises for developing reliable and correct software. At the beginning of the development, my enjoyment of writing C quickly became a chore as I was bombarded with segmentation faults and core dumps and not understanding why the software was behaving in particular ways. After hours spent debugging, I slowly learnt how making assumptions about how a computer may act is a crucial mistake, and quickly began ensuring all data was formatted to the exact specifications of the C language. After correcting my mistakes, I could now see how my code had let to unpredictable behaviour and now know how to avoid it.

In regards to technical knowledge, this project has required me to gain a thorough understanding of operating systems, distributed systems and blockchain. I had some prior knowledge of operating systems however I was required to learn about entirely new aspects including IPC, sockets and networking in general. Additionally, I learnt a number of lessons with regard to cross compatibility and learnt a number of quirks in regards to different operating systems and how they behave. I also only had a limited amount of knowledge regarding distributed systems, and throughout this project I have gained a thorough understanding of how they use stubs to collaboratively establish middleware, which permits transactions to exchange information across a network.

Finally, I developed my understanding of blockchain, and unlike before this project, blockchain is no longer just a buzzword. I have learnt that blockchain is not only useful in cryptocurrency, but has a number of additional uses including those discussed in this literature.

7.3 Future Work

There are a number of changes and most importantly improvements that can be made to this project. There are a wide range of minor changes which could include the structure of the blocks in the chain, or simply the hashing algorithms used, however there are also some less trivial changes which are listed below:

- **Security** - Currently there is no security regarding encapsulating the protocol, and furthermore, the protocol itself does contain known vulnerabilities which are not desirable. Ideal future work would include securing the packets sent across the network, which could be achieved by using some kind of shared key across the entire network, however this would only prevent outside devices from snooping, and would still allow bad actors within the network to snoop. Alternatively, UDP could be replaced with TCP which would allow common security protocols such as Transport Layer Security (TLS) to be utilised, however this would add a large amount of overhead, hence reducing the speed at which messages can propagate.

- **Message Efficiency** - Currently the protocol is hugely inefficient in regards to the number of messages transmitted across the network. The use of hops and the fact that each host relays any message it receives to all of its known hosts creates an exponential amount of messages on the network, which means that should the number of hosts reach a high enough level, the network essentially creates a Distributed Denial-of-Service (DDOS) attack, potentially causing devices to spend too much time processing messages instead

of processing genuine traffic. The efficiency of messages could be vastly improved by integrating a more effective mechanism for providing redundancy.

- **Programming Language** - The implementation provided in this project is written in C, however there are a number of alternatives that could offer much more resilience and tolerance to error. A number of these were considered but dismissed due to their immaturity. Whilst this is still true, upon reflection I have learnt that benefits such as Go's efficiency and Rust's memory safety could provide a much more safety and resiliency. Furthermore, these languages are both compiled which means they would be no less efficient that the C implementation, and could be much easily developed for, with less quirks than C. The only current barrier is that these languages do not currently support all of the operating system interfaces that C currently does, but an implementation could easily use different IPC methods or networking interfaces to ensure compatibility.

Glossary

blockchain A growing list of records, called blocks, that are linked using cryptography. 1, 3, 6, 7, 10, 11, 17, 24, 25, 32, 37, 38

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, 24, 27, 29, 38

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. 1, 3, 6, 8, 9, 38

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. 6–9, 14, 17–19, 22–24, 26, 29, 32, 38

Acronyms

ACL Access Control List. 32

API Application Programming Interface. 24, 33

APT Advanced Persistent Threat. 5

DMZ Demilitarised Zone. 3, 12

GPU Graphics Processing Unit. 32

IP Internet Protocol. 15, 17, 26, 32

IPC Inter-Process Communication. 4, 14, 16–19, 21–24, 32, 38, 39

NCSC National Cyber Security Center. 5

NIC Network Interface Card. 15, 22, 23

OS Operating System. 6, 9, 13–15, 21, 23, 33

POSIX Portable Operating System Interface. 21–24, 33

RPC Remote Procedure Call. 3, 9, 10

TCP Transmission Control Protocol. 12, 13, 38

TDD Test Driven Development. 33, 35

UDP User Datagram Protocol. 12, 13, 31, 37, 38

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://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] J. M. Hart, *Windows system programming*. Pearson Education, 2010.

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

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

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

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

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

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

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

[37] “cmocka - uni testing framework for c.” https://cmocka.org/, 2020. Accessed: 12/04/2020.

[38] “Valgrind home.” https://www.valgrind.org/, 2021. Accessed: 14/04/2020.

[39] A. D. Bruce, “Jpmq - POSIX Message Queues for Java.” https://adambruce.net/jpmq, 2021. Accessed: 08/04/2021.

Appendix A

Framework Source Code

Listing A.1: main.c

```
/**
 * @file main.c
 * @brief Entry point for the application.
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "firewall.h"
#include "net.h"
#include "ipc.h"
#include "blockchain.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifdef _WIN32
#define HAVE_STRUCT_TIMESPEC /* Prevent pthread from redefining timespec */
#undef INET_ADDRSTRLEN
#define INET_ADDRSTRLEN 16
#else
#include <unistd.h>
#endif

#include <pthread.h>

/* Flags */
static int enabled_flag = 1;
static int shutdown_flag = 0;

/**
 * @brief Receiving thread.
 *
 * This function is automatically run on the second thread, receiving and
 * processing data from the network.
 */
void *recv_thread_func(void *data)
{
  char buffer[sizeof(RuleMessage)];
  while(!shutdown_flag)
    {
      memset(buffer, 0, sizeof(RuleMessage));
      if(enabled_flag)
  {
    poll_message(buffer, sizeof(RuleMessage));
  }
#ifdef _WIN32
      Sleep(10);
#else
      usleep(10 * 1000);
#endif
    }
```

```
53   return NULL;
54 }
55
56 int main(int argc, char** argv)
57 {
58   IPCMessage ipc_msg;
59   AdvertisementMessage adv_msg;
60   pthread_t recv_thread;
61   char local_addr[INET_ADDRSTRLEN];
62
63   /* Banner */
64   printf("*****************************************************************\n");
65   printf("*         Decentralised Distributed Firewall Framework          *\n");
66   printf("*                          by Adam Bruce                        *\n");
67   printf("*****************************************************************\n");
68   printf("\n");
69
70   /* Initialise IPC */
71   if(init_ipc_server())
72     {
73       perror("[ IPC  ] Failed to initialise IPC");
74       return 1;
75     }
76   printf("[ IPC  ] Initialised IPC\n");
77
78   /* Initialise the network stack */
79   if(init_net())
80     {
81       cleanup_ipc();
82       perror("[ ERR  ] Failed to initialise network stack");
83       return 1;
84     }
85   printf("[ NET  ] Initialised network stack\n");
86   load_hosts_from_file("hosts.txt");
87
88   /* Load any stored blocks */
89   load_blocks_from_file("chain.txt");
90
91   /* Create the receiving thread */
92   if(pthread_create(&recv_thread, NULL, recv_thread_func, NULL))
93     {
94       perror("[ ERR  ] Failed to initialise receiving thread");
95       cleanup_net();
96       cleanup_ipc();
97       return 1;
98     }
99   printf("[ INFO ] Initialised receiving thread\n");
100
101  /* Send advertisement when joining the network */
102  if(get_host_count() > 0)
103    {
104      adv_msg.type = ADVERTISEMENT;
105      adv_msg.hops = 0;
106      adv_msg.advertisement_type = BROADCAST;
107      get_local_address(local_addr);
108      strncpy(adv_msg.source_addr, local_addr, INET_ADDRSTRLEN);
109      send_to_all_advertisement_message(&adv_msg);
110      printf("[ ADV  ] Sent advertisement to %d known host(s)\n",
111       get_host_count());
112    }
113
114 #ifdef _WIN32
115   connect_ipc();
116 #endif
117
118   printf("[ INFO ] Ready\n");
119
120   /* Process IPC commands */
121   while(!shutdown_flag)
122     {
123       memset(&ipc_msg, 0, sizeof(IPCMessage));
124       recv_ipc_message(&ipc_msg);
125        switch(ipc_msg.message_type)
```

```
126   {
127   case I_SHUTDOWN:
128     printf("[ IPC  ] Received IPC Message: Shutting down\n");
129     shutdown_flag = 1;
130     break;
131   case I_ENABLE:
132     printf("[ IPC  ] Received IPC Message: Enabling Transactions\n");
133     enabled_flag = 1;
134     break;
135   case I_DISABLE:
136     printf("[ IPC  ] Received IPC Message: Disabling Transactions\n");
137     enabled_flag = 0;
138     break;
139   case I_RULE:
140     printf("[ IPC  ] Received IPC Message: New Firewall Rule\n");
141     send_new_rule(&ipc_msg.rule);
142       break;
143   case O_RULE:
144     break;
145   default:
146     printf("[ ERR  ] Recieved Unknown IPC Message Type\n");
147   }
148     }
149
150   /* Cleanup and terminate */
151   printf("[ INFO ] Waiting for receiving thread to terminate\n");
152   pthread_join(recv_thread, NULL);
153   cleanup_net();
154   cleanup_ipc();
155   free_chain();
156   return 0;
157 }
```

Listing A.2: blockchain.h

```
1  /**
2   * @file blockchain.h
3   * @brief Functions for creating and validating blockchains.
4   * @author Adam Bruce
5   * @date 22 Mar 2021
6   */
7
8  #ifndef BLOCKCHAIN_H
9  #define BLOCKCHAIN_H
10
11 #include "firewall.h"
12
13 #include <openssl/sha.h>
14
15 #ifdef _WIN32
16 #include <ws2tcpip.h>
17 #else
18 #include <arpa/inet.h>
19 #endif
20
21 /**
22  * @brief The length of SHA256 string representations.
23  */
24 #define SHA256_STRING_LENGTH 64
25
26 /**
27  * A block containing information for a firewall transaction.
28  */
29 struct FirewallBlock
30 {
31   unsigned char last_hash[SHA256_DIGEST_LENGTH]; /**< The hash of the
32                 previous block            */
33   char author[INET_ADDRSTRLEN];                  /**< The address of the block
34                 author                    */
35   FirewallRule rule;                             /**< The firewall rule
36                 associated with the block */
37   struct FirewallBlock *next;
38 };
```

```
39 typedef struct FirewallBlock FirewallBlock;
40
41 /**
42  * The firewall block used to store this host's proposed new rule.
43  */
44 static FirewallBlock block;
45
46 /**
47  * The blockchain of current firewall rules.
48  */
49 static FirewallBlock *chain;
50
51 /**
52  * @brief Calculates the SHA256 hash of a block.
53  *
54  * Calculates the SHA256 hash of a block, storing the digest in the given
55  * buffer. This buffer should have a size of SHA256_DIGEST_LENGTH.
56  * @param buffer the buffer to store the digest in.
57  * @param block a pointer to the block to hash.
58  * @param buffer_size the size of the buffer to store the hash in.
59  * @return whether the hash has been calculated successfully. If any parameters
60  * are invalid, the return value will be 1, otherwise the return value will be
61  * 0.
62  */
63 int get_block_hash(unsigned char *buffer, FirewallBlock *block,
64        int buffer_size);
65
66 /**
67  * @brief Formats a SHA256 digest into human-readable string.
68  *
69  * Formats a SHA256 digest into a human-readable string, storing the result
70  * into the given buffer. This buffer should have a size of
71  * SHA256_STRING_LENGTH.
72  * @param buffer the buffer to store the string in.
73  * @param hash the hash digest to format into a string.
74  * @param buffer_size the size of the buffer to store the string in.
75  * @return whether the string has been formatted succesfully. If any parameters
76  * are invalid, the return value will be 1, otherwise the return value will be
77  * 0.
78  */
79 int get_hash_string(char *buffer, unsigned char *hash, int buffer_size);
80
81 int get_hash_from_string(unsigned char *buffer, char *hash_string,
82        int buffer_size);
83
84 /**
85  * @brief Adds a new firewall block onto the chain.
86  *
87  * Appends the new firewall block to the linked list of firewall block.
88  * @param block the new block to add to the chain.
89  * @return whether the block has been added to the chain. If an the block is
90  * is null or the block's memory could not be allocated, the return value
91  * will be 1, otherwise the return value will be 0.
92  */
93 int add_block_to_chain(FirewallBlock *block);
94
95 /**
96  * @brief Rotates the pending firewall rules.
97  *
98  * Rotates this host's list of pending firewall rules, such that the oldest
99  * rule is removed from the list, allowing a new block to be added.
100  * @return whether the list was rotated. If an error has occurred, the return
101  * value will be 1, otherwise the return value will be 0.
102  */
103 int rotate_pending_rules(void);
104
105 /**
106  * @brief Adds a new rule to the list of pending rules.
107  *
108  * Appends a new rule to the list of pending rules, this involves rotating the
109  * list, and adding the new rule's author.
110  * @param addr the author of the new pending rule.
111  * @return whether the rule was added. If an error has occurred, the return
```

```
112  * value will be 1, otherwise the return value will be 0.
113  */
114 int add_pending_rule(char *addr);
115
116 /**
117  * @brief Checks if the given address has a pending rule.
118  *
119  * Searches the pending rule list for the given address. If the address is
120  * found then the host has a pending rule.
121  * @param addr the author to check for pending rules.
122  * @return whether any pending rules for the author were found. If a pending
123  * rule is found, the return value will be 1, otherwise the return value will
124  * be 0.
125  */
126 int is_pending(char *addr);
127
128 /**
129  * @brief Removes a pending rule from the list.
130  *
131  * Searches for a pending rule with the given address. If a matching rule is
132  * found, the rule is removed.
133  * @param addr the address to remove.
134  * @return whether the pending rule was removed. If an error has occurred, the
135  * return value will be 1, otherwise the return value will be 0.
136  */
137 int remove_pending_rule(char *addr);
138
139 /**
140  * @brief Returns a pointer to the last firewall block.
141  *
142  * Returns a pointer to the last firewall block in the chain.
143  * @param block pointer to point to the last block.
144  * @return whether the last block was successfully found. If an error has
145  * occurred, the return value will be 1, otherwise the return value will be 0.
146  */
147 int get_last_block(FirewallBlock *block);
148
149 /**
150  * @brief Returns the hash of the last firewall block in the chain.
151  *
152  * Gets the SHA256 hash of the last firewall block in the chain. If the chain
153  * is empty, the buffer will be empty.
154  * @param buffer the buffer that the hash value will be copied into. This
155  * buffer should be at least SHA256_DIGEST_LENGTH bytes in size.
156  * @return whether the hash value was copied successfully. If an error has
157  * occurred, the return value will be 1, otherwise the return value will be 0.
158  */
159 int get_last_hash(unsigned char *buffer);
160
161 /**
162  * @brief Loads a list of firewall blocks from a file.
163  *
164  * Loads a list of firewalls blocks from the given file and constructs the
165  * local blockchain.
166  * @param fname the name of the file containing the chain.
167  * @return whether the chain was sucecssfully loaded. If an error has occurred,
168  * the return value will be 1, otherwise the return value will be 0.
169  */
170 int load_blocks_from_file(const char *fname);
171
172 /**
173  * @brief Saves the current loaded blockchain into a file.
174  *
175  * Saves all blocks currently loaded into the blockchain.
176  * @param fname the name of the file to save the blockchain.
177  * @return whether the blockchain was successfully saved. If an error has
178  * occurred, the return value will be 1, otherwise the return value will be 0.
179  */
180 int save_blocks_to_file(const char *fname);
181
182 /**
183  * @brief Frees the currently loaded blockchain.
184  *
```

```
 * Frees the memory currently allocated to blocks on the chain.
 * @return whether the chain was successfully freed. If an error has occurred,
 * the return value will be 1, otherwise the return value will be 0.
 */
int free_chain(void);

#endif
```

