Using Blockchain to Create a Decentralised Security Model for Distributed Systems

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

April 2021

Abstract

//TODO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 COVID-19 and Cyberattacks

In the summer of 2020 during the midst of the COVID-19 pandemic, universities and research institutions worldwide were working hard to understand the structure of the virus and develop a vaccine in an attempt to return to normality. However, whereas some countries were making fast progress in understanding the virus, others were falling behind, and the virus began to put a strain on healthcare, and increasing critique on governments. In order to keep up with the nations at the forefront of vaccine development, nations turned to state-sponsored cyberattacks 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 [3].

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 [4]. 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 [5].

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 [6, 7]. The attack crippled both Newcastle and Northumbria Universities, however the attackers only managed to exfiltrate data from Newcastle University. Why was the attack successful on both occasions? Why wasn't knowledge of the attack shared?

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

1.2 Distributed Systems

A distributed system is defined by Tanenbaum and van Steen as a "collection of independent computers that appears to its users as a single coherent system" [1]. Such systems are commonplace in peer-to-peer computing and sensor networks where each systems contributes some data via transactions to the system. A distributed system therefore should be autonomous and to the user, should appear as though they are interacting with a single system. Furthermore users and applications should be able to interact with the distributed system in a consistent and uniform way, regardless of where and when system interaction takes place. This requires a common interface provided by a stub which is used to bridge the gap between a programming language or protocol and the distributed system. This stub hides the differences in machine architecture and communication between the computer and the distributed system. The use of stubs creates a new software layer, known as middleware which runs on an Operating System (OS) and exposes distributed functions to higher-level applications and users.

1.3 Decentralised Systems

Reed defines a decentralised computer system as a computer system that "involves separation of the computers in the system by physical distance, by boundaries of administrative responsibility for individual computers and their applications, and by firewalls" [8]. Reed suggests that for a computer system to be decentralised, it must be separated by both physical distance and administrative responsibility, such that no single body administrates the system. One of the most well-known examples of decentralisation is cryptocurrency, a currency which takes no physical form, but instead exists entirely digitally. If cryptocurrency were to be governed by a central body, nefarious transactions could be used to launder money. Using a decentralised system ensures the transaction can only take place if all nodes within the system are in consensus that the transaction is genuine.

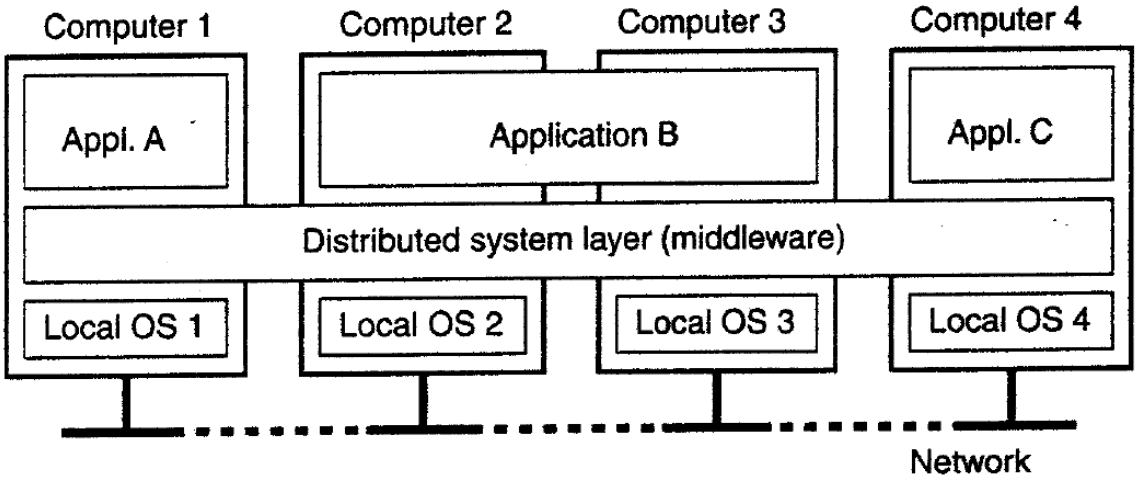

Figure 1.1: A Distributed System visualised as middleware [1]

1.4 Blockchain

Nofer et al. define blockchains as "data sets which are composed of a chain of data packages (blocks) where a block comprises multiple transactions. The blockchain is extended by each additional block and hence represents a complete ledger of the transaction history." [2]. Nofer et al. describe the basic fundamentals of a blockchain, which is that numerous blocks of transactions contribute to a larger chain. This chain is never controlled by a single body, instead a copy of the chain is stored at each node within a system, making blockchain a popular candidate for controlling transactions over a decentralised computer system. Hence, blockchain is the foundation for the vast majority of cryptocurrencies including Bitcoin[9] and Ethereum[10]. One of the key aspects of blockchain is the use of cryptographic hashing algorithms, these algorithms represent a block as a fixed-length string. For a block to be added to the chain, it must contain the hash of the previous block.

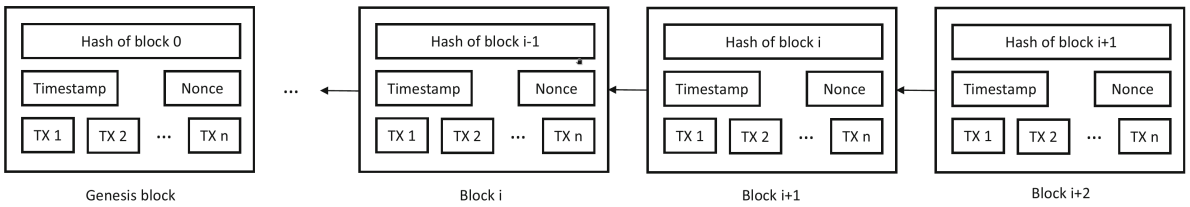

Figure 1.2: An example of a blockchain [2]

Chapter 2

Project Aims and Objectives

2.1 Aim

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

2.2 Objectives

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

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

Chapter 3

Background

3.1 Distributed Systems

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

3.1.1 Architecture

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

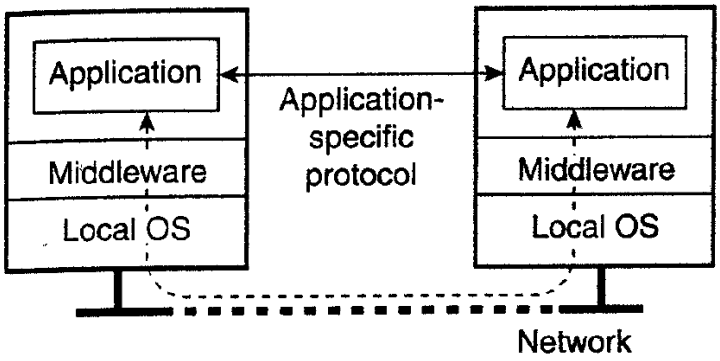

Figure 3.1: Application layer protocol running over middleware [1]

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

3.1.2 Remote Procedure Calls (RPC)

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

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

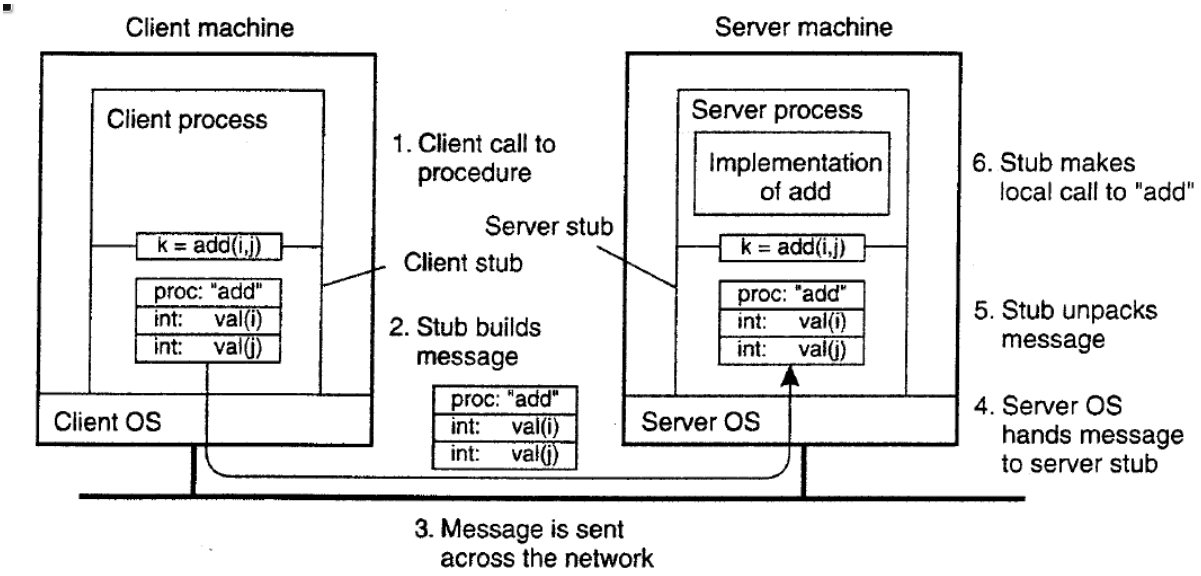

Figure 3.2: A breakdown of an RPC [1]

3.2 Decentralised Systems

3.3 Blockchain

The primary reference used for blockchain was Nofer et al.'s "Blockchain"[2].

3.4 Distributed Security

3.5 Firewall Rules

3.6 Fault Tolerance

3.7 Inter-Process Communication (IPC)

Glossary

blockchain A growing list of records, called blocks, that are linked using cryptography. 2, 6, 7

cryptocurrency A digital currency produces by a public network. 5

hashing The practise of taking data and representing that data as a fixed-length string. 6

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. 4

middleware Software that functions at an intermediate layer between applications and the operating system to provide distributed functions. 2, 5, 8, 9

phishing Sending an email that falsely claims to be from a legitimate organisation, usually combined with a threat or request for information. 4

smishing Sending a text message via SMS that falsely claims to be from a legitimate organisation, usually containing a link to a malicious website. 4

stub A piece of code that is used to marshal parameters for transmission across the network. 5, 7–9

Acronyms

APT Advanced Persistent Threat. 4

NCSC National Cyber Security Center. 4

OS Operating System. 5, 9

RPC Remote Procedure Call. 2, 9, 10

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] 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.

[4] P. Ozili and T. Arun, "Spillover of COVID-19: Impact on the global economy." `https://mpra.ub.uni-muenchen.de/99850/1/MPRA_paper_99850.pdf`, 2020.

[5] H. S. Lallie, L. A. Shepherd, J. R. C. Nurse, A. Erola, G. Epiphaniou, C. Maple, and X. Bellekens, "Cyber security in the age of COVID-19: A timeline and analysis of cyber-crime and cyber-attacks during the pandemic." `https://arxiv.org/pdf/2006.11929.pdf`, 2020.

[6] 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.

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

[8] D. P. Reed, "Naming and synchronisation in a decentralized computer system." `https://dspace.mit.edu/bitstream/handle/1721.1/16279/05331643-MIT.pdf`, 1978.

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

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

Appendix A

Framework Source Code

Listing A.1: main.c

```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 */
#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
    }
  return NULL;
}
```

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

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

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 the hash of the last firewall block in the chain.
141  *
142  * Gets the SHA256 hash of the last firewall block in the chain. If the chain
143  * is empty, the buffer will be empty.
144  * @param buffer the buffer that the hash value will be copied into. This
145  * buffer should be at least SHA256_DIGEST_LENGTH bytes in size.
146  * @return whether the hash value was copied successfully, If an error has
147  * occurred, the return value will be 1, otherwuse the return value will be 0.
148  */
149 int get_last_hash(unsigned char *buffer);
150
151 /**
152  * @brief Loads a list of firewall blocks from a file.
153  *
154  * Loads a list of firewalls blocks from the given file and constructs the
155  * local blockchain.
156  * @param fname the name of the file containing the chain.
157  * @return whether the chain was sucecssfully loaded. If an error has occurred,
158  * the return value will be 1, otherwise the return value will be 0.
159  */
160 int load_blocks_from_file(const char *fname);
161
162 /**
163  * @brief Saves the current loaded blockchain into a file.
164  *
165  * Saves all blocks currently loaded into the blockchain.
166  * @param fname the name of the file to save the blockchain.
167  * @return whether the blockchain was successfully saved. If an error has
168  * occurred, the return value will be 1, otherwise the return value will be 0.
169  */
170 int save_blocks_to_file(const char *fname);
171
172 /**
173  * @brief Frees the currently loaded blockchain.
174  *
175  * Frees the memory currently allocated to blocks on the chain.
176  * @return whether the chain was successfully freed. If an error has occurred,
177  * the return value will be 1, otherwise the return value will be 0.
178  */
179 int free_chain(void);
180
181 #endif
```

Listing A.3: blockchain.c