Listing A.3: blockchain.c

```
/**
 * @file blockchain.c
 * @brief Functions for creating and validating blockchains.
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "blockchain.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <openssl/sha.h>

#ifdef _WIN32
#include <ws2tcpip.h>
#undef INET_ADDRSTRLEN
#define INET_ADDRSTRLEN 16
#else
#include <arpa/inet.h>
#endif

#define PENDING_RULES_BUF_LEN 10
static char pending_rules[PENDING_RULES_BUF_LEN][INET_ADDRSTRLEN];

int get_block_hash(unsigned char *buffer, FirewallBlock *block,
       int buffer_size)
{
  SHA256_CTX sha256;
  unsigned char data_to_hash[(INET_ADDRSTRLEN * 3) + 8];

  if(buffer_size < SHA256_DIGEST_LENGTH || !buffer || !block)
    {
      return 1;
    }

  memcpy(data_to_hash, block->author, INET_ADDRSTRLEN);
  memcpy(data_to_hash + INET_ADDRSTRLEN, block->rule.source_addr,
   INET_ADDRSTRLEN);
  memcpy(data_to_hash + INET_ADDRSTRLEN * 2, block->rule.dest_addr,
   INET_ADDRSTRLEN);
  memcpy(data_to_hash + INET_ADDRSTRLEN * 3,
   (void*)&block->rule.source_port, 2);
  memcpy(data_to_hash + (INET_ADDRSTRLEN * 3) + 2,
   (void*)&block->rule.dest_port, 2);
  memcpy(data_to_hash + (INET_ADDRSTRLEN * 3) + 4,
   (void*)&block->rule.action, 4);

  SHA256_Init(&sha256);
  SHA256_Update(&sha256, data_to_hash, (INET_ADDRSTRLEN * 3) + 4 + 4);
  SHA256_Final(buffer, &sha256);

  return 0;
}

int get_hash_string(char *buffer, unsigned char *hash, int buffer_size)
{
  int i;

  if(buffer_size < SHA256_STRING_LENGTH + 1 || !buffer || !hash)
    {
      return 1;
```

```
64      }
65
66    for(i = 0; i < SHA256_DIGEST_LENGTH; i++)
67      {
68        sprintf(buffer + (i * 2), "%02x", hash[i]);
69      }
70    buffer[SHA256_STRING_LENGTH] = '\0';
71    return 0;
72  }
73
74  int get_hash_from_string(unsigned char *buffer, char *hash_string,
75          int buffer_size)
76  {
77    int i;
78    uint16_t hex_val;
79    char buf[3];
80
81    if(buffer_size < SHA256_DIGEST_LENGTH || !buffer || !hash_string)
82      {
83        return 1;
84      }
85
86    buf[2] = '\0';
87    for(i = 0; i < SHA256_DIGEST_LENGTH; i++)
88      {
89        memcpy(buf, hash_string + (i * 2), 2);
90        hex_val = strtol(buf, NULL, 16);
91        memcpy(buffer + i, &hex_val, 2);
92      }
93
94    return 0;
95  }
96
97  int add_block_to_chain(FirewallBlock *block)
98  {
99    FirewallBlock *fw_chain;
100   unsigned char hash[SHA256_DIGEST_LENGTH + 1];
101   char hash_string[SHA256_STRING_LENGTH + 1];
102
103   get_block_hash(hash, block, SHA256_DIGEST_LENGTH + 1);
104   get_hash_string(hash_string, hash, SHA256_STRING_LENGTH + 1);
105   hash_string[9] = '\0';
106
107   if(!chain)
108     {
109       chain = (FirewallBlock*)malloc(sizeof(FirewallBlock));
110       memset(chain, 0, sizeof(FirewallBlock));
111       memcpy(chain, block, sizeof(FirewallBlock));
112
113       printf("[ BLOC ] Added new block with hash %s...%s\n",
114        hash_string, hash_string + (SHA256_STRING_LENGTH - 10));
115       save_blocks_to_file("chain.txt");
116       return 0;
117     }
118
119   fw_chain = chain;
120   while(fw_chain && fw_chain->next)
121     {
122       fw_chain = fw_chain->next;
123     }
124
125   fw_chain->next = (FirewallBlock*)malloc(sizeof(FirewallBlock));
126   memset(fw_chain->next, 0, sizeof(FirewallBlock));
127   memcpy(fw_chain->next, block, sizeof(FirewallBlock));
128   printf("[ BLOC ] Added new block with hash %s...%s\n",
129    hash_string, hash_string + (SHA256_STRING_LENGTH - 10));
130
131   save_blocks_to_file("chain.txt");
132   return 0;
133
134 }
135
136 int rotate_pending_rules(void)
```

```
137 {
138   int index;
139 
140   for(index = 0; index < PENDING_RULES_BUF_LEN - 2; index++)
141     {
142       strncpy(pending_rules[index], pending_rules[index + 1], INET_ADDRSTRLEN);
143     }
144   memset(pending_rules[0], '\0', INET_ADDRSTRLEN);
145 
146   return 0;
147 }
148 
149 int add_pending_rule(char *addr)
150 {
151   rotate_pending_rules();
152   strncpy(pending_rules[0], addr, INET_ADDRSTRLEN);
153   return 0;
154 }
155 
156 int is_pending(char *addr)
157 {
158   int index;
159 
160   for(index = 0; index < PENDING_RULES_BUF_LEN; index++)
161     {
162       if(strncmp(pending_rules[index], addr, INET_ADDRSTRLEN) == 0)
163   {
164     return 1;
165   }
166     }
167   return 0;
168 }
169 
170 int remove_pending_rule(char *addr)
171 {
172   int index, match;
173 
174   match = 0;
175   for(index = 0; index < PENDING_RULES_BUF_LEN; index++)
176     {
177       if(strncmp(pending_rules[index], addr, INET_ADDRSTRLEN) == 0)
178   {
179     match = 1;
180     break;
181   }
182     }
183 
184   if(match)
185     {
186       for(; index > 0; index--)
187   {
188     strncpy(pending_rules[index], pending_rules[index - 1], INET_ADDRSTRLEN);
189   }
190       memset(pending_rules[0], '\0', INET_ADDRSTRLEN);
191     }
192 
193   return 0;
194 }
195 
196 int get_last_block(FirewallBlock *block)
197 {
198   FirewallBlock *fw_block;
199 
200   if(!chain)
201     {
202       block = NULL;
203       return 1;
204     }
205 
206   fw_block = chain;
207   while(fw_block && fw_block->next)
208     {
209       fw_block = fw_block->next;
```

```
210     }
211
212   block = fw_block;
213   return 0;
214 }
215
216 int get_last_hash(unsigned char *buffer)
217 {
218   FirewallBlock *fw_chain;
219
220   if(!chain)
221     {
222       memset(buffer, '\0', SHA256_DIGEST_LENGTH);
223       return 0;
224     }
225
226   fw_chain = chain;
227   while(fw_chain && fw_chain->next)
228     {
229       fw_chain = fw_chain->next;
230     }
231
232   get_block_hash(buffer, fw_chain, SHA256_DIGEST_LENGTH);
233   return 0;
234 }
235
236 int load_blocks_from_file(const char *fname)
237 {
238   FILE *file;
239   FirewallBlock block;
240   char buffer[256], temp_buf[6], *next_delim;
241   int c;
242   size_t pos;
243
244   file = fopen(fname, "r");
245   if(!file)
246     {
247       printf("[ BLOC ] No chain file found\n");
248       return 1;
249     }
250
251   pos = 0;
252   while((c = fgetc(file)) != EOF)
253     {
254       if((char)c == '\r')
255   {
256     continue;
257   }
258
259       if((char)c == '\n')
260   {
261     buffer[pos] = '\0';
262     memset(&block, 0, sizeof(FirewallBlock));
263
264     /* Hash */
265     get_hash_from_string(block.last_hash, buffer, SHA256_DIGEST_LENGTH);
266     next_delim = strchr(buffer, ',');
267
268     /* Author */
269     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
270     next_delim = strchr(buffer, ',');
271     memcpy(block.author, buffer, next_delim - buffer);
272
273     /* Source address */
274     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
275     next_delim = strchr(buffer, ',');
276     memcpy(block.rule.source_addr, buffer, next_delim - buffer);
277
278     /* Source port */
279     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
280     next_delim = strchr(buffer, ',');
281     memset(temp_buf, '\0', 6);
282     memcpy(temp_buf, buffer, next_delim - buffer);
```

```
283     block.rule.source_port = atoi(temp_buf);
284
285     /* Destination address */
286     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
287     next_delim = strchr(buffer, ',');
288     memcpy(block.rule.dest_addr, buffer, next_delim - buffer);
289
290     /* Destination port */
291     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
292     next_delim = strchr(buffer, ',');
293     memset(temp_buf, '\0', 6);
294     memcpy(temp_buf, buffer, next_delim - buffer);
295     block.rule.dest_port = atoi(temp_buf);
296
297     memcpy(buffer, next_delim + 1, (buffer + 265) - next_delim - 1);
298     if(strcmp("ALLOW", buffer) == 0)
299       {
300         block.rule.action = ALLOW;
301       }
302     else if(strcmp("DENY", buffer) == 0)
303       {
304         block.rule.action = DENY;
305       }
306     else if(strcmp("BYPASS", buffer) == 0)
307       {
308         block.rule.action = BYPASS;
309       }
310     else if(strcmp("FORCE_ALLOW", buffer) == 0)
311       {
312         block.rule.action = FORCE_ALLOW;
313       }
314     else
315       {
316         block.rule.action = LOG;
317       }
318
319     add_block_to_chain(&block);
320
321     memset(buffer, '\0', 256);
322     pos = 0;
323     continue;
324   }
325       buffer[pos++] = (char)c;
326     }
327
328   fclose(file);
329   return 0;
330 }
331
332 int save_blocks_to_file(const char *fname)
333 {
334   FILE *file;
335   FirewallBlock *block;
336   char hash_string[SHA256_STRING_LENGTH + 1];
337
338   file = fopen(fname, "w+");
339   if(!file)
340     {
341       printf("[ ERR  ] Could not create block file\n");
342       return 1;
343     }
344
345   block = chain;
346   if(block)
347     {
348       while(block && strlen(block->author) > 0)
349   {
350     memset(hash_string, '\0', SHA256_STRING_LENGTH + 1);
351     get_hash_string(hash_string, block->last_hash,
352         SHA256_STRING_LENGTH + 1);
353     fwrite(hash_string, SHA256_STRING_LENGTH, 1, file);
354     fputc(',', file);
355     fwrite(block->author, strlen(block->author), 1, file);
```

```
356     fputc(',', file);
357     fwrite(block->rule.source_addr, strlen(block->rule.source_addr), 1,
358      file);
359     fputc(',', file);
360     fprintf(file, "%hd,", block->rule.source_port);
361     fwrite(block->rule.dest_addr, strlen(block->rule.dest_addr), 1,
362      file);
363     fputc(',', file);
364     fprintf(file, "%hd,", block->rule.dest_port);
365 
366     switch(block->rule.action)
367       {
368       case ALLOW:
369         fputs("ALLOW", file);
370         break;
371       case DENY:
372         fputs("DENY", file);
373         break;
374       case BYPASS:
375         fputs("BYPASS", file);
376         break;
377       case FORCE_ALLOW:
378         fputs("FORCE_ALLOW", file);
379         break;
380       case LOG:
381         fputs("LOG", file);
382       }
383 
384     fputc('\n', file);
385     block = block->next;
386   }
387     }
388 
389   fclose(file);
390   return 0;
391 }
392 
393 int free_chain(void)
394 {
395   FirewallBlock *block, *temp;
396   block = chain;
397 
398   while(block)
399     {
400       temp = block;
401       block = block->next;
402       free(temp);
403     }
404 
405   return 0;
406 }
```

Listing A.4: firewall.h

```
1  /**
2   * @file firewall.h
3   * @brief High level functions for handling firewall interactions.
4   * @author Adam Bruce
5   * @date 22 Mar 2021
6   */
7 
8  #ifndef FIREWALL_H
9  #define FIREWALL_H
10 
11 #ifdef _WIN32
12 #include <ws2tcpip.h>
13 #include <stdint.h>
14 typedef uint16_t u_int16_t;
15 #else
16 #include <arpa/inet.h>
17 #endif
18 
19 /**
```

```
 * @brief All valid firewall rule actions.
 */
typedef enum
  {
    ALLOW,          /**< The connection should be allowed            */
    BYPASS,         /**< The connection should be bypassed           */
    DENY,           /**< The connection should be denied             */
    FORCE_ALLOW,    /**< The connection shoule be forcefully allowed */
    LOG             /**< The connection should be logged             */
  } FirewallAction;

/**
 * @brief The structure of a firewall rule.
 */
typedef struct
{
  char source_addr[INET_ADDRSTRLEN]; /**< The rule's source address      */
  uint16_t source_port;              /**< The rule's source port         */
  char dest_addr[INET_ADDRSTRLEN];   /**< The rule's destination address */
  uint16_t dest_port;                /**< The rule's destination port    */
  FirewallAction action;             /**< The rule's action              */
} FirewallRule;

/**
 * @brief The function called once a new firewall rule is available.
 *
 * This function is called once a firewall rule has been submitted by remote
 * host, and the network has given consenus to the new firewall rule.
 * @param rule the new firewall rule that was received.
 * @return whether the corresponding IPC message to the OS was sent
 * successfully. If an error has occurred, the return value will be 1,
 * otherwise the return value will be 0.
 */
int recv_new_rule(FirewallRule *rule);

/**
 * @brief The function used to send a new firewall rule.
 *
 * This function is called when a firewall rule is sent from the OS via IPC.
 * The function will first attempt to gain consensus within the network, and if
 * successfull, it will transmit the new rule to all known hosts.
 * @param rule the new firewall to send.
 * @return whether the firewall rule was sent. If an error has occurred, the
 * return value will be 1, otherwise the return value will be 0.
 */
int send_new_rule(FirewallRule *rule);

#endif
```

Listing A.5: firewall.c

```
/**
 * @file firewall.c
 * @brief High level functions for handling firewall interactions.
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "blockchain.h"
#include "firewall.h"
#include "ipc.h"
#include "net.h"

#include <string.h>
#include <stdio.h>

#ifdef _WIN32
#undef INET_ADDRSTRLEN
#define INET_ADDRSTRLEN 16
#else
#include <unistd.h>
#endif

```

```
23 #include <openssl/sha.h>
24
25 #define TIMEOUT 500
26
27 int recv_new_rule(FirewallRule *rule)
28 {
29   IPCMessage msg;
30   msg.message_type = O_RULE;
31   memcpy(&msg.rule, rule, sizeof(FirewallRule));
32   return send_ipc_message(&msg);
33 }
34
35 int send_new_rule(FirewallRule *rule)
36 {
37   ConsensusMessage consensus_msg;
38   RuleMessage rule_msg;
39   FirewallBlock block;
40
41   get_local_address(local_address);
42   consensus_msg.type = CONSENSUS;
43   consensus_msg.hops = 0;
44   consensus_msg.consensus_type = BROADCAST;
45   strncpy(consensus_msg.source_addr, local_address, INET_ADDRSTRLEN);
46   strncpy(consensus_msg.target_addr, local_address, INET_ADDRSTRLEN);
47   get_last_hash(consensus_msg.last_block_hash);
48   send_to_all_consensus_message(&consensus_msg);
49   printf("[ CONS ] Sent consensus message to %d known host(s)\n",
50    get_host_count());
51
52 #ifdef _WIN32
53   Sleep(TIMEOUT);
54 #else
55   usleep(TIMEOUT * 1000);
56 #endif
57
58   /* At least half known hosts have consensus */
59   if(get_acks() < (get_host_count() + 1) / 2)
60     {
61       printf("[ CONS ] Consensus not achieved (%d/%d required hosts)\n",
62        get_acks(), get_host_count());
63       reset_acks();
64       return 1;
65     }
66
67   printf("[ CONS ] Consensus achieved (%d/%d required hosts)\n", get_acks(),
68    (get_host_count() + 1) / 2);
69   reset_acks();
70
71   rule_msg.type = RULE;
72   rule_msg.hops = 0;
73   rule_msg.rule_type = BROADCAST;
74   strncpy(rule_msg.source_addr, local_address, INET_ADDRSTRLEN);
75   strncpy(rule_msg.target_addr, local_address, INET_ADDRSTRLEN);
76   memcpy(&rule_msg.rule, rule, sizeof(FirewallRule));
77   send_to_all_rule_message(&rule_msg);
78
79   printf("[ RULE ] Sent new rule message to %d known hosts(s)\n",
80    get_host_count());
81
82   memset(&block, 0, sizeof(FirewallBlock));
83   get_last_hash(block.last_hash);
84   strncpy(block.author, local_address, INET_ADDRSTRLEN);
85   memcpy(&block.rule, rule, sizeof(FirewallRule));
86   add_block_to_chain(&block);
87
88   return 0;
89 }
```

Listing A.6: ipc.h

```
1 /**
2  * @file ipc.h
3  * @brief Inter-process Communication interface.
```

```
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "firewall.h"

#ifndef IPC_H
#define IPC_H

#ifdef _WIN32
#include <ws2tcpip.h>
#else
#include <fcntl.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#endif

/**
 * @brief All valid IPC message types.
 */
typedef enum
  {
    I_RULE,         /**< New rule                       */
    I_ENABLE,       /**< Enable network communication   */
    I_DISABLE,      /**< Disable network communication  */
    I_SHUTDOWN,     /**< Shutdown the framework         */
    O_RULE          /**< New rule from another node     */
  } IPCMessageType;

/**
 * @brief The structure of a IPC message.
 */
typedef struct
{
  IPCMessageType message_type; /**< The IPC message type */
  FirewallRule rule;           /**< The firewall rule (if type is I_RULE) */
} IPCMessage;

/**
 * @brief Initialise the IPC in server mode.
 *
 * Initialises the underlying IPC mechanism, and creates a new connection. If
 * on *nix this is achieved using the POSIX message queue, or Named Pipes if
 * on windows.
 * @return whether the IPC was successfully initialised, and a connection
 * esatblished. If an error has occurred, the return value will be 1, otherwise
 * the return value will be 0.
 */
int init_ipc_server(void);

#ifdef _WIN32
/**
 * @brief Connects to the relevant Named Pipe (Windows Only).
 *
 * Establishes a connection to the previously created Named Pipe on the Windows
 * operating system.
 * @return whether the connection to the Named Pipe was succesful. If an error
 * has occurred, the return value will be 1, otherwise the return value will be
 * 0.
 */
int connect_ipc(void);
#endif

/**
 * @brief Initialise the IPC in client mode.
 *
 * Connects to a previously established IPC server.
 * @return whether the connection was succesfully established. If an error has
 * occurred the return value will be 1, otherwise the return value will be 0.
 */
int init_ipc_client(void);

/**
```

```
 * @brief Cleans up the IPC session.
 *
 * Terminates the connection to the IPC session, and tears down the underlying
 * session.
 * @return whether the connection was successfully terminated. If an error has
 * occurred, the return value will be 1, otherwise the return value will be 0.
 */
int cleanup_ipc(void);

/**
 * @brief Send and IPC message to a client application.
 *
 * Sends an IPC message to a client connected via IPC.
 * @param message the message to send.
 * @return whether the message was sent successfully. If an error has occurred,
 * the return value will be 1, otherwise the return value will be 0.
 */
int send_ipc_message(IPCMessage *message);

/**
 * @brief Retrieves an IPC message.
 *
 * Checks for an IPC message waiting in the queue. If a message is found, it is
 * copied into the message parameter.
 * @param message the message that a waiting message will be copied into.
 * @return whether an IPC message has been copied from the queue. If an error
 * has occurred, the return value will be 1, otherwise the return value will be
 * 0.
 */
int recv_ipc_message(IPCMessage *message);

#endif
```

Listing A.7: ipc.c

```
/**
 * @file ipc.c
 * @brief Inter-process Communication interface
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "ipc.h"

#ifdef _WIN32
#define WIN32_LEAN_AND_MEAN /* We don't need all Windows headers */
#include <windows.h>
#include <ws2tcpip.h>
#else
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#endif

#ifdef _WIN32
static HANDLE queue;
#else
static mqd_t queue;
#endif

#ifdef _WIN32
int init_ipc_server(void)
{
  HANDLE mqueue;

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

  if(!mqueue || mqueue == INVALID_HANDLE_VALUE)
    {
      return 1;
```

```
39      }
40
41    queue = mqueue;
42
43    return 0;
44  }
45
46  int connect_ipc(void)
47  {
48    if(!ConnectNamedPipe(queue, NULL))
49      {
50        CloseHandle(queue);
51        return 1;
52      }
53    return 0;
54  }
55
56  int init_ipc_client(void)
57  {
58    HANDLE mqueue;
59
60    mqueue = CreateFile(TEXT("\\\\.\\pipe\\dfw"), PIPE_ACCESS_DUPLEX,
61            PIPE_TYPE_MESSAGE, NULL, OPEN_EXISTING,
62            FILE_ATTRIBUTE_NORMAL, NULL);
63
64    if(mqueue == INVALID_HANDLE_VALUE)
65      {
66        return 1;
67      }
68
69    queue = mqueue;
70
71    return 0;
72  }
73
74  int cleanup_ipc(void)
75  {
76    return CloseHandle(queue);
77  }
78
79  int send_ipc_message(IPCMessage *message)
80  {
81    DWORD bytes_sent = 0;
82    BOOL result = FALSE;
83
84    result = WriteFile(queue, message, sizeof(message), &bytes_sent, NULL);
85    if(!result)
86      {
87        return 1;
88      }
89
90    return 0;
91  }
92
93  int recv_ipc_message(IPCMessage *message)
94  {
95    DWORD bytes_read = 0;
96    BOOL result = FALSE;
97
98    result = ReadFile(queue, message, sizeof(message), &bytes_read, NULL);
99
100   if(!result)
101     {
102       return 1;
103     }
104   return 0;
105 }
106
107 #else
108 int init_ipc_server(void)
109 {
110   struct mq_attr attr;
111   mqd_t mqueue;
```

```
112
113   attr.mq_flags = 0;
114   attr.mq_maxmsg = 2;
115   attr.mq_msgsize = sizeof(IPCMessage);
116   attr.mq_curmsgs = 0;
117   mqueue = mq_open("/dfw", O_CREAT | O_RDWR, 0644, &attr);
118
119   if(mqueue == (mqd_t)-1)
120     {
121       return 1;
122     }
123
124   queue = mqueue;
125   return 0;
126 }
127
128 int init_ipc_client(void)
129 {
130   mqd_t mqueue;
131
132   mqueue = mq_open("/dfw", O_RDWR);
133
134   if(mqueue == (mqd_t)-1)
135     {
136       return 1;
137     }
138
139   queue = mqueue;
140   return 0;
141 }
142
143 int cleanup_ipc(void)
144 {
145   return mq_close(queue);
146 }
147
148 int send_ipc_message(IPCMessage *message)
149 {
150   if(mq_send(queue, (void*)message, sizeof(IPCMessage), 0) == -1)
151     {
152       return 1;
153     }
154   return 0;
155 }
156
157 int recv_ipc_message(IPCMessage *message)
158 {
159   if(mq_receive(queue, (void*)message, sizeof(IPCMessage), 0) == -1)
160     {
161       return 1;
162     }
163
164   return 0;
165 }
166 #endif
```

Listing A.8: net.h

```
1  /**
2   * @file net.h
3   * @brief Network and protocol interface.
4   * @author Adam Bruce
5   * @date 22 Mar 2021
6   */
7
8  #ifndef NET_H
9  #define NET_H
10
11 #include "socket.h"
12 #include "firewall.h"
13
14 #include <openssl/sha.h>
15
```

```c
#ifdef _WIN32
#include <inttypes.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#else
#include <sys/socket.h>
#include <netdb.h>
#endif

/**
 * @brief Port for receiving messages
 */
#define PORT_RECV 8070

/**
 * @brief Port for sending messages
 */
#define PORT_SEND 8071

/**
 * @brief The maximum number of network hops.
 */
#define MAX_HOPS 5

/**
 * @brief The local IP address.
 */
static char local_address[INET_ADDRSTRLEN];

/**
 * @brief All available message types for network transactions.
 */
typedef enum
  {
    ADVERTISEMENT,  /**< Host advertisement message             */
    CONSENSUS,      /**< Firewall transaction consensus message */
    RULE            /**< Firewall transaction rule message      */
  } MessageType;

/**
 * @brief All available message subtypes for network transactions.
 */
typedef enum
  {
    BROADCAST,      /**< Broadcast message       */
    ACK             /**< Acknowledgement message */
  } MessageSubType;

/**
 * @brief The structure to store all known hosts as a linked list.
 */
struct HostList
{
  struct HostList *next;       /**< The next host in the list   */
  char addr[INET_ADDRSTRLEN];  /**< The host's address          */
  uint8_t ack;                 /**< The ack status for the host */
};
typedef struct HostList HostList;

/**
 * @brief The structure to store an advertisement message.
 */
typedef struct
{
  MessageType type;                  /**< The message type (ADVERTISEMENT)  */
  uint8_t hops;                      /**< The hop count                     */
  MessageSubType advertisement_type; /**< The message subtype               */
  char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
  char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
  char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
} AdvertisementMessage;

/**
```

```
89  * @brief The structure to store a consensus message.
90  */
91 typedef struct
92 {
93   MessageType type;                  /**< The message type (CONSENSUS)      */
94   uint8_t hops;                      /**< The hop count                     */
95   MessageSubType consensus_type;     /**< The message subtype               */
96   char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
97   char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
98   char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
99   unsigned char last_block_hash[SHA256_DIGEST_LENGTH]; /**< The hash of the
100                  last block      */
101 } ConsensusMessage;
102
103 /**
104  * @brief The structure of a firewall rule message.
105  */
106 typedef struct
107 {
108   MessageType type;                  /**< The message type (RULE)           */
109   uint8_t hops;                      /**< The hop count                     */
110   MessageSubType rule_type;          /**< The message subtype               */
111   char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
112   char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
113   char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
114   FirewallRule rule;                 /**< The firewall rule                 */
115 } RuleMessage;
116
117 /**
118  * @brief Retrieves the local address of the host's Ethernet adapter.
119  *
120  * Retrieves the local address of the host's Ethernet adapter using the network
121  * API of the OS.
122  * @param buffer the buffer to copy the address into.
123  * @return whether the address was succesfully obtained. If an error has
124  * occurred, the return value will be 1, otherwise the return value will be 0.
125  */
126 int get_local_address(char *buffer);
127
128 /**
129  * @brief Returns the current number of acknowledgements.
130  *
131  * Returns the current number of acknowledgements this host has received since
132  * sending it's consensus message.
133  * @return The number of acknowledgements.
134  */
135 int get_acks(void);
136
137 /**
138  * @brief Resets the number of acknowledgements.
139  *
140  * Sets the ack state of each host to 0.
141  * @return whether the acknowledgements were succesfully reset. If an error has
142  * occurred, the return value will be 1, otherwise the return value will be 0.
143  */
144 int reset_acks(void);
145
146 /**
147  * @brief Sets the acknowledgement state of a host.
148  *
149  * Sets the acknowledgement state of the given host to 1.
150  * @param addr the address of the host who's acknowledgement should be set.
151  * @return if the acknowledgement was set successfully. If an error has
152  * occurred, the return value will be 1, otherwise the return value will be 0.
153  */
154 int set_ack(char *addr);
155
156 /**
157  * @brief Loads a list of hosts from a file.
158  *
159  * Loads a list of hosts from the given file into the HostList struct.
160  * @param fname the name of the file containing the hosts.
161  * @return whether the list of hosts was successfully loaded. If an error has
```

```
162  * occurred, the return value will be 1, otherwise the return value will be 0.
163  */
164 int load_hosts_from_file(const char *fname);
165
166 /**
167  * @brief Saves all known hosts currently loaded into a file.
168  *
169  * Saves all hosts currently stored in the HostList struct into a file.
170  * @param fname the name of the file to save the hosts.
171  * @return whether the list of hosts was successfully saved. If an error has
172  * occurred, the return value will be 1, otherwise the return value will be 0.
173  */
174 int save_hosts_to_file(const char *fname);
175
176 /**
177  * @brief Adds a host to the host list.
178  *
179  * Appends the given host to the list of hosts.
180  * @param addr the address of the new host.
181  * @return whether the host was appended successfully. If an error has
182  * occurred, the return value will be 1, otherwise the return value will be 0.
183  */
184 int add_host(char* addr);
185
186 /**
187 * @brief Checks if a given host exists in the host list.
188 *
189 * Searches the list of hosts for the given address.
190 * @param addr the address to search for.
191 * @return whether the host was found. If the host was found, the return value
192 * will be 1, otherwise the return value will be 0.
193 */
194 int check_host_exists(char *addr);
195
196 /**
197  * @brief Returns the number of remote hosts known by the local host.
198  *
199  * Counts how many hosts are known locally.
200  * @return the number of hosts.
201  */
202 int get_host_count(void);
203
204 /**
205  * @brief Initialises the network API.
206  *
207  * Initialises the network API by initialising the underlying socket API and
208  * creating the necessary sockets for sending and receiving messages.
209  * @return the status of the network API. If an error has occurred, a non-zero
210  * value will be returned, otherwise the return value will be 0.
211  */
212 int init_net(void);
213
214 /**
215  * @brief Uninitialises the network API.
216  *
217  * Uninitialises the network API by closing the underlying sockets and cleaning
218  * up the relevant socket API.
219  * @return whether the network API was successfully cleaned up. If an error has
220  * occurred, a non-zero value will be returned, otherwise the return value will
221  * be 0.
222  */
223 int cleanup_net(void);
224
225 /**
226  * @brief Sends a message to a remote host.
227  *
228  * Sends a message to the remote host specified by their IP address.
229  * @param ip_address the remote host's IP address.
230  * @param message the message / data to send to the remote host.
231  * @param length the length of the message / data.
232  * @return the number of bytes sent to the remote host. If an error has
233  * occurred, a negative value will be returned.
234  */
```

```
235 int send_to_host(char *ip_address, void *message, size_t length);
236
237 /**
238  * @brief Sends an advertisement message.
239  *
240  * Sends an advertisement to a remote host using the address information within
241  * the message.
242  * @param message the message to send.
243  * @return the number of bytes sent. If an error has occurred, the return value
244  * will be negative.
245  */
246 int send_advertisement_message(AdvertisementMessage *message);
247
248 /**
249  * @brief Sends an advertisement message to all known hosts.
250  *
251  * Sends an advertisement message to all known hosts. The address within the
252  * given message will be modified.
253  * @param message the message to send.
254  * @return whether all messages were sent successfully. If an error has
255  * occurred, the return value will be 1, otherwise the return value will be 0.
256  */
257 int send_to_all_advertisement_message(AdvertisementMessage *message);
258
259 /**
260  * @brief Parses a received raw advertisement message.
261  *
262  * Parses raw memory into an instance of an AdvertisementMessage. Upon
263  * identifying the message subtype, either recv_advertisement_broadcast or
264  * recv_advertisement_ack is called.
265  * @param buffer the raw memory of the message.
266  * @return whether the advertisement message was parsed successfully. If an
267  * error has occurred, the return value will be 1, otherwise the return value
268  * will be 0.
269  */
270 int recv_advertisement_message(void *buffer);
271
272 /**
273  * @brief Handles advertisement broadcasts.
274  *
275  * Handles advertisement broadcast messages. If the host is not known, then
276  * they are appended to the host list. Additionally, if the hop count has not
277  * exceeded the hop limit, it is forwarded to all known hosts.
278  * @param message the received message.
279  * @return whether the message was handled correctly. If an error has occurred,
280  * the return value will be 1, otherwise the return value will be 0.
281  */
282 int recv_advertisement_broadcast(AdvertisementMessage *message);
283
284 /**
285  * @brief Handles advertisement acknowledgements.
286  *
287  * Handles advertisement acknowledgement messages. Upon receiving an ack, if
288  * the host is not known, then thay are appended to the host list.
289  * @param message the received message.
290  * @return whether the message was handled correctly. If an error has occurred,
291  * the return value will be 1, otherwise the return value will be 0.
292  */
293 int recv_advertisement_ack(AdvertisementMessage *message);
294
295 /**
296  * @brief Sends a consensus message.
297  *
298  * Sends a consensus message to a remote host using the address information
299  * within the message.
300  * @param message the message to send.
301  * @return the number of bytes sent. If an error has occurred, the return value
302  * will be negative.
303  */
304 int send_consensus_message(ConsensusMessage *message);
305
306 /**
307  * @brief Sends a consensus message to all known hosts.
```

```
308  *
309  * Sends a consensus message to all known hosts. The address within the given
310  * message will be modified.
311  * @param message the message to send.
312  * @return whether all messages were sent successfully. If an error has
313  * occurred, the return value will be 1, otherwise the return value will be 0.
314  */
315 int send_to_all_consensus_message(ConsensusMessage *message);
316
317 /**
318  * @brief Parses a received raw consensus message.
319  *
320  * Parses raw memory into an instance of an ConsensusMessage. Upon
321  * identifying the message subtype, either recv_consensus_broadcast or
322  * recv_consensus_ack is called.
323  * @param buffer the raw memory of the message.
324  * @return whether the consensus message was parsed successfully. If an error
325  * has occurred, the return value will be 1, otherwise the return value will be
326  * 0.
327  */
328 int recv_consensus_message(void *buffer);
329
330 /**
331  * @brief Handles consensus broadcasts.
332  *
333  * Handles consensus broadcast messages. If the host is known, and the
334  * consensus hash matches the host's last hash, then an ack is sent.
335  * Additionally, if the hop count has not exceeded the hop limit, the broadcast
336  * is forwarded to all known hosts.
337  * @param message the received message.
338  * @return whether the message was handled correctly. If an error has occurred,
339  * the return value will be 1, otherwise the return value will be 0.
340  */
341 int recv_consensus_broadcast(ConsensusMessage *message);
342
343 /**
344  * @brief Handles consensus acknowledgements.
345  *
346  * Handles consensus acknowledgement messages. Upon receiving an ack, the
347  * ack_count is incremented.
348  * @param message the received message.
349  * @return whether the message was handled correctly. If an error has occurred,
350  * the return value will be 1, otherwise the return value will be 0.
351  */
352 int recv_consensus_ack(ConsensusMessage *message);
353
354 /**
355  * @brief Sends a firewall rule message.
356  *
357  * Sends a firewall rule message to a remote host using the address information
358  * within the message.
359  * @param message the message to send.
360  * @return the number of bytes sent. If an error has occurred, the return value
361  * will be negative.
362  */
363 int send_rule_message(RuleMessage *message);
364
365 /**
366  * @brief Sends a firewall rule message to all known hosts.
367  *
368  * Sends a firewall rule message to all known hosts. The address within the
369  * given message will be modified.
370  * @param message the message to send.
371  * @return whether all messages were sent successfully. If an error has
372  * occurred, the return value will be 1, otherwise the return value will be 0.
373  */
374 int send_to_all_rule_message(RuleMessage *message);
375
376 /**
377  * @brief Parses a received raw firewall rule message.
378  *
379  * Parses raw memory into an instance of an RuleMessage. Upon identifying the
380  * message subtype, recv_rule_broadcast is called.
```

```
 * @param buffer the raw memory of the message.
 * @return whether the firewall rule message was parsed successfully. If an
 * error has occurred, the return value will be 1, otherwise the return value
 * will be 0.
 */
int recv_rule_message(void *buffer);

/**
 * @brief Handles firewall rule broadcasts.
 *
 * Handles firewall rule messages. If the host is known, and the host has sent
 * a consensus ack, then the firewall rule is accepted and appended to the
 * chain.
 * @param message the received message.
 * @return whether the message was handled correctly. If an error has occurred,
 * the return value will be 1, otherwise the return value will be 0.
 */
int recv_rule_broadcast(RuleMessage *message);

/**
 * @brief Waits for a message to be received.
 *
 * Waits for a message to be recieved. Once received, <length> bytes will be
 * copied into the given buffer.
 * @param buffer the buffer to copy the message into.
 * @param length the number of bytes to read.
 * @return the number of bytes received. If an error has occurred, a negative
 * value will be returned.
 */
int poll_message(void *buffer, size_t length);

#endif
```