```
1 /**
2  * @file blockchain.c
3  * @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>
#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,
   (void*)&block->rule.source_addr, INET_ADDRSTRLEN);
  memcpy(data_to_hash + INET_ADDRSTRLEN * 2,
   (void*)&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;
    }

  for(i = 0; i < SHA256_DIGEST_LENGTH; i++)
    {
      sprintf(buffer + (i * 2), "%02x", hash[i]);
    }
  buffer[SHA256_STRING_LENGTH] = '\0';
  return 0;
}

int get_hash_from_string(unsigned char *buffer, char *hash_string,
       int buffer_size)
{
  int i;
  uint16_t hex_val;
  char buf[3];
```

```
77
78    if(buffer_size < SHA256_DIGEST_LENGTH || !buffer || !hash_string)
79      {
80        return 1;
81      }
82
83    buf[2] = '\0';
84    for(i = 0; i < SHA256_DIGEST_LENGTH; i++)
85      {
86        memcpy(buf, hash_string + (i * 2), 2);
87        hex_val = strtol(buf, NULL, 16);
88        memcpy(buffer + i, &hex_val, 2);
89      }
90
91    return 0;
92  }
93
94  int add_block_to_chain(FirewallBlock *block)
95  {
96    FirewallBlock *fw_chain;
97    unsigned char hash[SHA256_DIGEST_LENGTH + 1];
98    char hash_string[SHA256_STRING_LENGTH + 1];
99
100   get_block_hash(hash, block, SHA256_DIGEST_LENGTH + 1);
101   get_hash_string(hash_string, hash, SHA256_STRING_LENGTH + 1);
102   hash_string[9] = '\0';
103
104   if(!chain)
105     {
106       chain = (FirewallBlock*)malloc(sizeof(FirewallBlock));
107       memset(chain, 0, sizeof(FirewallBlock));
108       memcpy(chain, block, sizeof(FirewallBlock));
109
110       printf("[ BLOC ] Added new block with hash %s...%s\n",
111        hash_string, hash_string + (SHA256_STRING_LENGTH - 10));
112       save_blocks_to_file("chain.txt");
113       return 0;
114     }
115
116   fw_chain = chain;
117   while(fw_chain && fw_chain->next)
118     {
119       fw_chain = fw_chain->next;
120     }
121
122   fw_chain->next = (FirewallBlock*)malloc(sizeof(FirewallBlock));
123   memset(fw_chain->next, 0, sizeof(FirewallBlock));
124   memcpy(fw_chain->next, block, sizeof(FirewallBlock));
125   printf("[ BLOC ] Added new block with hash %s...%s\n",
126    hash_string, hash_string + (SHA256_STRING_LENGTH - 10));
127
128   save_blocks_to_file("chain.txt");
129   return 0;
130
131 }
132
133 int rotate_pending_rules(void)
134 {
135   int index;
136
137   for(index = 0; index < PENDING_RULES_BUF_LEN - 2; index++)
138     {
139       strncpy(pending_rules[index], pending_rules[index + 1], INET_ADDRSTRLEN);
140     }
141   memset(pending_rules[0], '\0', INET_ADDRSTRLEN);
142
143   return 0;
144 }
145
146 int add_pending_rule(char *addr)
147 {
148   rotate_pending_rules();
149   strncpy(pending_rules[0], addr, INET_ADDRSTRLEN);
```

```
150   return 0;
151 }
152
153 int is_pending(char *addr)
154 {
155   int index;
156
157   for(index = 0; index < PENDING_RULES_BUF_LEN; index++)
158     {
159       if(strncmp(pending_rules[index], addr, INET_ADDRSTRLEN) == 0)
160   {
161     return 1;
162   }
163     }
164   return 0;
165 }
166
167 int remove_pending_rule(char *addr)
168 {
169   int index, match;
170
171   match = 0;
172   for(index = 0; index < PENDING_RULES_BUF_LEN; index++)
173     {
174       if(strncmp(pending_rules[index], addr, INET_ADDRSTRLEN) == 0)
175   {
176     match = 1;
177     break;
178   }
179     }
180
181   if(match)
182     {
183       for(; index > 0; index--)
184   {
185     strncpy(pending_rules[index], pending_rules[index - 1], INET_ADDRSTRLEN);
186   }
187       memset(pending_rules[0], '\0', INET_ADDRSTRLEN);
188     }
189
190   return 0;
191 }
192
193 int get_last_hash(unsigned char *buffer)
194 {
195   FirewallBlock *fw_chain;
196
197   if(!chain)
198     {
199       memset(buffer, '\0', SHA256_DIGEST_LENGTH);
200       return 0;
201     }
202
203   fw_chain = chain;
204   while(fw_chain && fw_chain->next)
205     {
206       fw_chain = fw_chain->next;
207     }
208
209   get_block_hash(buffer, fw_chain, SHA256_DIGEST_LENGTH);
210   return 0;
211 }
212
213 int load_blocks_from_file(const char *fname)
214 {
215   FILE *file;
216   FirewallBlock block;
217   char buffer[256], temp_buf[6], *next_delim;
218   int c;
219   size_t pos;
220
221   file = fopen(fname, "r");
222   if(!file)
```