Listing A.9: net.c

```
/**
 * @file net.c
 * @brief Network and protocol interface
 * @author Adam Bruce
 * @date 22 Mar 2021
 */

#include "net.h"
#include "blockchain.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#include <openssl/sha.h>

#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#undef INET_ADDRSTRLEN
#define INET_ADDRSTRLEN 16
#else
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
#endif

/**
 * @brief The maximum length of a *nix ethernet adapter prefix.
 */
#define ETH_PREFIX_LEN 5

/**
 * @brief The local sending socket.
 */
```

```
39 static socket_t socket_send;
40
41 /**
42  * @brief The local receiving socket.
43  */
44 static socket_t socket_recv;
45
46 /**
47  * @brief The list of known hosts.
48  */
49 static HostList *host_list;
50
51 #ifndef _WIN32
52 /**
53  * @brief Known ethernet adapter prefixes.
54  *
55  * Used for obtaining the assigned ethernet address.
56  */
57 static char eth_prefixes[6][ETH_PREFIX_LEN + 1] = {
58   "eth", "em", "ed", "genet", "usmsc", "\0"
59 };
60 #endif
61
62
63 #ifdef _WIN32
64 #define ADAPTER_NAME_LEN 8
65 int get_local_address(char* buffer)
66 {
67   DWORD rv, size;
68   PIP_ADAPTER_ADDRESSES adapter_addresses, aa;
69   PIP_ADAPTER_UNICAST_ADDRESS ua;
70   char address[INET_ADDRSTRLEN], name[ADAPTER_NAME_LEN];
71
72   rv = GetAdaptersAddresses(AF_INET, GAA_FLAG_INCLUDE_PREFIX, NULL, NULL,
73           &size);
74   if (rv != ERROR_BUFFER_OVERFLOW)
75     {
76       return 1;
77     }
78   adapter_addresses = (PIP_ADAPTER_ADDRESSES)malloc(size);
79
80   rv = GetAdaptersAddresses(AF_INET, GAA_FLAG_INCLUDE_PREFIX, NULL,
81           adapter_addresses, &size);
82   if (rv != ERROR_SUCCESS)
83     {
84       free(adapter_addresses);
85       return 1;
86     }
87
88   for (aa = adapter_addresses; aa != NULL; aa = aa->Next)
89     {
90       memset(name, '\0', ADAPTER_NAME_LEN);
91       WideCharToMultiByte(CP_ACP, 0, aa->FriendlyName,
92         wcslen(aa->FriendlyName), name, ADAPTER_NAME_LEN,
93         NULL, NULL);
94
95       if(strncmp(name, "Ethernet", ADAPTER_NAME_LEN) == 0)
96   {
97     for (ua = aa->FirstUnicastAddress; ua != NULL; ua = ua->Next)
98       {
99         memset(address, '\0', INET_ADDRSTRLEN);
100        getnameinfo(ua->Address.lpSockaddr, ua->Address.iSockaddrLength,
101        address, INET_ADDRSTRLEN, NULL, 0, NI_NUMERICHOST);
102
103        strncpy(buffer, address, INET_ADDRSTRLEN);
104        free(adapter_addresses);
105        return 0;
106      }
107  }
108    }
109
110  free(adapter_addresses);
111  return 1;
```

```
112 }
113 #else
114 int get_local_address(char *buffer)
115 {
116   struct ifaddrs *interfaces = NULL, *addr = NULL;
117   void *addr_ptr = NULL;
118   char addr_str[INET_ADDRSTRLEN];
119   int prefix_index, match;
120
121   if(getifaddrs(&interfaces) != 0)
122     {
123       return 1;
124     }
125
126   match = 0;
127   for(addr = interfaces; addr != NULL; addr = addr->ifa_next)
128     {
129       if(addr->ifa_addr->sa_family == AF_INET)
130   {
131     prefix_index = 0;
132     match = 0;
133     while(eth_prefixes[prefix_index][0] != '\0')
134       {
135         if(strstr(addr->ifa_name, eth_prefixes[prefix_index]))
136     {
137       match = 1;
138       break;
139     }
140         prefix_index++;
141       }
142   }
143
144       if(match)
145   {
146     addr_ptr = &((struct sockaddr_in*)addr->ifa_addr)->sin_addr;
147     inet_ntop(addr->ifa_addr->sa_family,
148         addr_ptr,
149         addr_str,
150         sizeof(addr_str));
151
152     strcpy(buffer, addr_str);
153     break;
154   }
155     }
156   freeifaddrs(interfaces);
157   return !match;
158 }
159 #endif
160
161 int get_acks(void)
162 {
163   HostList *host;
164   int count;
165
166   host = host_list;
167   if(!host || strlen(host->addr) == 0)
168     {
169       return 0;
170     }
171
172   count = 0;
173   while(host)
174     {
175       if(host->ack > 0)
176   {
177     count++;
178   }
179       host = host->next;
180     }
181
182   return count;
183 }
184
```

```
185 int reset_acks(void)
186 {
187   HostList *host;
188
189   host = host_list;
190   if(!host || strlen(host->addr) == 0)
191     {
192       return 1;
193     }
194
195   while(host)
196     {
197       host->ack = 0;
198       host = host->next;
199     }
200
201   return 0;
202 }
203
204 int set_ack(char *addr)
205 {
206   HostList *host;
207
208   host = host_list;
209   if(!host || strlen(host->addr) == 0)
210     {
211       return 1;
212     }
213
214   while(host)
215     {
216       if(strncmp(host->addr, addr, INET_ADDRSTRLEN) == 0)
217   {
218     host->ack = 1;
219     return 0;
220   }
221       host = host->next;
222     }
223
224   return 1;
225 }
226
227 int load_hosts_from_file(const char *fname)
228 {
229   FILE *file;
230   int c;
231   size_t pos;
232   char buffer[INET_ADDRSTRLEN + 10];
233
234   if(!host_list)
235     {
236       printf("[ ERR  ] Network stack not yet initialised\n");
237       return 1;
238     }
239
240   file = fopen(fname, "r");
241   if(!file)
242     {
243       printf("[ NET  ] No hostfile found\n");
244       return 1;
245     }
246
247   pos = 0;
248   while((c = fgetc(file)) != EOF)
249     {
250       if((char)c == '\r')
251   {
252     continue;
253   }
254
255       if((char)c == '\n')
256   {
257     buffer[pos] = '\0';
```

```
258       add_host(buffer);
259       memset(buffer, '\0', INET_ADDRSTRLEN + 10);
260       pos = 0;
261       continue;
262     }
263
264         buffer[pos++] = (char)c;
265       }
266
267     fclose(file);
268     return 0;
269   }
270
271   int save_hosts_to_file(const char *fname)
272   {
273     FILE *file;
274     HostList *host;
275
276     if(!host_list)
277       {
278         printf("[ ERR  ] Network stack not yet initialised\n");
279         return 1;
280       }
281
282     file = fopen(fname, "w+");
283     if(!file)
284       {
285         printf("[ ERR  ] Could not create hostfile\n");
286         return 1;
287       }
288
289     host = host_list;
290     if(host)
291       {
292         while(host && strlen(host->addr) > 0)
293     {
294       fwrite(host->addr, strlen(host->addr), 1, file);
295       fputc('\n', file);
296       host = host->next;
297     }
298       }
299
300     fclose(file);
301     return 0;
302   }
303
304   int add_host(char *addr)
305   {
306     HostList *host;
307
308     printf("[ NET  ] Adding new host (%s)\n", addr);
309
310     if(!host_list)
311       {
312         return 1;
313       }
314     if(!addr)
315       {
316         return 1;
317       }
318
319     host = host_list;
320     if(strlen(host->addr) == 0)
321       {
322         strncpy(host->addr, addr, INET_ADDRSTRLEN);
323       }
324     else
325       {
326         while(host && host->next)
327     {
328       host = host->next;
329     }
330
```

```
331       host->next = (HostList*)malloc(sizeof(HostList));
332       memset(host->next, 0, sizeof(HostList));
333       strncpy(host->next->addr, addr, INET_ADDRSTRLEN);
334     }
335   return save_hosts_to_file("hosts.txt");
336 }
337
338
339
340 int check_host_exists(char *addr)
341 {
342   HostList *host;
343
344   host = host_list;
345   if(!host)
346     {
347       printf("[ ERR  ] Network stack not yet initialised\n");
348       return 0;
349     }
350   if(!addr)
351     {
352       return 0;
353     }
354
355   while(host)
356     {
357       if(strncmp(addr, host->addr, INET_ADDRSTRLEN) == 0)
358   {
359     return 1;
360   }
361       host = host->next;
362     }
363
364   return 0;
365 }
366
367 int get_host_count(void)
368 {
369   HostList *host;
370   int count;
371
372   host = host_list;
373   if(!host || strlen(host->addr) == 0)
374     {
375       return 0;
376     }
377
378   count = 0;
379   while(host)
380     {
381       count++;
382       host = host->next;
383     }
384
385   return count;
386 }
387
388 int init_net(void)
389 {
390   if(init_sockets() != 0)
391     {
392       return 1;
393     }
394   if((socket_send = create_socket()) == 0)
395     {
396       return 1;
397     }
398   if(bind_socket(socket_send, PORT_SEND) != 0)
399     {
400       return 1;
401     }
402   if((socket_recv = create_socket()) == 0)
403     {
```

```
404       return 1;
405     }
406
407   if(bind_socket(socket_recv, PORT_RECV) != 0)
408     {
409       return 1;
410     }
411
412   if(get_local_address(local_address) != 0)
413     {
414       return 1;
415     }
416
417   host_list = (HostList*)malloc(sizeof(HostList));
418   if(!host_list)
419     {
420       return 1;
421     }
422   memset(host_list, 0, sizeof(HostList));
423
424   return 0;
425 }
426
427 int cleanup_net(void)
428 {
429   HostList *host, *temp;
430   host = host_list;
431
432   while(host)
433     {
434       temp = host;
435       host = host->next;
436       free(temp);
437     }
438
439   close_socket(socket_send);
440   close_socket(socket_recv);
441   return cleanup_sockets();
442 }
443
444 int send_to_host(char *ip_address, void *message, size_t length)
445 {
446   struct sockaddr_in remote_addr;
447
448   remote_addr.sin_family = AF_INET;
449   remote_addr.sin_addr.s_addr = inet_addr(ip_address);
450   remote_addr.sin_port = htons(PORT_RECV);
451
452   return send_to_socket(socket_send, message, length, 0, remote_addr);
453 }
454
455 int send_advertisement_message(AdvertisementMessage *message)
456 {
457   char buffer[11];
458   struct in_addr addr;
459   int status;
460
461   buffer[0] = message->type;
462   buffer[1] = message->hops;
463   buffer[2] = message->advertisement_type;
464
465   status = inet_pton(AF_INET, message->source_addr, &addr);
466   if(status == 0)
467     {
468       return 1;
469     }
470   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
471
472   status = inet_pton(AF_INET, message->next_addr, &addr);
473   if(status == 0)
474     {
475       return 1;
476     }
```

```
477   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
478
479   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
480 }
481
482 int send_to_all_advertisement_message(AdvertisementMessage *message)
483 {
484   HostList *host;
485   host = host_list;
486   while(host)
487     {
488       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
489       send_advertisement_message(message);
490       host = host->next;
491     }
492   return 0;
493 }
494
495 int recv_advertisement_message(void *buffer)
496 {
497   AdvertisementMessage message;
498   char *char_buffer;
499   struct sockaddr_in target, source;
500
501   char_buffer = (char*)buffer;
502   message.type = char_buffer[0];
503   message.hops = char_buffer[1];
504   message.advertisement_type = char_buffer[2];
505
506   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
507   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
508   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
509   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
510
511   if(strncmp(local_address, message.source_addr, INET_ADDRSTRLEN) == 0)
512     {
513       return 0;
514     }
515
516   switch(message.advertisement_type)
517     {
518     case BROADCAST:
519       recv_advertisement_broadcast(&message);
520       break;
521     case ACK:
522       recv_advertisement_ack(&message);
523       break;
524     }
525
526   return 0;
527 }
528
529 int recv_advertisement_broadcast(AdvertisementMessage *message)
530 {
531   AdvertisementMessage new_message;
532
533   if(!message)
534     {
535       return 1;
536     }
537
538   /* If new host, add */
539   if(!check_host_exists(message->source_addr))
540     {
541       add_host(message->source_addr);
542       new_message.type = ADVERTISEMENT;
543       new_message.hops = 0;
544       new_message.advertisement_type = ACK;
545       strncpy(new_message.source_addr, local_address, INET_ADDRSTRLEN);
546       strncpy(new_message.target_addr, message->source_addr, INET_ADDRSTRLEN);
547
548       /* Send ACK */
549       send_to_all_advertisement_message(&new_message);
```

```
550     }
551
552   /* If under max hop count, forward to all hosts */
553   if(message->hops < MAX_HOPS)
554     {
555       memcpy(&new_message, message, sizeof(AdvertisementMessage));
556       new_message.hops++;
557       send_to_all_advertisement_message(&new_message);
558     }
559
560   return 0;
561 }
562
563 int recv_advertisement_ack(AdvertisementMessage* message)
564 {
565   AdvertisementMessage new_message;
566
567   if(!message)
568     {
569       return 1;
570     }
571
572   /* Check if we are the intended recipient */
573   if(strncmp(local_address, message->target_addr, INET_ADDRSTRLEN) == 0
574      && !check_host_exists(message->source_addr))
575     {
576       add_host(message->source_addr);
577     }
578   else
579     {
580       if(message->hops < MAX_HOPS)
581   {
582     memcpy(&new_message, message, sizeof(AdvertisementMessage));
583     new_message.hops++;
584     send_to_all_advertisement_message(&new_message);
585   }
586     }
587
588   return 0;
589 }
590
591 int send_consensus_message(ConsensusMessage *message)
592 {
593   char buffer[11 + SHA256_DIGEST_LENGTH];
594   char hash_string[SHA256_STRING_LENGTH + 1];
595   struct in_addr addr;
596   int status;
597
598   buffer[0] = message->type;
599   buffer[1] = message->hops;
600   buffer[2] = message->consensus_type;
601
602   status = inet_pton(AF_INET, message->source_addr, &addr);
603   if(status == 0)
604     {
605       return 1;
606     }
607   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
608
609   status = inet_pton(AF_INET, message->target_addr, &addr);
610   if(status == 0)
611     {
612       return 1;
613     }
614   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
615
616
617   memcpy(buffer + 11, message->last_block_hash, SHA256_DIGEST_LENGTH);
618   get_hash_string(hash_string, message->last_block_hash,
619       SHA256_STRING_LENGTH + 1);
620
621   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
622 }
```

```
623
624 int send_to_all_consensus_message(ConsensusMessage *message)
625 {
626   HostList *host;
627   host = host_list;
628   while(host && strlen(host->addr) > 0)
629     {
630       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
631       send_consensus_message(message);
632       host = host->next;
633     }
634   return 0;
635 }
636
637 int recv_consensus_message(void *buffer)
638 {
639   ConsensusMessage message;
640   char *char_buffer;
641   struct sockaddr_in target, source;
642
643   char_buffer = (char*)buffer;
644   message.type = char_buffer[0];
645   message.hops = char_buffer[1];
646   message.consensus_type = char_buffer[2];
647
648   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
649   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
650   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
651   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
652
653   memcpy(message.last_block_hash, char_buffer + 11, SHA256_DIGEST_LENGTH);
654
655   switch(message.consensus_type)
656     {
657     case BROADCAST:
658       recv_consensus_broadcast(&message);
659       break;
660     case ACK:
661       recv_consensus_ack(&message);
662       break;
663     }
664
665   return 0;
666 }
667
668 int recv_consensus_broadcast(ConsensusMessage *message)
669 {
670   ConsensusMessage new_message;
671   unsigned char last_hash[SHA256_DIGEST_LENGTH];
672   char hash_string[SHA256_STRING_LENGTH + 1];
673   FirewallBlock *last_block;
674
675
676   if(!message)
677     {
678       return 1;
679     }
680
681   /* We've just received our own broadcast, do nothing */
682   if(strncmp(message->source_addr, local_address, INET_ADDRSTRLEN) == 0)
683     {
684       return 0;
685     }
686
687   get_hash_string(hash_string, message->last_block_hash,
688       SHA256_STRING_LENGTH + 1);
689
690   /* If hashes match, add to pending_rules */
691   get_last_hash(last_hash);
692   if(memcmp(last_hash, "\0", 1) == 0 ||
693      memcmp(message->last_block_hash, last_hash, SHA256_DIGEST_LENGTH) == 0)
694     {
695       if(!is_pending(message->source_addr))
```

```
696   {
697     printf("[ CONS ] Received consensus message with matching hash\n");
698     add_pending_rule(message->source_addr);
699     new_message.type = CONSENSUS;
700     new_message.hops = 0;
701     new_message.consensus_type = ACK;
702     strncpy(new_message.source_addr, local_address, INET_ADDRSTRLEN);
703     strncpy(new_message.target_addr, message->source_addr, INET_ADDRSTRLEN);
704     memcpy(new_message.last_block_hash, message->last_block_hash,
705      SHA256_DIGEST_LENGTH);
706     send_to_all_consensus_message(&new_message);
707   }
708
709     }
710   else
711     {
712       last_block = NULL;
713       get_last_block(last_block);
714       if(last_block != NULL && memcmp(message->last_block_hash, last_block->last_hash,↩
              SHA256_DIGEST_LENGTH) != 0)
715   {
716     printf("[ CONS ] Received consensus message with mismatched hash\n");
717   }
718     }
719
720   /* If under max hop count, forward to all hosts */
721   if(message->hops < MAX_HOPS)
722     {
723       memcpy(&new_message, message, sizeof(ConsensusMessage));
724       new_message.hops++;
725       send_to_all_consensus_message(&new_message);
726     }
727
728   return 0;
729 }
730
731 int recv_consensus_ack(ConsensusMessage *message)
732 {
733   ConsensusMessage new_message;
734
735   if(!message)
736     {
737       return 1;
738     }
739
740   /* Check if we are the intended recipient */
741   if(strncmp(local_address, message->target_addr, INET_ADDRSTRLEN) == 0)
742     {
743       set_ack(message->source_addr);
744     }
745   else
746     {
747       if(message->hops < MAX_HOPS)
748   {
749     memcpy(&new_message, message, sizeof(ConsensusMessage));
750     new_message.hops++;
751     send_to_all_consensus_message(&new_message);
752   }
753     }
754
755   return 0;
756 }
757
758 int send_rule_message(RuleMessage *message)
759 {
760   char buffer[11 + 4 + 4 + 2 + 2 + 4];
761   struct in_addr addr;
762   int status;
763
764   buffer[0] = message->type;
765   buffer[1] = message->hops;
766   buffer[2] = message->rule_type;
767
```

```
768   status = inet_pton(AF_INET, message->source_addr, &addr);
769   if(status == 0)
770     {
771       return 1;
772     }
773   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
774
775   status = inet_pton(AF_INET, message->target_addr, &addr);
776   if(status == 0)
777     {
778       return 1;
779     }
780   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
781
782   status = inet_pton(AF_INET, (void*)&message->rule.source_addr, &addr);
783   if(status == 0)
784     {
785       return 1;
786     }
787   memcpy(buffer + 11, &addr.s_addr, sizeof(addr.s_addr));
788
789   status = inet_pton(AF_INET, (void*)&message->rule.dest_addr, &addr);
790   if(status == 0)
791     {
792       return 1;
793     }
794   memcpy(buffer + 15, &addr.s_addr, sizeof(addr.s_addr));
795
796   memcpy(buffer + 19, (void*)&message->rule.source_port, 2);
797   memcpy(buffer + 21, (void*)&message->rule.dest_port, 2);
798   memcpy(buffer + 23, (void*)&message->rule.action, 4);
799
800   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
801 }
802
803 int send_to_all_rule_message(RuleMessage *message)
804 {
805   HostList *host;
806   host = host_list;
807   while(host)
808     {
809       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
810       send_rule_message(message);
811       host = host->next;
812     }
813   return 0;
814 }
815
816 int recv_rule_message(void *buffer)
817 {
818   RuleMessage message;
819   char *char_buffer;
820   struct sockaddr_in target, source, fw_source, fw_dest;
821
822   memset(&message, '\0', sizeof(RuleMessage));
823   char_buffer = (char*)buffer;
824   message.type = char_buffer[0];
825   message.hops = char_buffer[1];
826   message.rule_type = char_buffer[2];
827
828   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
829   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
830   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
831   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
832
833   fw_source.sin_addr.s_addr = *(int*)(char_buffer + 11);
834   fw_dest.sin_addr.s_addr = *(int*)(char_buffer + 15);
835   inet_ntop(AF_INET, &fw_source.sin_addr, message.rule.source_addr,
836       INET_ADDRSTRLEN);
837   inet_ntop(AF_INET, &fw_dest.sin_addr, message.rule.dest_addr,
838       INET_ADDRSTRLEN);
839
840   memcpy(&message.rule.source_port, (uint16_t*)(char_buffer + 19), 2);
```

```
841   memcpy(&message.rule.dest_port, (uint16_t*)(char_buffer + 21), 2);
842   memcpy(&message.rule.action, (int*)(char_buffer + 23), 4);
843
844   switch(message.rule_type)
845     {
846     case BROADCAST:
847       recv_rule_broadcast(&message);
848       break;
849     case ACK:
850       break;
851     }
852
853   return 0;
854 }
855
856 int recv_rule_broadcast(RuleMessage *message)
857 {
858   FirewallBlock new_block;
859   unsigned char last_hash[SHA256_DIGEST_LENGTH];
860   RuleMessage new_message;
861
862   if(!message)
863     {
864       return 1;
865     }
866
867   if(is_pending(message->source_addr))
868     {
869       get_last_hash(last_hash);
870       memset(&new_block, '\0', sizeof(FirewallBlock));
871       memcpy(new_block.last_hash, last_hash, SHA256_DIGEST_LENGTH);
872       strncpy(new_block.author, message->source_addr, INET_ADDRSTRLEN);
873       memcpy(&new_block.rule, &message->rule, sizeof(FirewallRule));
874       add_block_to_chain(&new_block);
875       recv_new_rule(&new_block.rule);
876
877       remove_pending_rule(message->source_addr);
878     }
879
880   /* If under max hop count, forward to all hosts */
881   if(message->hops < MAX_HOPS)
882     {
883       memcpy(&new_message, message, sizeof(RuleMessage));
884       new_message.hops++;
885       send_to_all_rule_message(&new_message);
886     }
887
888   return 0;
889 }
890
891 int poll_message(void *buffer, size_t length)
892 {
893   int bytes_read;
894
895   bytes_read = recv_from_socket(socket_recv, buffer, length, 0);
896   if(bytes_read <= 0)
897     {
898       return 0;
899     }
900
901   switch(((char*)buffer)[0])
902     {
903     case ADVERTISEMENT:
904       return recv_advertisement_message(buffer);
905       break;
906     case CONSENSUS:
907       return recv_consensus_message(buffer);
908       break;
909     case RULE:
910       return recv_rule_message(buffer);
911       break;
912     default:
913       return bytes_read;
```

```
914     }
915 }
```