```
223     {
224       printf("[ BLOC ] No block file found\n");
225       return 1;
226     }
227 
228   pos = 0;
229   while((c = fgetc(file)) != EOF)
230     {
231       if((char)c == '\r')
232   {
233     continue;
234   }
235 
236       if((char)c == '\n')
237   {
238     buffer[pos] = '\0';
239     memset(&block, 0, sizeof(FirewallBlock));
240 
241     /* Hash */
242     get_hash_from_string(block.last_hash, buffer, SHA256_DIGEST_LENGTH);
243     next_delim = strchr(buffer, ',');
244 
245     /* Author */
246     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
247     next_delim = strchr(buffer, ',');
248     memcpy(block.author, buffer, next_delim - buffer);
249 
250     /* Source address */
251     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
252     next_delim = strchr(buffer, ',');
253     memcpy(block.rule.source_addr, buffer, next_delim - buffer);
254 
255     /* Source port */
256     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
257     next_delim = strchr(buffer, ',');
258     memset(temp_buf, '\0', 6);
259     memcpy(temp_buf, buffer, next_delim - buffer);
260     block.rule.source_port = atoi(temp_buf);
261 
262     /* Destination address */
263     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
264     next_delim = strchr(buffer, ',');
265     memcpy(block.rule.dest_addr, buffer, next_delim - buffer);
266 
267     /* Destination port */
268     memcpy(buffer, next_delim + 1, (buffer + 256) - next_delim - 1);
269     next_delim = strchr(buffer, ',');
270     memset(temp_buf, '\0', 6);
271     memcpy(temp_buf, buffer, next_delim - buffer);
272     block.rule.dest_port = atoi(temp_buf);
273 
274     memcpy(buffer, next_delim + 1, (buffer + 265) - next_delim - 1);
275     if(strcmp("ALLOW", buffer) == 0)
276       {
277         block.rule.action = ALLOW;
278       }
279     else if(strcmp("DENY", buffer) == 0)
280       {
281         block.rule.action = DENY;
282       }
283     else if(strcmp("BYPASS", buffer) == 0)
284       {
285         block.rule.action = BYPASS;
286       }
287     else if(strcmp("FORCE_ALLOW", buffer) == 0)
288       {
289         block.rule.action = FORCE_ALLOW;
290       }
291     else
292       {
293         block.rule.action = LOG;
294       }
295 
```

```
296     add_block_to_chain(&block);
297
298     memset(buffer, '\0', 256);
299     pos = 0;
300     continue;
301   }
302       buffer[pos++] = (char)c;
303     }
304
305   fclose(file);
306   return 0;
307 }
308
309 int save_blocks_to_file(const char *fname)
310 {
311   FILE *file;
312   FirewallBlock *block;
313   char hash_string[SHA256_STRING_LENGTH + 1];
314
315   file = fopen(fname, "w+");
316   if(!file)
317     {
318       printf("[ ERR  ] Could not create block file\n");
319       return 1;
320     }
321
322   block = chain;
323   if(block)
324     {
325       while(block && strlen(block->author) > 0)
326   {
327     memset(hash_string, '\0', SHA256_STRING_LENGTH + 1);
328     get_hash_string(hash_string, block->last_hash,
329         SHA256_STRING_LENGTH + 1);
330     fwrite(hash_string, SHA256_STRING_LENGTH, 1, file);
331     fputc(',', file);
332     fwrite(block->author, strlen(block->author), 1, file);
333     fputc(',', file);
334     fwrite(block->rule.source_addr, strlen(block->rule.source_addr), 1,
335      file);
336     fputc(',', file);
337     fprintf(file, "%hd,", block->rule.source_port);
338     fwrite(block->rule.dest_addr, strlen(block->rule.dest_addr), 1,
339      file);
340     fputc(',', file);
341     fprintf(file, "%hd,", block->rule.dest_port);
342
343     switch(block->rule.action)
344       {
345       case ALLOW:
346         fputs("ALLOW", file);
347         break;
348       case DENY:
349         fputs("DENY", file);
350         break;
351       case BYPASS:
352         fputs("BYPASS", file);
353         break;
354       case FORCE_ALLOW:
355         fputs("FORCE_ALLOW", file);
356         break;
357       case LOG:
358         fputs("LOG", file);
359       }
360
361     fputc('\n', file);
362     block = block->next;
363   }
364     }
365
366   fclose(file);
367   return 0;
368 }
```

```
369
370 int free_chain(void)
371 {
372   FirewallBlock *block, *temp;
373   block = chain;
374
375   while(block)
376     {
377       temp = block;
378       block = block->next;
379       free(temp);
380     }
381
382   return 0;
383 }
```

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 /**
20  * @brief All valid firewall rule actions.
21  */
22 typedef enum
23   {
24     ALLOW,          /**< The connection should be allowed            */
25     BYPASS,         /**< The connection should be bypassed           */
26     DENY,           /**< The connection should be denied             */
27     FORCE_ALLOW,    /**< The connection shoule be forcefully allowed */
28     LOG             /**< The connection should be logged             */
29   } FirewallAction;
30
31 /**
32  * @brief The structure of a firewall rule.
33  */
34 typedef struct
35 {
36   char source_addr[INET_ADDRSTRLEN]; /**< The rule's source address      */
37   uint16_t source_port;              /**< The rule's source port         */
38   char dest_addr[INET_ADDRSTRLEN];   /**< The rule's destination address */
39   uint16_t dest_port;                /**< The rule's destination port    */
40   FirewallAction action;             /**< The rule's action              */
41 } FirewallRule;
42
43 /**
44  * @brief The function called once a new firewall rule is available.
45  *
46  * This function is called once a firewall rule has been submitted by remote
47  * host, and the network has given consenus to the new firewall rule.
48  * @param rule the new firewall rule that was received.
49  * @return whether the corresponding IPC message to the OS was sent
50  * successfully. If an error has occurred, the return value will be 1,
51  * otherwise the return value will be 0.
52  */
53 int recv_new_rule(FirewallRule *rule);
54
55 /**
```

```
 * @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>

#ifndef _WIN32
#include <unistd.h>
#endif

#include <openssl/sha.h>

#define TIMEOUT 500

static char local_address[INET_ADDRSTRLEN];

int recv_new_rule(FirewallRule *rule)
{
  IPCMessage msg;
  msg.message_type = I_RULE;
  memcpy(&msg.rule, rule, sizeof(FirewallRule));
  return send_ipc_message(&msg);
}

int send_new_rule(FirewallRule *rule)
{
  ConsensusMessage consensus_msg;
  RuleMessage rule_msg;
  FirewallBlock block;

  get_local_address(local_address);
  consensus_msg.type = CONSENSUS;
  consensus_msg.hops = 0;
  consensus_msg.consensus_type = BROADCAST;
  strncpy(consensus_msg.source_addr, local_address, INET_ADDRSTRLEN);
  strncpy(consensus_msg.target_addr, local_address, INET_ADDRSTRLEN);
  get_last_hash(consensus_msg.last_block_hash);
  send_to_all_consensus_message(&consensus_msg);
  printf("[ CONS ] Sent consensus message to %d known host(s)\n",
   get_host_count());

#ifdef _WIN32
  Sleep(TIMEOUT);
#else
  usleep(TIMEOUT * 1000);
#endif

  /* At least half known hosts have consensus */
  if(get_acks() < (get_host_count() + 1) / 2)
```

```
    {
      printf("[ CONS ] Consensus not achieved (%d/%d hosts)\n", get_acks(),
       get_host_count());
      reset_acks();
      return 1;
    }

  printf("[ CONS ] Consensus achieved (%d/%d hosts)\n", get_acks(),
   (get_host_count() + 1) / 2);
  reset_acks();

  rule_msg.type = RULE;
  rule_msg.hops = 0;
  rule_msg.rule_type = BROADCAST;
  strncpy(rule_msg.source_addr, local_address, INET_ADDRSTRLEN);
  strncpy(rule_msg.target_addr, local_address, INET_ADDRSTRLEN);
  memcpy(&rule_msg.rule, rule, sizeof(FirewallRule));
  send_to_all_rule_message(&rule_msg);
  printf("[ RULE ] Sent new rule message to %d known hosts(s)\n",
   get_host_count());

  memset(&block, 0, sizeof(FirewallBlock));
  get_last_hash(block.last_hash);
  strncpy(block.author, local_address, INET_ADDRSTRLEN);
  memcpy(&block.rule, rule, sizeof(FirewallRule));
  add_block_to_chain(&block);

  return 0;
}
```

Listing A.6: ipc.h

```
/**
 * @file ipc.h
 * @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        */
  } 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;
    }

  queue = mqueue;

  return 0;
}

int connect_ipc(void)
{
  if(!ConnectNamedPipe(queue, NULL))
    {
      CloseHandle(queue);
      return 1;
    }
  return 0;
}

int init_ipc_client(void)
{
  HANDLE mqueue;

  mqueue = CreateFile(TEXT("\\\\.\\pipe\\dfw"), PIPE_ACCESS_DUPLEX,
          PIPE_TYPE_MESSAGE, NULL, OPEN_EXISTING,
          FILE_ATTRIBUTE_NORMAL, NULL);

  if(mqueue == INVALID_HANDLE_VALUE)
    {
      return 1;
    }

  queue = mqueue;

  return 0;
}

int cleanup_ipc(void)
{
  return CloseHandle(queue);
}
```

```
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
16 #ifdef _WIN32
17 #include <inttypes.h>
18 #include <winsock2.h>
19 #include <ws2tcpip.h>
20 #else
21 #include <sys/socket.h>
22 #include <netdb.h>
23 #endif
24
25 /**
26  * @brief Port for receiving messages
27  */
28 #define PORT_RECV 8070
29
30 /**
31  * @brief Port for sending messages
32  */
33 #define PORT_SEND 8071
34
35 /**
36  * @brief The maximum number of network advertisement hops.
37  */
38 #define MAX_ADVERTISEMENT_HOPS 5
39
40 /**
41  * @brief The maximum number of hops before a message is destroyed.
42  */
43 #define MAX_CONSENSUS_HOPS 5
44
45 /**
46  * @brief The local IP address.
47  */
48 static char local_address[INET_ADDRSTRLEN];
49
50 /**
51  * @brief All available message types for network transactions.
52  */
53 typedef enum
54   {
```

```
55      ADVERTISEMENT,  /**< Host advertisement message             */
56      CONSENSUS,      /**< Firewall transaction consensus message */
57      RULE            /**< Firewall transaction rule message      */
58    } MessageType;
59
60  /**
61   * @brief All available message subtypes for network transactions.
62   */
63  typedef enum
64    {
65      BROADCAST,      /**< Broadcast message       */
66      ACK             /**< Acknowledgement message */
67    } MessageSubType;
68
69  /**
70   * @brief The structure to store all known hosts as a linked list.
71   */
72  struct HostList
73  {
74    struct HostList *next;       /**< The next host in the list   */
75    char addr[INET_ADDRSTRLEN];  /**< The host's address          */
76    uint8_t ack;                 /**< The ack status for the host */
77  };
78  typedef struct HostList HostList;
79
80  /**
81   * @brief The structure to store an advertisement message.
82   */
83  typedef struct
84  {
85    MessageType type;                   /**< The message type (ADVERTISEMENT)  */
86    uint8_t hops;                       /**< The hop count                     */
87    MessageSubType advertisement_type; /**< The message subtype               */
88    char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
89    char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
90    char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
91  } AdvertisementMessage;
92
93  /**
94   * @brief The structure to store a consensus message.
95   */
96  typedef struct
97  {
98    MessageType type;                   /**< The message type (CONSENSUS)      */
99    uint8_t hops;                       /**< The hop count                     */
100   MessageSubType consensus_type;      /**< The message subtype               */
101   char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
102   char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
103   char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
104   unsigned char last_block_hash[SHA256_DIGEST_LENGTH]; /**< The hash of the
105                   last block      */
106 } ConsensusMessage;
107
108 /**
109  * @brief The structure of a firewall rule message.
110  */
111 typedef struct
112 {
113   MessageType type;                   /**< The message type (RULE)           */
114   uint8_t hops;                       /**< The hop count                     */
115   MessageSubType rule_type;           /**< The message subtype               */
116   char source_addr[INET_ADDRSTRLEN]; /**< The source address of the message */
117   char target_addr[INET_ADDRSTRLEN]; /**< The target address of the message */
118   char next_addr[INET_ADDRSTRLEN];   /**< The next address of the message   */
119   FirewallRule rule;                  /**< The firewall rule                 */
120 } RuleMessage;
121
122 /**
123  * @brief Retrieves the local address of the host's Ethernet adapter.
124  *
125  * Retrieves the local address of the host's Ethernet adapter using the network
126  * API of the OS.
127  * @param buffer the buffer to copy the address into.
```

```
128  * @return whether the address was succesfully obtained. If an error has
129  * occurred, the return value will be 1, otherwise the return value will be 0.
130  */
131 int get_local_address(char *buffer);
132
133 /**
134  * @brief Returns the current number of acknowledgements.
135  *
136  * Returns the current number of acknowledgements this host has received since
137  * sending it's consensus message.
138  * @return The number of acknowledgements.
139  */
140 int get_acks(void);
141
142 /**
143  * @brief Resets the number of acknowledgements.
144  *
145  * Sets the ack state of each host to 0.
146  * @return whether the acknowledgements were succesfully reset. If an error has
147  * occurred, the return value will be 1, otherwise the return value will be 0.
148  */
149 int reset_acks(void);
150
151 /**
152  * @brief Sets the acknowledgement state of a host.
153  *
154  * Sets the acknowledgement state of the given host to 1.
155  * @param addr the address of the host who's acknowledgement should be set.
156  * @return if the acknowledgement was set successfully. If an error has
157  * occurred, the return value will be 1, otherwise the return value will be 0.
158  */
159 int set_ack(char *addr);
160
161 /**
162  * @brief Loads a list of hosts from a file.
163  *
164  * Loads a list of hosts from the given file into the HostList struct.
165  * @param fname the name of the file containing the hosts.
166  * @return whether the list of hosts was successfully loaded. If an error has
167  * occurred, the return value will be 1, otherwise the return value will be 0.
168  */
169 int load_hosts_from_file(const char *fname);
170
171 /**
172  * @brief Saves all known hosts currently loaded into a file.
173  *
174  * Saves all hosts currently stored in the HostList struct into a file.
175  * @param fname the name of the file to save the hosts.
176  * @return whether the list of hosts was successfully saved. If an error has
177  * occurred, the return value will be 1, otherwise the return value will be 0.
178  */
179 int save_hosts_to_file(const char *fname);
180
181 /**
182  * @brief Adds a host to the host list.
183  *
184  * Appends the given host to the list of hosts.
185  * @param addr the address of the new host.
186  * @return whether the host was appended successfully. If an error has
187  * occurred, the return value will be 1, otherwise the return value will be 0.
188  */
189 int add_host(char* addr);
190
191 /**
192 * @brief Checks if a given host exists in the host list.
193 *
194 * Searches the list of hosts for the given address.
195 * @param addr the address to search for.
196 * @return whether the host was found. If the host was found, the return value
197 * will be 1, otherwise the return value will be 0.
198 */
199 int check_host_exists(char *addr);
200
```

```
201 /**
202  * @brief Returns the number of remote hosts known by the local host.
203  *
204  * Counts how many hosts are known locally.
205  * @return the number of hosts.
206  */
207 int get_host_count(void);
208
209 /**
210  * @brief Initialises the network API.
211  *
212  * Initialises the network API by initialising the underlying socket API and
213  * creating the necessary sockets for sending and receiving messages.
214  * @return the status of the network API. If an error has occurred, a non-zero
215  * value will be returned, otherwise the return value will be 0.
216  */
217 int init_net(void);
218
219 /**
220  * @brief Uninitialises the network API.
221  *