Listing A.10: socket.h

```
1  /**
2   * @file socket.h
3   * @brief Cross-platform socket interface.
4   * @author Adam Bruce
5   * @date 15 Dec 2020
6   */
7
8  #ifndef SOCKET_H
9  #define SOCKET_H
10
11 #ifdef _WIN32
12
13 #ifndef _WIN32_WINNT
14 #define _WIN32_WINNT 0x0501
15 #endif
16
17 #include <winsock2.h>
18 #else
19 #include <sys/socket.h>
20 #include <netinet/in.h>
21 #endif
22
23
24 #ifdef _WIN32
25 /**
26  * @brief Cross platform socket type.
27  */
28 typedef SOCKET socket_t;
29 #else
30 /**
31  * @brief Cross platform socket type.
32  */
33 typedef int socket_t;
34
35 /**
36  * @brief UNIX equivalent to WinSocks's INVALID_SOCKET constant.
37  */
38 #define INVALID_SOCKET -1
39 #endif
40
41 /**
42  * @brief Initialises the socket API.
43  *
44  * Initialises the relevent socket APIs for each operating system.
45  * For the NT kernel, this involves initialising Winsock. For UNIX systems,
46  * this function does nothing.
47  * @return the status of the socket API. If an error has occurred, a non-zero
48  * value will be returned, otherwise the return value will be 0.
49  */
50 int init_sockets(void);
51
52 /**
53  * @brief Uninitialises the socket API.
54  *
55  * Uninitialises the relevent socket APIs for each operating system.
56  * For the NT kernel, this involves uninitialising Winsock. For UNIX systems,
57  * this function does nothing.
58  * @return whether the API was succesfully cleaned up. If an error has
59  * occurred, a non-zero value will be returned, otherwise the return value will
60  * be 0.
61  */
62 int cleanup_sockets(void);
63
64 /**
65  * @brief Creates a new socket.
66  *
67  * Creates a UDP socket using the relevant API for the operating system.
68  * @return a new socket descriptor, or 0 if a socket could not be created.
```

```
 */
socket_t create_socket(void);

/**
 * @brief Closes a socket.
 *
 * Closes the socket using the relevant API for the operating system.
 * @param sock the socket to close.
 */
void close_socket(socket_t sock);

/**
 * @brief Binds a socket to a port.
 *
 * Binds the socket to a port, and configures it to use IP and UDP.
 * @param sock the socket to bind.
 * @param port the port to bind the socket to.
 * @return whether the socket was successfully binded. If an error has
 * occurred, the return value will be -1, otherwise the return value will
 * be 0.
 */
int bind_socket(socket_t sock, int port);

/**
 * @brief Sends a message to a remote socket.
 *
 * Sends the data stored within the buffer to a remote socket.
 * @param sock the local socket.
 * @param message the data to send.
 * @param length the length of the data.
 * @param flags the flags used to configure the sendto operation.
 * @param dest_addr the destination address
 * @return how many bytes were successfully sent. If an error has occurred,
 * a negative value will be returned.
 */
int send_to_socket(socket_t sock, void *message, size_t length, int flags,
                   struct sockaddr_in dest_addr);

/**
 * @brief Receives a message from a socket.
 *
 * Receives a message from a socket.
 * @param sock the socket.
 * @param buffer the buffer to read the message into.
 * @param length the number of bytes to read.
 * @param flags the flags used to configure the recv operation.
 * @return how many bytes were successfully read. If an error has occurred,
 * a negative value will be returned.
 */
int recv_from_socket(socket_t sock, void *buffer, size_t length, int flags);

#endif
```

Listing A.11: socket.c

```
/**
 * @file socket.c
 * @brief Cross-pflatform socket interface.
 * @author Adam Bruce
 * @date 15 Dec 2020
 */

#include "socket.h"
#include <stdio.h>
#include <string.h>

#ifdef _WIN32
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0501
#endif

#include <io.h>
#include <winsock2.h>
```

```
19 #else
20 #include <unistd.h>
21 #include <sys/time.h>
22 #include <sys/socket.h>
23 #include <arpa/inet.h>
24 #include <netinet/in.h>
25 #endif
26
27
28 int init_sockets(void)
29 {
30 #ifdef _WIN32
31   WSADATA wsa_data;
32 #endif
33   /*printf("[ INFO ] Setting up sockets.\n");*/
34 #ifdef _WIN32
35   return WSAStartup(MAKEWORD(1,1), &wsa_data);
36 #else
37   return 0;
38 #endif
39 }
40
41 int cleanup_sockets(void)
42 {
43   /*  printf("[ INFO ] Cleaning up sockets.\n");*/
44 #ifdef _WIN32
45   return WSACleanup();
46 #else
47   return 0;
48 #endif
49 }
50
51 socket_t create_socket(void)
52 {
53   socket_t sock;
54   #ifdef _WIN32
55   DWORD ival;
56   #else
57   struct timeval tv;
58   #endif
59
60   /*  printf("[ INFO ] Creating new socket.\n");*/
61 #ifdef _WIN32
62     ival = 1000;
63   sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
64   setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (const char*)&ival, sizeof(DWORD));
65 #else
66   memset(&tv, 0, sizeof(struct timeval));
67   tv.tv_sec = 1;
68   sock = socket(AF_INET, SOCK_DGRAM, 0);
69   setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(struct timeval));
70 #endif
71   return sock;
72 }
73
74 void close_socket(socket_t sock)
75 {
76   /*  printf("[ INFO ] Closing socket.\n");*/
77 #ifdef _WIN32
78   closesocket(sock);
79 #else
80   close(sock);
81 #endif
82 }
83
84 int bind_socket(socket_t sock, int port)
85 {
86   struct sockaddr_in addr;
87   /*  printf("[ INFO ] Binding socket. \n"); */
88
89   addr.sin_family = AF_INET;
90   addr.sin_addr.s_addr = INADDR_ANY;
91   addr.sin_port = htons(port);
```

```
92
93    return bind(sock, (struct sockaddr*)&addr, sizeof(addr));
94  }
95
96  int send_to_socket(socket_t sock, void *message, size_t length, int flags,
97                     struct sockaddr_in dest_addr)
98  {
99    /*  printf("[ INFO ] Sending message of length %zu to socket.\n", length);*/
100   return sendto(sock, message, length, flags, (struct sockaddr*)&dest_addr,
101     sizeof(dest_addr));
102 }
103
104 int recv_from_socket(socket_t sock, void *buffer, size_t length, int flags)
105 {
106   /* printf("[ INFO ] Attempting to read %zu bytes from socket.\n", length); */
107   return recv(sock, buffer, length, flags);
108 }
```

Appendix B

Client Program Source Code

Listing B.1: client.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#include "../ipc.h"

int main(void)
{
  IPCMessage m;
  char buffer[100];
  int running = 1;

  init_ipc_client();
  printf("*********************************************************************\n");
  printf("*             Decentralised Firewall IPC Interface                  *\n");
  printf("*                        by Adam Bruce                              *\n");
  printf("*********************************************************************\n");

  printf("\nAvailable commands:\n");
  printf("  enable   :  Enables communication over the network.\n");
  printf("  disable  :  Disables communication over the network.\n");
  printf("  rule     :  Generates a new block containing the rule,\n");
  printf("              and broadcasts the block for consensus.\n");
  printf("  shutdown :  Terminates the framework.\n");
  printf("  quit     :  Quits this program.\n\n");
  while(running)
    {
      printf("dfw>");
      memset(buffer, 0, 100);
      memset(&m, 0, sizeof(IPCMessage));
      scanf("%s", buffer);

      if(strlen(buffer) >= 4 && strncmp(buffer, "quit", 4) == 0)
  {
    running = 0;
  }
      else if(strlen(buffer) >= 4 && strncmp(buffer, "rule", 4) == 0)
  {
    scanf("%s", m.rule.source_addr);
    scanf("%hd", &m.rule.source_port);
    scanf("%s", m.rule.dest_addr);
    scanf("%hd", &m.rule.dest_port);
    scanf("%s", buffer);

    m.rule.action = DENY;
    if(strncmp(buffer, "ALLOW", 5) == 0 ||
       strncmp(buffer, "allow", 5) == 0)
      {
        m.rule.action = ALLOW;
      }

```

```
53      m.message_type = I_RULE;
54
55      printf("Sending firewall rule to daemon:\n");
56      printf("    Source Address:      %s\n", m.rule.source_addr);
57      printf("    Source Port:         %hd\n", m.rule.source_port);
58      printf("    Destination Address: %s\n", m.rule.dest_addr);
59      printf("    Destination Port:    %hd\n", m.rule.dest_port);
60      printf("    Action:              %s\n",
61       (m.rule.action == ALLOW ? "ALLOW" : "DENY"));
62
63      send_ipc_message(&m);
64    }
65        else if(strlen(buffer) >= 6 && strncmp(buffer, "enable", 6) == 0)
66    {
67      printf("Sending Enable message to daemon\n");
68      m.message_type = I_ENABLE;
69      send_ipc_message(&m);
70    }
71        else if(strlen(buffer) >= 7 && strncmp(buffer, "disable", 7) == 0)
72    {
73      printf("Sending Disable message to daemon\n");
74            m.message_type = I_DISABLE;
75      send_ipc_message(&m);
76    }
77        else if(strlen(buffer) >= 8 && strncmp(buffer, "shutdown", 8) == 0)
78    {
79      printf("Sending Shutdown message to daemon\n");
80      m.message_type = I_SHUTDOWN;
81      send_ipc_message(&m);
82    }
83        else
84    {
85      printf("Unknown command\n");
86    }
87      }
88
89    printf("Bye!\n");
90    cleanup_ipc();
91    return 0;
92  }
```

Appendix C

Code and Framework Documentation

Decentralised Distributed Firewall Framework

1.0

Generated by Doxygen 1.9.1

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 AdvertisementMessage Struct Reference

The structure to store an advertisement message.

`#include <net.h>`

Data Fields

- MessageType type
- uint8_t hops
- MessageSubType advertisement_type
- char source_addr [INET_ADDRSTRLEN]
- char target_addr [INET_ADDRSTRLEN]
- char next_addr [INET_ADDRSTRLEN]

3.1.1 Detailed Description

The structure to store an advertisement message.

3.1.2 Field Documentation

3.1.2.1 advertisement_type

`MessageSubType AdvertisementMessage::advertisement_type`

The message subtype

3.1.2.2 hops

```
uint8_t AdvertisementMessage::hops
```

The hop count

3.1.2.3 next_addr

```
char AdvertisementMessage::next_addr[INET_ADDRSTRLEN]
```

The next address of the message

3.1.2.4 source_addr

```
char AdvertisementMessage::source_addr[INET_ADDRSTRLEN]
```

The source address of the message

3.1.2.5 target_addr

```
char AdvertisementMessage::target_addr[INET_ADDRSTRLEN]
```

The target address of the message

3.1.2.6 type

```
MessageType AdvertisementMessage::type
```

The message type (ADVERTISEMENT)

The documentation for this struct was generated from the following file:

- src/net.h

3.2 ConsensusMessage Struct Reference

The structure to store a consensus message.

```
#include <net.h>
```

Data Fields

- MessageType type
- uint8_t hops
- MessageSubType consensus_type
- char source_addr [INET_ADDRSTRLEN]
- char target_addr [INET_ADDRSTRLEN]
- char next_addr [INET_ADDRSTRLEN]
- unsigned char last_block_hash [SHA256_DIGEST_LENGTH]

3.2.1 Detailed Description

The structure to store a consensus message.

3.2.2 Field Documentation

3.2.2.1 consensus_type

```
MessageSubType ConsensusMessage::consensus_type
```

The message subtype

3.2.2.2 hops

```
uint8_t ConsensusMessage::hops
```

The hop count

3.2.2.3 last_block_hash

```
unsigned char ConsensusMessage::last_block_hash[SHA256_DIGEST_LENGTH]
```

The hash of the last block

3.2.2.4 next_addr

```
char ConsensusMessage::next_addr[INET_ADDRSTRLEN]
```

The next address of the message

3.2.2.5 source_addr

```
char ConsensusMessage::source_addr[INET_ADDRSTRLEN]
```

The source address of the message

3.2.2.6 target_addr

```
char ConsensusMessage::target_addr[INET_ADDRSTRLEN]
```

The target address of the message

3.2.2.7 type

```
MessageType ConsensusMessage::type
```

The message type (CONSENSUS)

The documentation for this struct was generated from the following file:

- src/net.h

3.3 FirewallBlock Struct Reference

```
#include <blockchain.h>
```

Data Fields

- unsigned char last_hash [SHA256_DIGEST_LENGTH]
- char author [INET_ADDRSTRLEN]
- FirewallRule rule
- struct FirewallBlock * **next**

3.3.1 Detailed Description

A block containing information for a firewall transaction.

3.3.2 Field Documentation

3.3.2.1 author

```
char FirewallBlock::author[INET_ADDRSTRLEN]
```

The address of the block author

3.3.2.2 last_hash

```
unsigned char FirewallBlock::last_hash[SHA256_DIGEST_LENGTH]
```

The hash of the previous block

3.3.2.3 rule

```
FirewallRule FirewallBlock::rule
```

The firewall rule associated with the block

The documentation for this struct was generated from the following file:

- src/blockchain.h

3.4 FirewallRule Struct Reference

The structure of a firewall rule.

```
#include <firewall.h>
```

Data Fields

- char source_addr [INET_ADDRSTRLEN]
- uint16_t source_port
- char dest_addr [INET_ADDRSTRLEN]
- uint16_t dest_port
- FirewallAction action

3.4.1 Detailed Description

The structure of a firewall rule.

3.4.2 Field Documentation

3.4.2.1 action

```
FirewallAction FirewallRule::action
```

The rule's action

3.4.2.2 dest_addr

```
char FirewallRule::dest_addr[INET_ADDRSTRLEN]
```

The rule's destination address

3.4.2.3 dest_port

```
uint16_t FirewallRule::dest_port
```

The rule's destination port

3.4.2.4 source_addr

```
char FirewallRule::source_addr[INET_ADDRSTRLEN]
```

The rule's source address

3.4.2.5 source_port

```
uint16_t FirewallRule::source_port
```

The rule's source port

The documentation for this struct was generated from the following file:

- src/firewall.h

3.5 HostList Struct Reference

The structure to store all known hosts as a linked list.

```
#include <net.h>
```

Data Fields

- struct HostList * next
- char addr [INET_ADDRSTRLEN]
- uint8_t ack

3.5.1 Detailed Description

The structure to store all known hosts as a linked list.

3.5.2 Field Documentation

3.5.2.1 ack

```
uint8_t HostList::ack
```

The ack status for the host

3.5.2.2 addr

```
char HostList::addr[INET_ADDRSTRLEN]
```

The host's address

3.5.2.3 next

```
struct HostList* HostList::next
```

The next host in the list

The documentation for this struct was generated from the following file:

- src/net.h

3.6 IPCMessage Struct Reference

The structure of a IPC message.