222  * Uninitialises the network API by closing the underlying sockets and cleaning
223  * up the relevant socket API.
224  * @return whether the network API was successfully cleaned up. If an error has
225  * occurred, a non-zero value will be returned, otherwise the return value will
226  * be 0.
227  */
228 int cleanup_net(void);
229
230 /**
231  * @brief Sends a message to a remote host.
232  *
233  * Sends a message to the remote host specified by their IP address.
234  * @param ip_address the remote host's IP address.
235  * @param message the message / data to send to the remote host.
236  * @param length the length of the message / data.
237  * @return the number of bytes sent to the remote host. If an error has
238  * occurred, a negative value will be returned.
239  */
240 int send_to_host(char *ip_address, void *message, size_t length);
241
242 /**
243  * @brief Sends an advertisement message.
244  *
245  * Sends an advertisement to a remote host using the address information within
246  * the message.
247  * @param message the message to send.
248  * @return the number of bytes sent. If an error has occurred, the return value
249  * will be negative.
250  */
251 int send_advertisement_message(AdvertisementMessage *message);
252
253 /**
254  * @brief Sends an advertisement message to all known hosts.
255  *
256  * Sends an advertisement message to all known hosts. The address within the
257  * given message will be modified.
258  * @param message the message to send.
259  * @return whether all messages were sent successfully. If an error has
260  * occurred, the return value will be 1, otherwise the return value will be 0.
261  */
262 int send_to_all_advertisement_message(AdvertisementMessage *message);
263
264 /**
265  * @brief Parses a received raw advertisement message.
266  *
267  * Parses raw memory into an instance of an AdvertisementMessage. Upon
268  * identifying the message subtype, either recv_advertisement_broadcast or
269  * recv_advertisement_ack is called.
270  * @param buffer the raw memory of the message.
271  * @return whether the advertisement message was parsed successfully. If an
272  * error has occurred, the return value will be 1, otherwise the return value
273  * will be 0.
```

```
274  */
275 int recv_advertisement_message(void *buffer);
276
277 /**
278  * @brief Handles advertisement broadcasts.
279  *
280  * Handles advertisement broadcast messages. If the host is not known, then
281  * they are appended to the host list. Additionally, if the hop count has not
282  * exceeded the hop limit, it is forwarded to all known hosts.
283  * @param message the received message.
284  * @return whether the message was handled correctly. If an error has occurred,
285  * the return value will be 1, otherwise the return value will be 0.
286  */
287 int recv_advertisement_broadcast(AdvertisementMessage *message);
288
289 /**
290  * @brief Handles advertisement acknowledgements.
291  *
292  * Handles advertisement acknowledgement messages. Upon receiving an ack, if
293  * the host is not known, then thay are appended to the host list.
294  * @param message the received message.
295  * @return whether the message was handled correctly. If an error has occurred,
296  * the return value will be 1, otherwise the return value will be 0.
297  */
298 int recv_advertisement_ack(AdvertisementMessage *message);
299
300 /**
301  * @brief Sends a consensus message.
302  *
303  * Sends a consensus message to a remote host using the address information
304  * within the message.
305  * @param message the message to send.
306  * @return the number of bytes sent. If an error has occurred, the return value
307  * will be negative.
308  */
309 int send_consensus_message(ConsensusMessage *message);
310
311 /**
312  * @brief Sends a consensus message to all known hosts.
313  *
314  * Sends a consensus message to all known hosts. The address within the given
315  * message will be modified.
316  * @param message the message to send.
317  * @return whether all messages were sent successfully. If an error has
318  * occurred, the return value will be 1, otherwise the return value will be 0.
319  */
320 int send_to_all_consensus_message(ConsensusMessage *message);
321
322 /**
323  * @brief Parses a received raw consensus message.
324  *
325  * Parses raw memory into an instance of an ConsensusMessage. Upon
326  * identifying the message subtype, either recv_consensus_broadcast or
327  * recv_consensus_ack is called.
328  * @param buffer the raw memory of the message.
329  * @return whether the consensus message was parsed successfully. If an error
330  * has occurred, the return value will be 1, otherwise the return value will be
331  * 0.
332  */
333 int recv_consensus_message(void *buffer);
334
335 /**
336  * @brief Handles consensus broadcasts.
337  *
338  * Handles consensus broadcast messages. If the host is known, and the
339  * consensus hash matches the host's last hash, then an ack is sent.
340  * Additionally, if the hop count has not exceeded the hop limit, the broadcast
341  * is forwarded to all known hosts.
342  * @param message the received message.
343  * @return whether the message was handled correctly. If an error has occurred,
344  * the return value will be 1, otherwise the return value will be 0.
345  */
346 int recv_consensus_broadcast(ConsensusMessage *message);
```

```
347
348 /**
349  * @brief Handles consensus acknowledgements.
350  *
351  * Handles consensus acknowledgement messages. Upon receiving an ack, the
352  * ack_count is incremented.
353  * @param message the received message.
354  * @return whether the message was handled correctly. If an error has occurred,
355  * the return value will be 1, otherwise the return value will be 0.
356  */
357 int recv_consensus_ack(ConsensusMessage *message);
358
359 /**
360  * @brief Sends a firewall rule message.
361  *
362  * Sends a firewall rule message to a remote host using the address information
363  * within the message.
364  * @param message the message to send.
365  * @return the number of bytes sent. If an error has occurred, the return value
366  * will be negative.
367  */
368 int send_rule_message(RuleMessage *message);
369
370 /**
371  * @brief Sends a firewall rule message to all known hosts.
372  *
373  * Sends a firewall rule message to all known hosts. The address within the
374  * given message will be modified.
375  * @param message the message to send.
376  * @return whether all messages were sent successfully. If an error has
377  * occurred, the return value will be 1, otherwise the return value will be 0.
378  */
379 int send_to_all_rule_message(RuleMessage *message);
380
381 /**
382  * @brief Parses a received raw firewall rule message.
383  *
384  * Parses raw memory into an instance of an RuleMessage. Upon identifying the
385  * message subtype, recv_rule_broadcast is called.
386  * @param buffer the raw memory of the message.
387  * @return whether the firewall rule message was parsed successfully. If an
388  * error has occurred, the return value will be 1, otherwise the return value
389  * will be 0.
390  */
391 int recv_rule_message(void *buffer);
392
393 /**
394  * @brief Handles firewall rule broadcasts.
395  *
396  * Handles firewall rule messages. If the host is known, and the host has sent
397  * a consensus ack, then the firewall rule is accepted and appended to the
398  * chain.
399  * @param message the received message.
400  * @return whether the message was handled correctly. If an error has occurred,
401  * the return value will be 1, otherwise the return value will be 0.
402  */
403 int recv_rule_broadcast(RuleMessage *message);
404
405 /**
406  * @brief Waits for a message to be received.
407  *
408  * Waits for a message to be recieved. Once received, <length> bytes will be
409  * copied into the given buffer.
410  * @param buffer the buffer to copy the message into.
411  * @param length the number of bytes to read.
412  * @return the number of bytes received. If an error has occurred, a negative
413  * value will be returned.
414  */
415 int poll_message(void *buffer, size_t length);
416
417 #endif
```

Listing A.9: net.c

```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>
#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.
 */
static socket_t socket_send;

/**
 * @brief The local receiving socket.
 */
static socket_t socket_recv;

/**
 * @brief The list of known hosts.
 */
static HostList *host_list;

#ifndef _WIN32
/**
 * @brief Known ethernet adapter prefixes.
 *
 * Used for obtaining the assigned ethernet address.
 */
static char eth_prefixes[6][ETH_PREFIX_LEN + 1] = {
  "eth", "em", "ed", "genet", "usmsc", "\0"
};
#endif


#ifdef _WIN32
#define ADAPTER_NAME_LEN 8
int get_local_address(char* buffer)
{
  DWORD rv, size;
  PIP_ADAPTER_ADDRESSES adapter_addresses, aa;
  PIP_ADAPTER_UNICAST_ADDRESS ua;
  char address[INET_ADDRSTRLEN], name[ADAPTER_NAME_LEN];

  rv = GetAdaptersAddresses(AF_INET, GAA_FLAG_INCLUDE_PREFIX, NULL, NULL,
          &size);
  if (rv != ERROR_BUFFER_OVERFLOW)
```

```
73      {
74        return 1;
75      }
76    adapter_addresses = (PIP_ADAPTER_ADDRESSES)malloc(size);
77
78    rv = GetAdaptersAddresses(AF_INET, GAA_FLAG_INCLUDE_PREFIX, NULL,
79            adapter_addresses, &size);
80    if (rv != ERROR_SUCCESS)
81      {
82        free(adapter_addresses);
83        return 1;
84      }
85
86    for (aa = adapter_addresses; aa != NULL; aa = aa->Next)
87      {
88        memset(name, '\0', ADAPTER_NAME_LEN);
89        WideCharToMultiByte(CP_ACP, 0, aa->FriendlyName,
90          wcslen(aa->FriendlyName), name, ADAPTER_NAME_LEN,
91          NULL, NULL);
92
93        if(strncmp(name, "Ethernet", ADAPTER_NAME_LEN) == 0)
94    {
95      for (ua = aa->FirstUnicastAddress; ua != NULL; ua = ua->Next)
96        {
97          memset(address, '\0', INET_ADDRSTRLEN);
98          getnameinfo(ua->Address.lpSockaddr, ua->Address.iSockaddrLength,
99          address, INET_ADDRSTRLEN, NULL, 0, NI_NUMERICHOST);
100
101         strncpy(buffer, address, INET_ADDRSTRLEN);
102         free(adapter_addresses);
103         return 0;
104       }
105   }
106     }
107
108   free(adapter_addresses);
109   return 1;
110 }
111 #else
112 int get_local_address(char *buffer)
113 {
114   struct ifaddrs *interfaces = NULL, *addr = NULL;
115   void *addr_ptr = NULL;
116   char addr_str[INET_ADDRSTRLEN];
117   int prefix_index, match;
118
119   if(getifaddrs(&interfaces) != 0)
120     {
121       return 1;
122     }
123
124   match = 0;
125   for(addr = interfaces; addr != NULL; addr = addr->ifa_next)
126     {
127       if(addr->ifa_addr->sa_family == AF_INET)
128   {
129     prefix_index = 0;
130     match = 0;
131     while(eth_prefixes[prefix_index][0] != '\0')
132       {
133         if(strstr(addr->ifa_name, eth_prefixes[prefix_index]))
134     {
135       match = 1;
136       break;
137     }
138         prefix_index++;
139       }
140   }
141
142       if(match)
143   {
144     addr_ptr = &((struct sockaddr_in*)addr->ifa_addr)->sin_addr;
145     inet_ntop(addr->ifa_addr->sa_family,
```

```
146         addr_ptr,
147         addr_str,
148         sizeof(addr_str));
149
150     strcpy(buffer, addr_str);
151     break;
152   }
153     }
154   freeifaddrs(interfaces);
155   return !match;
156 }
157 #endif
158
159 int get_acks(void)
160 {
161   HostList *host;
162   int count;
163
164   host = host_list;
165   if(!host || strlen(host->addr) == 0)
166     {
167       return 0;
168     }
169
170   count = 0;
171   while(host)
172     {
173       if(host->ack > 0)
174   {
175     count++;
176   }
177       host = host->next;
178     }
179
180   return count;
181 }
182
183 int reset_acks(void)
184 {
185   HostList *host;
186
187   host = host_list;
188   if(!host || strlen(host->addr) == 0)
189     {
190       return 1;
191     }
192
193   while(host)
194     {
195       host->ack = 0;
196       host = host->next;
197     }
198
199   return 0;
200 }
201
202 int set_ack(char *addr)
203 {
204   HostList *host;
205
206   host = host_list;