```
#include <ipc.h>
```

Data Fields

- IPCMessageType message_type
- FirewallRule rule

3.6.1 Detailed Description

The structure of a IPC message.

3.6.2 Field Documentation

3.6.2.1 message_type

```
IPCMessageType IPCMessage::message_type
```

The IPC message type

3.6.2.2 rule

```
FirewallRule IPCMessage::rule
```

The firewall rule (if type is I_RULE)

The documentation for this struct was generated from the following file:

- src/ipc.h

3.7 RuleMessage Struct Reference

The structure of a firewall rule message.

```
#include <net.h>
```

Data Fields

- MessageType type
- uint8_t hops
- MessageSubType rule_type
- char source_addr [INET_ADDRSTRLEN]
- char target_addr [INET_ADDRSTRLEN]
- char next_addr [INET_ADDRSTRLEN]
- FirewallRule rule

3.7.1 Detailed Description

The structure of a firewall rule message.

3.7.2 Field Documentation

3.7.2.1 hops

```
uint8_t RuleMessage::hops
```

The hop count

3.7.2.2 next_addr

```
char RuleMessage::next_addr[INET_ADDRSTRLEN]
```

The next address of the message

3.7.2.3 rule

```
FirewallRule RuleMessage::rule
```

The firewall rule

3.7.2.4 rule_type

```
MessageSubType RuleMessage::rule_type
```

The message subtype

3.7.2.5 source_addr

```
char RuleMessage::source_addr[INET_ADDRSTRLEN]
```

The source address of the message

3.7.2.6 target_addr

```
char RuleMessage::target_addr[INET_ADDRSTRLEN]
```

The target address of the message

3.7.2.7 type

```
MessageType RuleMessage::type
```

The message type (RULE)

The documentation for this struct was generated from the following file:

- src/net.h

Chapter 4

File Documentation

4.1 src/blockchain.c File Reference

Functions for creating and validating blockchains.

```
#include "blockchain.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <openssl/sha.h>
#include <arpa/inet.h>
```

Macros

- #define **PENDING_RULES_BUF_LEN** 10

Functions

- int get_block_hash (unsigned char *buffer, FirewallBlock *block, int buffer_size)

  Calculates the SHA256 hash of a block.
- int get_hash_string (char *buffer, unsigned char *hash, int buffer_size)

  Formats a SHA256 digest into human-readable string.
- int **get_hash_from_string** (unsigned char *buffer, char *hash_string, int buffer_size)
- int add_block_to_chain (FirewallBlock *block)

  Adds a new firewall block onto the chain.
- int rotate_pending_rules (void)

  Rotates the pending firewall rules.
- int add_pending_rule (char *addr)

  Adds a new rule to the list of pending rules.
- int is_pending (char *addr)

  Checks if the given address has a pending rule.
- int remove_pending_rule (char *addr)

  Removes a pending rule from the list.
- int get_last_block (FirewallBlock *block)

Returns a pointer to the last firewall block.

- int get_last_hash (unsigned char *buffer)

  Returns the hash of the last firewall block in the chain.

- int load_blocks_from_file (const char *fname)

  Loads a list of firewall blocks from a file.

- int save_blocks_to_file (const char *fname)

  Saves the current loaded blockchain into a file.

- int free_chain (void)

  Frees the currently loaded blockchain.

4.1.1 Detailed Description

Functions for creating and validating blockchains.

Author

Adam Bruce

Date

22 Mar 2021

4.1.2 Function Documentation

4.1.2.1 add_block_to_chain()

```
int add_block_to_chain (
            FirewallBlock * block )
```

Adds a new firewall block onto the chain.

Appends the new firewall block to the linked list of firewall block.

Parameters

block	the new block to add to the chain.

Returns

whether the block has been added to the chain. If an the block is is null or the block's memory could not be allocated, the return value will be 1, otherwise the return value will be 0.

4.1.2.2 add_pending_rule()

```
int add_pending_rule (
            char * addr )
```

Adds a new rule to the list of pending rules.

Appends a new rule to the list of pending rules, this involves rotating the list, and adding the new rule's author.

Parameters

addr	the author of the new pending rule.

Returns

whether the rule was added. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.3 free_chain()

```
int free_chain (
            void  )
```

Frees the currently loaded blockchain.

Frees the memory currently allocated to blocks on the chain.

Returns

whether the chain was successfully freed. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.4 get_block_hash()

```
int get_block_hash (
            unsigned char * buffer,
            FirewallBlock * block,
            int buffer_size )
```

Calculates the SHA256 hash of a block.

Calculates the SHA256 hash of a block, storing the digest in the given buffer. This buffer should have a size of SHA256_DIGEST_LENGTH.

Parameters

buffer	the buffer to store the digest in.
block	a pointer to the block to hash.
buffer_size	the size of the buffer to store the hash in.

Returns

whether the hash has been calculated successfully. If any parameters are invalid, the return value will be 1, otherwise the return value will be 0.

4.1.2.5 get_hash_string()

```
int get_hash_string (
            char * buffer,
            unsigned char * hash,
            int buffer_size )
```

Formats a SHA256 digest into human-readable string.

Formats a SHA256 digest into a human-readable string, storing the result into the given buffer. This buffer should have a size of SHA256_STRING_LENGTH.

Parameters

buffer	the buffer to store the string in.
hash	the hash digest to format into a string.
buffer_size	the size of the buffer to store the string in.

Returns

whether the string has been formatted succesfully. If any parameters are invalid, the return value will be 1, otherwise the return value will be 0.

4.1.2.6 get_last_block()

```
int get_last_block (
            FirewallBlock * block )
```

Returns a pointer to the last firewall block.

Returns a pointer to the last firewall block in the chain.

Parameters

block	pointer to point to the last block.

Returns

whether the last block was successfully found. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.7 get_last_hash()

```
int get_last_hash (
            unsigned char * buffer )
```

Returns the hash of the last firewall block in the chain.

Gets the SHA256 hash of the last firewall block in the chain. If the chain is empty, the buffer will be empty.

Parameters

buffer	the buffer that the hash value will be copied into. This buffer should be at least SHA256_DIGEST_LENGTH bytes in size.

Returns

whether the hash value was copied successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.8 is_pending()

```
int is_pending (
            char * addr )
```

Checks if the given address has a pending rule.

Searches the pending rule list for the given address. If the address is found then the host has a pending rule.

Parameters

addr	the author to check for pending rules.

Returns

whether any pending rules for the author were found. If a pending rule is found, the return value will be 1, otherwise the return value will be 0.

4.1.2.9 load_blocks_from_file()

```
int load_blocks_from_file (
            const char * fname )
```

Loads a list of firewall blocks from a file.

Loads a list of firewalls blocks from the given file and constructs the local blockchain.

Parameters

fname	the name of the file containing the chain.

Returns

whether the chain was sucecssfully loaded. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.10 remove_pending_rule()

```
int remove_pending_rule (
            char * addr )
```

Removes a pending rule from the list.

Searches for a pending rule with the given address. If a matching rule is found, the rule is removed.

Parameters

addr	the address to remove.

Returns

whether the pending rule was removed. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.11 rotate_pending_rules()

```
int rotate_pending_rules (
            void  )
```

Rotates the pending firewall rules.

Rotates this host's list of pending firewall rules, such that the oldest rule is removed from the list, allowing a new block to be added.

Returns

whether the list was rotated. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.1.2.12 save_blocks_to_file()

```
int save_blocks_to_file (
            const char * fname )
```

Saves the current loaded blockchain into a file.

Saves all blocks currently loaded into the blockchain.

Parameters

fname	the name of the file to save the blockchain.

Returns

whether the blockchain was successfully saved. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2 src/blockchain.h File Reference

Functions for creating and validating blockchains.

```
#include "firewall.h"
#include <openssl/sha.h>
#include <arpa/inet.h>
```

Data Structures

- struct FirewallBlock

Macros

- #define SHA256_STRING_LENGTH 64

  The length of SHA256 string representations.

Typedefs

- typedef struct FirewallBlock **FirewallBlock**

Functions

- int get_block_hash (unsigned char ∗buffer, FirewallBlock ∗block, int buffer_size)

  Calculates the SHA256 hash of a block.
- int get_hash_string (char ∗buffer, unsigned char ∗hash, int buffer_size)

  Formats a SHA256 digest into human-readable string.
- int **get_hash_from_string** (unsigned char ∗buffer, char ∗hash_string, int buffer_size)
- int add_block_to_chain (FirewallBlock ∗block)

  Adds a new firewall block onto the chain.
- int rotate_pending_rules (void)

  Rotates the pending firewall rules.
- int add_pending_rule (char ∗addr)

  Adds a new rule to the list of pending rules.
- int is_pending (char ∗addr)

  Checks if the given address has a pending rule.
- int remove_pending_rule (char ∗addr)

Removes a pending rule from the list.

- int get_last_block (FirewallBlock *block)

  Returns a pointer to the last firewall block.

- int get_last_hash (unsigned char *buffer)

  Returns the hash of the last firewall block in the chain.

- int load_blocks_from_file (const char *fname)

  Loads a list of firewall blocks from a file.

- int save_blocks_to_file (const char *fname)

  Saves the current loaded blockchain into a file.

- int free_chain (void)

  Frees the currently loaded blockchain.

4.2.1 Detailed Description

Functions for creating and validating blockchains.

Author

Adam Bruce

Date

22 Mar 2021

4.2.2 Function Documentation

4.2.2.1 add_block_to_chain()

```
int add_block_to_chain (
            FirewallBlock * block )
```

Adds a new firewall block onto the chain.

Appends the new firewall block to the linked list of firewall block.

Parameters

block	the new block to add to the chain.

Returns

whether the block has been added to the chain. If an the block is is null or the block's memory could not be allocated, the return value will be 1, otherwise the return value will be 0.

4.2.2.2 add_pending_rule()

```
int add_pending_rule (
            char * addr )
```

Adds a new rule to the list of pending rules.

Appends a new rule to the list of pending rules, this involves rotating the list, and adding the new rule's author.

Parameters

addr	the author of the new pending rule.

Returns

whether the rule was added. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.3 free_chain()

```
int free_chain (
            void  )
```

Frees the currently loaded blockchain.

Frees the memory currently allocated to blocks on the chain.

Returns

whether the chain was successfully freed. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.4 get_block_hash()

```
int get_block_hash (
            unsigned char * buffer,
            FirewallBlock * block,
            int buffer_size )
```

Calculates the SHA256 hash of a block.

Calculates the SHA256 hash of a block, storing the digest in the given buffer. This buffer should have a size of SHA256_DIGEST_LENGTH.

Parameters

buffer	the buffer to store the digest in.
block	a pointer to the block to hash.
buffer_size	the size of the buffer to store the hash in.

Returns

whether the hash has been calculated successfully. If any parameters are invalid, the return value will be 1, otherwise the return value will be 0.

4.2.2.5 get_hash_string()

```
int get_hash_string (
            char * buffer,
            unsigned char * hash,
            int buffer_size )
```

Formats a SHA256 digest into human-readable string.

Formats a SHA256 digest into a human-readable string, storing the result into the given buffer. This buffer should have a size of SHA256_STRING_LENGTH.

Parameters

buffer	the buffer to store the string in.
hash	the hash digest to format into a string.
buffer_size	the size of the buffer to store the string in.

Returns

whether the string has been formatted succesfully. If any parameters are invalid, the return value will be 1, otherwise the return value will be 0.

4.2.2.6 get_last_block()

```
int get_last_block (
            FirewallBlock * block )
```

Returns a pointer to the last firewall block.

Returns a pointer to the last firewall block in the chain.

Parameters

block	pointer to point to the last block.

Returns

whether the last block was successfully found. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.7 get_last_hash()

```
int get_last_hash (
            unsigned char * buffer )
```

Returns the hash of the last firewall block in the chain.

Gets the SHA256 hash of the last firewall block in the chain. If the chain is empty, the buffer will be empty.

Parameters

buffer	the buffer that the hash value will be copied into. This buffer should be at least SHA256_DIGEST_LENGTH bytes in size.

Returns

whether the hash value was copied successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.8 is_pending()

```
int is_pending (
            char * addr )
```

Checks if the given address has a pending rule.

Searches the pending rule list for the given address. If the address is found then the host has a pending rule.

Parameters

addr	the author to check for pending rules.

Returns

whether any pending rules for the author were found. If a pending rule is found, the return value will be 1, otherwise the return value will be 0.

4.2.2.9 load_blocks_from_file()

```
int load_blocks_from_file (
            const char * fname )
```

Loads a list of firewall blocks from a file.

Loads a list of firewalls blocks from the given file and constructs the local blockchain.

Parameters

fname	the name of the file containing the chain.

Returns

whether the chain was sucecssfully loaded. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.10 remove_pending_rule()

```
int remove_pending_rule (
            char * addr )
```

Removes a pending rule from the list.

Searches for a pending rule with the given address. If a matching rule is found, the rule is removed.

Parameters

addr	the address to remove.

Returns

whether the pending rule was removed. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.11 rotate_pending_rules()

```
int rotate_pending_rules (
            void  )
```

Rotates the pending firewall rules.

Rotates this host's list of pending firewall rules, such that the oldest rule is removed from the list, allowing a new block to be added.

Returns

whether the list was rotated. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.2.2.12 save_blocks_to_file()

```
int save_blocks_to_file (
            const char * fname )
```

Saves the current loaded blockchain into a file.

Saves all blocks currently loaded into the blockchain.

Parameters

fname	the name of the file to save the blockchain.

Returns

whether the blockchain was successfully saved. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.3 src/firewall.c File Reference

High level functions for handling firewall interactions.

```
#include "blockchain.h"
#include "firewall.h"
#include "ipc.h"
#include "net.h"
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <openssl/sha.h>
```

Macros

- #define **TIMEOUT** 500

Functions

- int recv_new_rule (FirewallRule *rule)

  The function called once a new firewall rule is available.
- int send_new_rule (FirewallRule *rule)

  The function used to send a new firewall rule.

4.3.1 Detailed Description

High level functions for handling firewall interactions.

Author

Adam Bruce

Date

22 Mar 2021

4.3.2 Function Documentation

4.3.2.1 recv_new_rule()

```
int recv_new_rule (
            FirewallRule * rule )
```

The function called once a new firewall rule is available.

This function is called once a firewall rule has been submitted by remote host, and the network has given consenus to the new firewall rule.

Parameters

rule	the new firewall rule that was received.

Returns

whether the corresponding IPC message to the OS was sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.3.2.2 send_new_rule()

```
int send_new_rule (
            FirewallRule * rule )
```

The function used to send a new firewall rule.

This function is called when a firewall rule is sent from the OS via IPC. The function will first attempt to gain consensus within the network, and if successfull, it will transmit the new rule to all known hosts.

Parameters

rule	the new firewall to send.

Returns

whether the firewall rule was sent. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.4 src/firewall.h File Reference

High level functions for handling firewall interactions.

```
#include <arpa/inet.h>
```

Data Structures

- struct FirewallRule

  The structure of a firewall rule.

Enumerations

- enum FirewallAction {
  ALLOW , BYPASS , DENY , FORCE_ALLOW ,
  LOG }

  All valid firewall rule actions.

Functions

- int recv_new_rule (FirewallRule *rule)

  The function called once a new firewall rule is available.
- int send_new_rule (FirewallRule *rule)

  The function used to send a new firewall rule.

4.4.1 Detailed Description

High level functions for handling firewall interactions.

Author

Adam Bruce

Date

22 Mar 2021

4.4.2 Enumeration Type Documentation

4.4.2.1 FirewallAction

`enum FirewallAction`

All valid firewall rule actions.

Enumerator

ALLOW	The connection should be allowed
BYPASS	The connection should be bypassed
DENY	The connection should be denied
FORCE_ALLOW	The connection shoule be forcefully allowed
LOG	The connection should be logged

4.4.3 Function Documentation

4.4.3.1 recv_new_rule()

```
int recv_new_rule (
            FirewallRule * rule )
```

The function called once a new firewall rule is available.

This function is called once a firewall rule has been submitted by remote host, and the network has given consenus to the new firewall rule.

Parameters

rule	the new firewall rule that was received.

Returns

whether the corresponding IPC message to the OS was sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.4.3.2 send_new_rule()

```
int send_new_rule (
            FirewallRule * rule )
```

The function used to send a new firewall rule.

This function is called when a firewall rule is sent from the OS via IPC. The function will first attempt to gain consensus within the network, and if successfull, it will transmit the new rule to all known hosts.

Parameters

rule	the new firewall to send.

Returns

whether the firewall rule was sent. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.5 src/ipc.c File Reference

Inter-process Communication interface.

```
#include "ipc.h"
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

Functions

- int init_ipc_server (void)

  Initialise the IPC in server mode.
- int init_ipc_client (void)

  Initialise the IPC in client mode.
- int cleanup_ipc (void)

  Cleans up the IPC session.
- int send_ipc_message (IPCMessage *message)

  Send and IPC message to a client application.
- int recv_ipc_message (IPCMessage *message)

  Retrieves an IPC message.

4.5.1 Detailed Description

Inter-process Communication interface.

Author

Adam Bruce

Date

22 Mar 2021

4.5.2 Function Documentation

4.5.2.1 cleanup_ipc()

```
int cleanup_ipc (
            void  )
```

Cleans up the IPC session.

Terminates the connection to the IPC session, and tears down the underlying session.

Returns

whether the connection was successfully terminated. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.5.2.2 init_ipc_client()

```
int init_ipc_client (
            void  )
```

Initialise the IPC in client mode.

Connects to a previously established IPC server.

Returns

whether the connection was succesfully established. If an error has occurred the return value will be 1, otherwise the return value will be 0.

4.5.2.3 init_ipc_server()

```
int init_ipc_server (
            void  )
```

Initialise the IPC in server mode.

Initialises the underlying IPC mechanism, and creates a new connection. If on ∗nix this is achieved using the POSIX message queue, or Named Pipes if on windows.

Returns

whether the IPC was successfully initialised, and a connection esatblished. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.5.2.4 recv_ipc_message()

```
int recv_ipc_message (
            IPCMessage * message )
```

Retrieves an IPC message.

Checks for an IPC message waiting in the queue. If a message is found, it is copied into the message parameter.

Parameters

message	the message that a waiting message will be copied into.

Returns

whether an IPC message has been copied from the queue. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.5.2.5 send_ipc_message()

```
int send_ipc_message (
            IPCMessage * message )
```

Send and IPC message to a client application.

Sends an IPC message to a client connected via IPC.

Parameters

message	the message to send.

Returns

whether the message was sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.6 src/ipc.h File Reference

Inter-process Communication interface.

```
#include "firewall.h"
#include <fcntl.h>
#include <sys/stat.h>
#include <arpa/inet.h>
```

Data Structures

- struct IPCMessage

  The structure of a IPC message.

Enumerations

- enum IPCMessageType {
  I_RULE , I_ENABLE , I_DISABLE , I_SHUTDOWN ,
  O_RULE }

  All valid IPC message types.

Functions

- int init_ipc_server (void)

  Initialise the IPC in server mode.
- int init_ipc_client (void)

  Initialise the IPC in client mode.
- int cleanup_ipc (void)

  Cleans up the IPC session.
- int send_ipc_message (IPCMessage *message)

  Send and IPC message to a client application.
- int recv_ipc_message (IPCMessage *message)

  Retrieves an IPC message.

4.6.1 Detailed Description

Inter-process Communication interface.

Author

Adam Bruce

Date

22 Mar 2021

4.6.2 Enumeration Type Documentation

4.6.2.1 IPCMessageType

`enum IPCMessageType`

All valid IPC message types.

Enumerator

I_RULE	New rule
I_ENABLE	Enable network communication
I_DISABLE	Disable network communication
I_SHUTDOWN	Shutdown the framework
O_RULE	New rule from another node

4.6.3 Function Documentation

4.6.3.1 cleanup_ipc()

```
int cleanup_ipc (
            void  )
```

Cleans up the IPC session.

Terminates the connection to the IPC session, and tears down the underlying session.

Returns

whether the connection was successfully terminated. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.6.3.2 init_ipc_client()

```
int init_ipc_client (
            void  )
```

Initialise the IPC in client mode.

Connects to a previously established IPC server.

Returns

whether the connection was succesfully established. If an error has occurred the return value will be 1, otherwise the return value will be 0.

4.6.3.3 init_ipc_server()

```
int init_ipc_server (
            void  )
```

Initialise the IPC in server mode.

Initialises the underlying IPC mechanism, and creates a new connection. If on ∗nix this is achieved using the POSIX message queue, or Named Pipes if on windows.

Returns

whether the IPC was successfully initialised, and a connection esatblished. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.6.3.4 recv_ipc_message()

```
int recv_ipc_message (
            IPCMessage * message )
```

Retrieves an IPC message.

Checks for an IPC message waiting in the queue. If a message is found, it is copied into the message parameter.

Parameters

message	the message that a waiting message will be copied into.

Returns

whether an IPC message has been copied from the queue. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.6.3.5 send_ipc_message()

```
int send_ipc_message (
            IPCMessage * message )
```

Send and IPC message to a client application.

Sends an IPC message to a client connected via IPC.

Parameters

message	the message to send.

Returns

whether the message was sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.7 src/main.c File Reference

Entry point for the application.

```
#include "firewall.h"
#include "net.h"
#include "ipc.h"
#include "blockchain.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
```

Functions

- void * recv_thread_func (void *data)

  Receiving thread.
- int **main** (int argc, char **argv)

4.7.1 Detailed Description

Entry point for the application.

Author

Adam Bruce

Date

22 Mar 2021

4.7.2 Function Documentation

4.7.2.1 recv_thread_func()

```
void* recv_thread_func (
            void * data )
```

Receiving thread.

This function is automatically run on the second thread, receiving and processing data from the network.

4.8 src/net.c File Reference

Network and protocol interface.

```
#include "net.h"
#include "blockchain.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <openssl/sha.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
```

Macros

- #define ETH_PREFIX_LEN 5

  The maximum length of a ∗nix ethernet adapter prefix.

Functions

- int get_local_address (char *buffer)

  Retrieves the local address of the host's Ethernet adapter.
- int get_acks (void)

  Returns the current number of acknowledgements.
- int reset_acks (void)

  Resets the number of acknowledgements.
- int set_ack (char *addr)

  Sets the acknowledgement state of a host.
- int load_hosts_from_file (const char *fname)

  Loads a list of hosts from a file.
- int save_hosts_to_file (const char *fname)

  Saves all known hosts currently loaded into a file.
- int add_host (char *addr)

  Adds a host to the host list.
- int check_host_exists (char *addr)

  Checks if a given host exists in the host list.
- int get_host_count (void)

  Returns the number of remote hosts known by the local host.
- int init_net (void)

  Initialises the network API.
- int cleanup_net (void)

  Uninitialises the network API.
- int send_to_host (char *ip_address, void *message, size_t length)

  Sends a message to a remote host.
- int send_advertisement_message (AdvertisementMessage *message)

  Sends an advertisement message.
- int send_to_all_advertisement_message (AdvertisementMessage *message)

  Sends an advertisement message to all known hosts.
- int recv_advertisement_message (void *buffer)

  Parses a received raw advertisement message.
- int recv_advertisement_broadcast (AdvertisementMessage *message)

  Handles advertisement broadcasts.
- int recv_advertisement_ack (AdvertisementMessage *message)

  Handles advertisement acknowledgements.
- int send_consensus_message (ConsensusMessage *message)

  Sends a consensus message.
- int send_to_all_consensus_message (ConsensusMessage *message)

  Sends a consensus message to all known hosts.
- int recv_consensus_message (void *buffer)

  Parses a received raw consensus message.
- int recv_consensus_broadcast (ConsensusMessage *message)

  Handles consensus broadcasts.
- int recv_consensus_ack (ConsensusMessage *message)

  Handles consensus acknowledgements.
- int send_rule_message (RuleMessage *message)

  Sends a firewall rule message.
- int send_to_all_rule_message (RuleMessage *message)

  Sends a firewall rule message to all known hosts.
- int recv_rule_message (void *buffer)

Parses a received raw firewall rule message.

- int recv_rule_broadcast (RuleMessage *message)

  Handles firewall rule broadcasts.

- int poll_message (void *buffer, size_t length)

  Waits for a message to be received.

4.8.1 Detailed Description

Network and protocol interface.

Author

Adam Bruce

Date

22 Mar 2021

4.8.2 Function Documentation

4.8.2.1 add_host()

```
int add_host (
            char * addr )
```

Adds a host to the host list.

Appends the given host to the list of hosts.

Parameters

addr	the address of the new host.

Returns

whether the host was appended successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.2 check_host_exists()

```
int check_host_exists (
            char * addr )
```

Checks if a given host exists in the host list.

Searches the list of hosts for the given address.

Parameters

addr	the address to search for.

Returns

whether the host was found. If the host was found, the return value will be 1, otherwise the return value will be 0.

4.8.2.3 cleanup_net()

```
int cleanup_net (
            void  )
```

Uninitialises the network API.

Uninitialises the network API by closing the underlying sockets and cleaning up the relevant socket API.

Returns

whether the network API was successfully cleaned up. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.8.2.4 get_acks()

```
int get_acks (
            void  )
```

Returns the current number of acknowledgements.

Returns the current number of acknowledgements this host has received since sending it's consensus message.

Returns

The number of acknowledgements.

4.8.2.5 get_host_count()

```
int get_host_count (
            void  )
```

Returns the number of remote hosts known by the local host.

Counts how many hosts are known locally.

Returns

the number of hosts.

4.8.2.6 get_local_address()

```
int get_local_address (
             char * buffer )
```

Retrieves the local address of the host's Ethernet adapter.

Retrieves the local address of the host's Ethernet adapter using the network API of the OS.

Parameters

buffer	the buffer to copy the address into.

Returns

whether the address was succesfully obtained. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.7 init_net()

```
int init_net (
              void  )
```

Initialises the network API.

Initialises the network API by initialising the underlying socket API and creating the necessary sockets for sending and receiving messages.

Returns

the status of the network API. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.8.2.8 load_hosts_from_file()

```
int load_hosts_from_file (
              const char * fname )
```

Loads a list of hosts from a file.

Loads a list of hosts from the given file into the HostList struct.

Parameters

fname	the name of the file containing the hosts.

Returns

whether the list of hosts was successfully loaded. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.9 poll_message()

```
int poll_message (
            void * buffer,
            size_t length )
```

Waits for a message to be received.

Waits for a message to be recieved. Once received, <length> bytes will be copied into the given buffer.

Parameters

buffer	the buffer to copy the message into.
length	the number of bytes to read.

Returns

the number of bytes received. If an error has occurred, a negative value will be returned.

4.8.2.10 recv_advertisement_ack()

```
int recv_advertisement_ack (
            AdvertisementMessage * message )
```

Handles advertisement acknowledgements.

Handles advertisement acknowledgement messages. Upon receiving an ack, if the host is not known, then thay are appended to the host list.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.11 recv_advertisement_broadcast()

```
int recv_advertisement_broadcast (
            AdvertisementMessage * message )
```

Handles advertisement broadcasts.

Handles advertisement broadcast messages. If the host is not known, then they are appended to the host list. Additionally, if the hop count has not exceeded the hop limit, it is forwarded to all known hosts.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.12 recv_advertisement_message()

```
int recv_advertisement_message (
            void * buffer )
```

Parses a received raw advertisement message.

Parses raw memory into an instance of an AdvertisementMessage. Upon identifying the message subtype, either recv_advertisement_broadcast or recv_advertisement_ack is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the advertisement message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.13 recv_consensus_ack()

```
int recv_consensus_ack (
            ConsensusMessage * message )
```

Handles consensus acknowledgements.

Handles consensus acknowledgement messages. Upon receiving an ack, the ack_count is incremented.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.14 recv_consensus_broadcast()

```
int recv_consensus_broadcast (
            ConsensusMessage * message )
```

Handles consensus broadcasts.

Handles consensus broadcast messages. If the host is known, and the consensus hash matches the host's last hash, then an ack is sent. Additionally, if the hop count has not exceeded the hop limit, the broadcast is forwarded to all known hosts.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.15 recv_consensus_message()

```
int recv_consensus_message (
            void * buffer )
```

Parses a received raw consensus message.

Parses raw memory into an instance of an ConsensusMessage. Upon identifying the message subtype, either recv_consensus_broadcast or recv_consensus_ack is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the consensus message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.16 recv_rule_broadcast()

```
int recv_rule_broadcast (
            RuleMessage * message )
```

Handles firewall rule broadcasts.

Handles firewall rule messages. If the host is known, and the host has sent a consensus ack, then the firewall rule is accepted and appended to the chain.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.17 recv_rule_message()

```
int recv_rule_message (
            void * buffer )
```

Parses a received raw firewall rule message.

Parses raw memory into an instance of an RuleMessage. Upon identifying the message subtype, recv_rule_↩ broadcast is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the firewall rule message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.18 reset_acks()

```
int reset_acks (
            void  )
```

Resets the number of acknowledgements.

Sets the ack state of each host to 0.

Returns

whether the acknowledgements were succesfully reset. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.19 save_hosts_to_file()

```
int save_hosts_to_file (
            const char * fname )
```

Saves all known hosts currently loaded into a file.

Saves all hosts currently stored in the HostList struct into a file.

Parameters

fname	the name of the file to save the hosts.

Returns

whether the list of hosts was successfully saved. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.20 send_advertisement_message()

```
int send_advertisement_message (
            AdvertisementMessage * message )
```

Sends an advertisement message.

Sends an advertisement to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.8.2.21 send_consensus_message()

```
int send_consensus_message (
            ConsensusMessage * message )
```

Sends a consensus message.

Sends a consensus message to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.8.2.22 send_rule_message()

```
int send_rule_message (
            RuleMessage * message )
```

Sends a firewall rule message.

Sends a firewall rule message to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.8.2.23 send_to_all_advertisement_message()

```
int send_to_all_advertisement_message (
            AdvertisementMessage * message )
```

Sends an advertisement message to all known hosts.

Sends an advertisement message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.24 send_to_all_consensus_message()

```
int send_to_all_consensus_message (
            ConsensusMessage * message )
```

Sends a consensus message to all known hosts.

Sends a consensus message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.25 send_to_all_rule_message()

```
int send_to_all_rule_message (
            RuleMessage * message )
```

Sends a firewall rule message to all known hosts.

Sends a firewall rule message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.8.2.26 send_to_host()

```
int send_to_host (
            char * ip_address,
            void * message,
            size_t length )
```

Sends a message to a remote host.

Sends a message to the remote host specified by their IP address.

Parameters

ip_address	the remote host's IP address.
message	the message / data to send to the remote host.
length	the length of the message / data.

Returns

the number of bytes sent to the remote host. If an error has occurred, a negative value will be returned.

4.8.2.27 set_ack()

```
int set_ack (
            char * addr )
```

Sets the acknowledgement state of a host.

Sets the acknowledgement state of the given host to 1.

Parameters

addr	the address of the host who's acknowledgement should be set.

Returns

if the acknowledgement was set successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9 src/net.h File Reference

Network and protocol interface.

```
#include "socket.h"
#include "firewall.h"
#include <openssl/sha.h>
#include <sys/socket.h>
#include <netdb.h>
```

Data Structures

- struct HostList

  The structure to store all known hosts as a linked list.
- struct AdvertisementMessage

  The structure to store an advertisement message.
- struct ConsensusMessage

  The structure to store a consensus message.
- struct RuleMessage

  The structure of a firewall rule message.

Macros

- #define PORT_RECV 8070

  Port for receiving messages.
- #define PORT_SEND 8071

  Port for sending messages.
- #define MAX_HOPS 5

  The maximum number of network hops.

Typedefs

- typedef struct HostList **HostList**

Enumerations

- enum MessageType { ADVERTISEMENT , CONSENSUS , RULE }

  All available message types for network transactions.
- enum MessageSubType { BROADCAST , ACK }

  All available message subtypes for network transactions.

Functions

- int get_local_address (char ∗buffer)

  Retrieves the local address of the host's Ethernet adapter.
- int get_acks (void)

  Returns the current number of acknowledgements.
- int reset_acks (void)

  Resets the number of acknowledgements.
- int set_ack (char ∗addr)

  Sets the acknowledgement state of a host.
- int load_hosts_from_file (const char ∗fname)

  Loads a list of hosts from a file.
- int save_hosts_to_file (const char ∗fname)

  Saves all known hosts currently loaded into a file.
- int add_host (char ∗addr)

  Adds a host to the host list.
- int check_host_exists (char ∗addr)

  Checks if a given host exists in the host list.
- int get_host_count (void)

  Returns the number of remote hosts known by the local host.
- int init_net (void)

  Initialises the network API.
- int cleanup_net (void)

  Uninitialises the network API.
- int send_to_host (char ∗ip_address, void ∗message, size_t length)

  Sends a message to a remote host.
- int send_advertisement_message (AdvertisementMessage ∗message)

  Sends an advertisement message.
- int send_to_all_advertisement_message (AdvertisementMessage ∗message)

Sends an advertisement message to all known hosts.

- int recv_advertisement_message (void *buffer)

  Parses a received raw advertisement message.
- int recv_advertisement_broadcast (AdvertisementMessage *message)

  Handles advertisement broadcasts.
- int recv_advertisement_ack (AdvertisementMessage *message)

  Handles advertisement acknowledgements.
- int send_consensus_message (ConsensusMessage *message)

  Sends a consensus message.
- int send_to_all_consensus_message (ConsensusMessage *message)

  Sends a consensus message to all known hosts.
- int recv_consensus_message (void *buffer)

  Parses a received raw consensus message.
- int recv_consensus_broadcast (ConsensusMessage *message)

  Handles consensus broadcasts.
- int recv_consensus_ack (ConsensusMessage *message)

  Handles consensus acknowledgements.
- int send_rule_message (RuleMessage *message)

  Sends a firewall rule message.
- int send_to_all_rule_message (RuleMessage *message)

  Sends a firewall rule message to all known hosts.
- int recv_rule_message (void *buffer)

  Parses a received raw firewall rule message.
- int recv_rule_broadcast (RuleMessage *message)

  Handles firewall rule broadcasts.
- int poll_message (void *buffer, size_t length)

  Waits for a message to be received.

4.9.1 Detailed Description

Network and protocol interface.

Author

Adam Bruce

Date

22 Mar 2021

4.9.2 Enumeration Type Documentation

4.9.2.1 MessageSubType

`enum MessageSubType`

All available message subtypes for network transactions.

Enumerator

BROADCAST	Broadcast message
ACK	Acknowledgement message

4.9.2.2 MessageType

```
enum MessageType
```

All available message types for network transactions.

Enumerator

ADVERTISEMENT	Host advertisement message
CONSENSUS	Firewall transaction consensus message
RULE	Firewall transaction rule message

4.9.3 Function Documentation

4.9.3.1 add_host()

```
int add_host (
            char * addr )
```

Adds a host to the host list.

Appends the given host to the list of hosts.

Parameters

addr	the address of the new host.

Returns

whether the host was appended successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.2 check_host_exists()

```
int check_host_exists (
            char * addr )
```

Checks if a given host exists in the host list.

Searches the list of hosts for the given address.

Parameters

addr	the address to search for.

Returns

whether the host was found. If the host was found, the return value will be 1, otherwise the return value will be 0.

4.9.3.3 cleanup_net()

```
int cleanup_net (
            void  )
```

Uninitialises the network API.

Uninitialises the network API by closing the underlying sockets and cleaning up the relevant socket API.

Returns

whether the network API was successfully cleaned up. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.9.3.4 get_acks()

```
int get_acks (
            void  )
```

Returns the current number of acknowledgements.

Returns the current number of acknowledgements this host has received since sending it's consensus message.

Returns

The number of acknowledgements.

4.9.3.5 get_host_count()

```
int get_host_count (
            void  )
```

Returns the number of remote hosts known by the local host.

Counts how many hosts are known locally.

Returns

the number of hosts.

4.9.3.6 get_local_address()

```
int get_local_address (
            char * buffer )
```

Retrieves the local address of the host's Ethernet adapter.

Retrieves the local address of the host's Ethernet adapter using the network API of the OS.

Parameters

buffer	the buffer to copy the address into.

Returns

whether the address was succesfully obtained. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.7 init_net()

```
int init_net (
            void  )
```

Initialises the network API.

Initialises the network API by initialising the underlying socket API and creating the necessary sockets for sending and receiving messages.

Returns

the status of the network API. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.9.3.8 load_hosts_from_file()

```
int load_hosts_from_file (
            const char * fname )
```

Loads a list of hosts from a file.

Loads a list of hosts from the given file into the HostList struct.

Parameters

fname	the name of the file containing the hosts.

Returns

whether the list of hosts was successfully loaded. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.9 poll_message()

```
int poll_message (
            void * buffer,
            size_t length )
```

Waits for a message to be received.

Waits for a message to be recieved. Once received, <length> bytes will be copied into the given buffer.

Parameters

buffer	the buffer to copy the message into.
length	the number of bytes to read.

Returns

the number of bytes received. If an error has occurred, a negative value will be returned.

4.9.3.10 recv_advertisement_ack()

```
int recv_advertisement_ack (
            AdvertisementMessage * message )
```

Handles advertisement acknowledgements.

Handles advertisement acknowledgement messages. Upon receiving an ack, if the host is not known, then thay are appended to the host list.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.11 recv_advertisement_broadcast()

```
int recv_advertisement_broadcast (
            AdvertisementMessage * message )
```

Handles advertisement broadcasts.

Handles advertisement broadcast messages. If the host is not known, then they are appended to the host list. Additionally, if the hop count has not exceeded the hop limit, it is forwarded to all known hosts.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.12 recv_advertisement_message()

```
int recv_advertisement_message (
            void * buffer )
```

Parses a received raw advertisement message.

Parses raw memory into an instance of an AdvertisementMessage. Upon identifying the message subtype, either recv_advertisement_broadcast or recv_advertisement_ack is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the advertisement message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.13 recv_consensus_ack()

```
int recv_consensus_ack (
            ConsensusMessage * message )
```

Handles consensus acknowledgements.

Handles consensus acknowledgement messages. Upon receiving an ack, the ack_count is incremented.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.14 recv_consensus_broadcast()

```
int recv_consensus_broadcast (
            ConsensusMessage * message )
```

Handles consensus broadcasts.

Handles consensus broadcast messages. If the host is known, and the consensus hash matches the host's last hash, then an ack is sent. Additionally, if the hop count has not exceeded the hop limit, the broadcast is forwarded to all known hosts.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.15 recv_consensus_message()

```
int recv_consensus_message (
            void * buffer )
```

Parses a received raw consensus message.

Parses raw memory into an instance of an ConsensusMessage. Upon identifying the message subtype, either recv_consensus_broadcast or recv_consensus_ack is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the consensus message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.16 recv_rule_broadcast()

```
int recv_rule_broadcast (
            RuleMessage * message )
```

Handles firewall rule broadcasts.

Handles firewall rule messages. If the host is known, and the host has sent a consensus ack, then the firewall rule is accepted and appended to the chain.

Parameters

message	the received message.

Returns

whether the message was handled correctly. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.17 recv_rule_message()

```
int recv_rule_message (
            void * buffer )
```

Parses a received raw firewall rule message.

Parses raw memory into an instance of an RuleMessage. Upon identifying the message subtype, recv_rule_↩ broadcast is called.

Parameters

buffer	the raw memory of the message.

Returns

whether the firewall rule message was parsed successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.18 reset_acks()

```
int reset_acks (
            void  )
```

Resets the number of acknowledgements.

Sets the ack state of each host to 0.

Returns

whether the acknowledgements were succesfully reset. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.19 save_hosts_to_file()

```
int save_hosts_to_file (
            const char * fname )
```

Saves all known hosts currently loaded into a file.

Saves all hosts currently stored in the HostList struct into a file.

Parameters

fname	the name of the file to save the hosts.

Returns

whether the list of hosts was successfully saved. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.20 send_advertisement_message()

```
int send_advertisement_message (
            AdvertisementMessage * message )
```

Sends an advertisement message.

Sends an advertisement to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.9.3.21 send_consensus_message()

```
int send_consensus_message (
            ConsensusMessage * message )
```

Sends a consensus message.

Sends a consensus message to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.9.3.22 send_rule_message()

```
int send_rule_message (
            RuleMessage * message )
```

Sends a firewall rule message.

Sends a firewall rule message to a remote host using the address information within the message.

Parameters

message	the message to send.

Returns

the number of bytes sent. If an error has occurred, the return value will be negative.

4.9.3.23 send_to_all_advertisement_message()

```
int send_to_all_advertisement_message (
            AdvertisementMessage * message )
```

Sends an advertisement message to all known hosts.

Sends an advertisement message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.24 send_to_all_consensus_message()

```
int send_to_all_consensus_message (
            ConsensusMessage * message )
```

Sends a consensus message to all known hosts.

Sends a consensus message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.25 send_to_all_rule_message()

```
int send_to_all_rule_message (
            RuleMessage * message )
```

Sends a firewall rule message to all known hosts.

Sends a firewall rule message to all known hosts. The address within the given message will be modified.

Parameters

message	the message to send.

Returns

whether all messages were sent successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.9.3.26 send_to_host()

```
int send_to_host (
            char * ip_address,
            void * message,
            size_t length )
```

Sends a message to a remote host.

Sends a message to the remote host specified by their IP address.

Parameters

ip_address	the remote host's IP address.
message	the message / data to send to the remote host.
length	the length of the message / data.

Returns

the number of bytes sent to the remote host. If an error has occurred, a negative value will be returned.

4.9.3.27 set_ack()

```
int set_ack (
            char * addr )
```

Sets the acknowledgement state of a host.

Sets the acknowledgement state of the given host to 1.

Parameters

addr	the address of the host who's acknowledgement should be set.

Returns

if the acknowledgement was set successfully. If an error has occurred, the return value will be 1, otherwise the return value will be 0.

4.10 src/socket.c File Reference

Cross-pflatform socket interface.

```
#include "socket.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
```

Functions

- int init_sockets (void)

  Initialises the socket API.
- int cleanup_sockets (void)

  Uninitialises the socket API.
- socket_t create_socket (void)

  Creates a new socket.
- void close_socket (socket_t sock)

  Closes a socket.
- int bind_socket (socket_t sock, int port)

  Binds a socket to a port.
- int send_to_socket (socket_t sock, void ∗message, size_t length, int flags, struct sockaddr_in dest_addr)

  Sends a message to a remote socket.
- int recv_from_socket (socket_t sock, void ∗buffer, size_t length, int flags)

  Receives a message from a socket.

4.10.1 Detailed Description

Cross-pflatform socket interface.

Author

Adam Bruce

Date

15 Dec 2020

4.10.2 Function Documentation

4.10.2.1 bind_socket()

```
int bind_socket (
            socket_t sock,
            int port )
```

Binds a socket to a port.

Binds the socket to a port, and configures it to use IP and UDP.

Parameters

sock	the socket to bind.
port	the port to bind the socket to.

Returns

whether the socket was successfully binded. If an error has occurred, the return value will be -1, otherwise the return value will be 0.

4.10.2.2 cleanup_sockets()

```
int cleanup_sockets (
            void  )
```

Uninitialises the socket API.

Uninitialises the relevent socket APIs for each operating system. For the NT kernel, this involves uninitialising Winsock. For UNIX systems, this function does nothing.

Returns

whether the API was succesfully cleaned up. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.10.2.3 close_socket()

```
void close_socket (
            socket_t sock )
```

Closes a socket.

Closes the socket using the relevant API for the operating system.

Parameters

sock	the socket to close.

4.10.2.4 create_socket()

```
socket_t create_socket (
            void  )
```

Creates a new socket.

Creates a UDP socket using the relevant API for the operating system.

Returns

a new socket descriptor, or 0 if a socket could not be created.

4.10.2.5 init_sockets()

```
int init_sockets (
            void  )
```

Initialises the socket API.

Initialises the relevent socket APIs for each operating system. For the NT kernel, this involves initialising Winsock. For UNIX systems, this function does nothing.

Returns

the status of the socket API. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.10.2.6 recv_from_socket()

```
int recv_from_socket (
            socket_t sock,
            void * buffer,
            size_t length,
            int flags )
```

Receives a message from a socket.

Receives a message from a socket.

Parameters

sock	the socket.
buffer	the buffer to read the message into.
length	the number of bytes to read.
flags	the flags used to configure the recv operation.

Returns

how many bytes were successfully read. If an error has occurred, a negative value will be returned.

4.10.2.7 send_to_socket()

```
int send_to_socket (
            socket_t sock,
            void * message,
            size_t length,
            int flags,
            struct sockaddr_in dest_addr )
```

Sends a message to a remote socket.

Sends the data stored within the buffer to a remote socket.

Parameters

sock	the local socket.
message	the data to send.
length	the length of the data.
flags	the flags used to configure the sendto operation.
dest_addr	the destination address

Returns

how many bytes were successfully sent. If an error has occurred, a negative value will be returned.

4.11 src/socket.h File Reference

Cross-platform socket interface.

```
#include <sys/socket.h>
#include <netinet/in.h>
```

Macros

- #define INVALID_SOCKET -1

  UNIX equivalent to WinSocks's INVALID_SOCKET constant.

Typedefs

- typedef int socket_t
  Cross platform socket type.

Functions

- int init_sockets (void)
  Initialises the socket API.
- int cleanup_sockets (void)
  Uninitialises the socket API.
- socket_t create_socket (void)
  Creates a new socket.
- void close_socket (socket_t sock)
  Closes a socket.
- int bind_socket (socket_t sock, int port)
  Binds a socket to a port.
- int send_to_socket (socket_t sock, void ∗message, size_t length, int flags, struct sockaddr_in dest_addr)
  Sends a message to a remote socket.
- int recv_from_socket (socket_t sock, void ∗buffer, size_t length, int flags)
  Receives a message from a socket.

4.11.1 Detailed Description

Cross-platform socket interface.

Author

Adam Bruce

Date

15 Dec 2020

4.11.2 Function Documentation

4.11.2.1 bind_socket()

```
int bind_socket (
            socket_t sock,
            int port )
```

Binds a socket to a port.

Binds the socket to a port, and configures it to use IP and UDP.

Parameters

sock	the socket to bind.
port	the port to bind the socket to.

Returns

whether the socket was successfully binded. If an error has occurred, the return value will be -1, otherwise the return value will be 0.

4.11.2.2 cleanup_sockets()

```
int cleanup_sockets (
            void  )
```

Uninitialises the socket API.

Uninitialises the relevent socket APIs for each operating system. For the NT kernel, this involves uninitialising Winsock. For UNIX systems, this function does nothing.

Returns

whether the API was succesfully cleaned up. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.11.2.3 close_socket()

```
void close_socket (
            socket_t sock )
```

Closes a socket.

Closes the socket using the relevant API for the operating system.

Parameters

sock	the socket to close.

4.11.2.4 create_socket()

```
socket_t create_socket (
            void  )
```

Creates a new socket.

Creates a UDP socket using the relevant API for the operating system.

Returns

a new socket descriptor, or 0 if a socket could not be created.

4.11.2.5 init_sockets()

```
int init_sockets (
            void  )
```

Initialises the socket API.

Initialises the relevent socket APIs for each operating system. For the NT kernel, this involves initialising Winsock. For UNIX systems, this function does nothing.

Returns

the status of the socket API. If an error has occurred, a non-zero value will be returned, otherwise the return value will be 0.

4.11.2.6 recv_from_socket()

```
int recv_from_socket (
            socket_t sock,
            void * buffer,
            size_t length,
            int flags )
```

Receives a message from a socket.

Receives a message from a socket.

Parameters

sock	the socket.
buffer	the buffer to read the message into.
length	the number of bytes to read.
flags	the flags used to configure the recv operation.

Returns

how many bytes were successfully read. If an error has occurred, a negative value will be returned.

4.11.2.7 send_to_socket()

```
int send_to_socket (
            socket_t sock,
            void * message,
            size_t length,
            int flags,
            struct sockaddr_in dest_addr )
```

Sends a message to a remote socket.

Sends the data stored within the buffer to a remote socket.

Parameters

sock	the local socket.
message	the data to send.
length	the length of the data.
flags	the flags used to configure the sendto operation.
dest_addr	the destination address

Returns

how many bytes were successfully sent. If an error has occurred, a negative value will be returned.

4.12 src/tests/blockchain_test.c File Reference

Tests for the functions in blockchain.c.

```
#include "../blockchain.h"
#include <openssl/sha.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
```

Functions

- void get_block_hash_null_buffer (void **state)
- void get_block_hash_buffer_too_small (void **state)
- void get_block_hash_null_block (void **state)
- void get_block_hash_valid (void **state)
- void get_hash_string_null_hash (void **state)
- void get_hash_string_buffer_too_small (void **state)
- void get_hash_string_null_buffer (void **state)
- void get_hash_string_valid (void **state)
- int **main** (void)

4.12.1 Detailed Description

Tests for the functions in blockchain.c.

Author

Adam Bruce

Date

03 Feb 2021

4.12.2 Function Documentation

4.12.2.1 get_block_hash_buffer_too_small()

```
void get_block_hash_buffer_too_small (
            void ** state )
```

Tests if get_block_hash returns a non-zero integer if the provided buffer is smaller than the hash digest length.

4.12.2.2 get_block_hash_null_block()

```
void get_block_hash_null_block (
            void ** state )
```

Tests if get_block_hash returns a non-zero integer if the pointer to block is NULL.

4.12.2.3 get_block_hash_null_buffer()

```
void get_block_hash_null_buffer (
            void ** state )
```

Tests if get_block_hash returns a non-zero integer if the pointer to buffer is NULL.

4.12.2.4 get_block_hash_valid()

```
void get_block_hash_valid (
            void ** state )
```

Tests if get_block_hash returns the correct hash digest for a valid block.

4.12.2.5 get_hash_string_buffer_too_small()

```
void get_hash_string_buffer_too_small (
            void ** state )
```

Tests if get_hash_string returns a non-zero integer if the buffer is smaller than the hash string.

4.12.2.6 get_hash_string_null_buffer()

```
void get_hash_string_null_buffer (
            void ** state )
```

Tests if get_hash_string returns a non-zero integer if the pointer to buffer is NULL.

4.12.2.7 get_hash_string_null_hash()

```
void get_hash_string_null_hash (
            void ** state )
```

Tests if get_hash_string returns a non-zero integer if the pointer to hash is NULL.

4.12.2.8 get_hash_string_valid()

```
void get_hash_string_valid (
            void ** state )
```

Tests if get__hash_string returns the correct hash for a valid digest.

4.13 src/tests/network_test.c File Reference

Tests the funtions declared in net.h.

```
#include "../net.h"
#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
```

Functions

- void init_net_valid (void **state)
- void cleanup_net_valid (void **state)
- void send_recv_valid (void **state)
- void **get_ip_address** (void **state)
- void **test_load_hosts_from_file** (void **state)
- void **test_add_host** (void **state)
- int **main** (void)

4.13.1 Detailed Description

Tests the funtions declared in net.h.

Author

Adam Bruce

Date

12 Feb 2021

4.13.2 Function Documentation

4.13.2.1 cleanup_net_valid()

```
void cleanup_net_valid (
            void ** state )
```

Tests if cleanup_net is able tp successfully cleanup the network API, including closing down the relevant sockets.

4.13.2.2 init_net_valid()

```
void init_net_valid (
            void ** state )
```

Tests if init_net is able to successfully initialise, and return a value to 0 to indicate success.

4.13.2.3 send_recv_valid()

```
void send_recv_valid (
            void ** state )
```

Tests if a message can be sent to another host using its IP address.

4.14 src/tests/socket_test.c File Reference

Tests the functions declared in socket.h.

```
#include "../socket.h"
#include <string.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include <arpa/inet.h>
```

Functions

- void init_sockets_valid (void **state)
- void cleanup_sockets_valid (void **state)
- void create_socket_valid (void **state)
- void bind_send_socket_valid (void **state)
- void bind_recv_socket_valid (void **state)
- void bind_socket_null_socket (void **state)
- void bind_socket_port_reuse (void **state)
- void send_recv_valid (void **state)
- int **main** (void)

4.14.1 Detailed Description

Tests the functions declared in socket.h.

Author

Adam Bruce

Date

12 Feb 2021

4.14.2 Function Documentation

4.14.2.1 bind_recv_socket_valid()

```
void bind_recv_socket_valid (
            void ** state )
```

Tests if a socket can be bound to the receiving port.

4.14.2.2 bind_send_socket_valid()

```
void bind_send_socket_valid (
            void ** state )
```

Tests if a socket can be bound to the sending port.

4.14.2.3 bind_socket_null_socket()

```
void bind_socket_null_socket (
            void ** state )
```

Tests to ensure a non-zero value is returned when attempting to bind to a non-existant socket.

4.14.2.4 bind_socket_port_reuse()

```
void bind_socket_port_reuse (
            void ** state )
```

Tests to ensure a non-zero value is returned when attempting to bind to a port that is already in use.

4.14.2.5 cleanup_sockets_valid()

```
void cleanup_sockets_valid (
            void ** state )
```

Tests if cleanup_sockets is able to successfully cleanup the relevant socket API, and return a value of 0 to indicate success.

4.14.2.6 create_socket_valid()

```
void create_socket_valid (
            void ** state )
```

Tests if a socket can be successfully created.

4.14.2.7 init_sockets_valid()

```
void init_sockets_valid (
            void ** state )
```

Tests if init_sockets is able to succesfully initialise, and return a value of 0 to indicate success.

4.14.2.8 send_recv_valid()

```
void send_recv_valid (
            void ** state )
```

Tests if a sending and receiving socket can be opened, and succesfully communicate between the two sockets.

Index