207   if(!host || strlen(host->addr) == 0)
208     {
209       return 1;
210     }
211
212   while(host)
213     {
214       if(strncmp(host->addr, addr, INET_ADDRSTRLEN) == 0)
215   {
216     host->ack = 1;
217     return 0;
218   }
```

```
219       host = host->next;
220     }
221
222   return 1;
223 }
224
225 int load_hosts_from_file(const char *fname)
226 {
227   FILE *file;
228   int c;
229   size_t pos;
230   char buffer[INET_ADDRSTRLEN + 10];
231
232   if(!host_list)
233     {
234       printf("[ ERR  ] Network stack not yet initialised\n");
235       return 1;
236     }
237
238   file = fopen(fname, "r");
239   if(!file)
240     {
241       printf("[ NET  ] No hostfile found\n");
242       return 1;
243     }
244
245   pos = 0;
246   while((c = fgetc(file)) != EOF)
247     {
248       if((char)c == '\r')
249   {
250     continue;
251   }
252
253       if((char)c == '\n')
254   {
255     add_host(buffer);
256     memset(buffer, '\0', INET_ADDRSTRLEN + 10);
257     pos = 0;
258     continue;
259   }
260       buffer[pos++] = (char)c;
261     }
262
263   fclose(file);
264   return 0;
265 }
266
267 int save_hosts_to_file(const char *fname)
268 {
269   FILE *file;
270   HostList *host;
271
272   if(!host_list)
273     {
274       printf("[ ERR  ] Network stack not yet initialised\n");
275       return 1;
276     }
277
278   file = fopen(fname, "w+");
279   if(!file)
280     {
281       printf("[ ERR  ] Could not create hostfile\n");
282       return 1;
283     }
284
285   host = host_list;
286   if(host)
287     {
288       while(host && strlen(host->addr) > 0)
289   {
290     fwrite(host->addr, strlen(host->addr), 1, file);
291     fputc('\n', file);
```

```
292       host = host->next;
293     }
294       }
295 
296     fclose(file);
297     return 0;
298 }
299 
300 int add_host(char *addr)
301 {
302   HostList *host;
303 
304   printf("[ NET  ] Adding new host (%s)\n", addr);
305 
306   if(!host_list)
307     {
308       return 1;
309     }
310   if(!addr)
311     {
312       return 1;
313     }
314 
315   host = host_list;
316   if(strlen(host->addr) == 0)
317     {
318       strncpy(host->addr, addr, INET_ADDRSTRLEN);
319     }
320   else
321     {
322       while(host && host->next)
323   {
324     host = host->next;
325   }
326 
327       host->next = (HostList*)malloc(sizeof(HostList));
328       memset(host->next, 0, sizeof(HostList));
329       strncpy(host->next->addr, addr, INET_ADDRSTRLEN);
330     }
331   return save_hosts_to_file("hosts.txt");
332 }
333 
334 
335 
336 int check_host_exists(char *addr)
337 {
338   HostList *host;
339 
340   host = host_list;
341   if(!host)
342     {
343       printf("[ ERR  ] Network stack not yet initialised\n");
344       return 0;
345     }
346   if(!addr)
347     {
348       return 0;
349     }
350 
351   while(host)
352     {
353       if(strncmp(addr, host->addr, INET_ADDRSTRLEN) == 0)
354   {
355     return 1;
356   }
357       host = host->next;
358     }
359 
360   return 0;
361 }
362 
363 int get_host_count(void)
364 {
```

```
365   HostList *host;
366   int count;
367
368   host = host_list;
369   if(!host || strlen(host->addr) == 0)
370     {
371       return 0;
372     }
373
374   count = 0;
375   while(host)
376     {
377       count++;
378       host = host->next;
379     }
380
381   return count;
382 }
383
384 int init_net(void)
385 {
386   if(init_sockets() != 0)
387     {
388       return 1;
389     }
390   if((socket_send = create_socket()) == 0)
391     {
392       return 1;
393     }
394   if(bind_socket(socket_send, PORT_SEND) != 0)
395     {
396       return 1;
397     }
398   if((socket_recv = create_socket()) == 0)
399     {
400       return 1;
401     }
402
403   if(bind_socket(socket_recv, PORT_RECV) != 0)
404     {
405       return 1;
406     }
407
408   if(get_local_address(local_address) != 0)
409     {
410       return 1;
411     }
412
413   host_list = (HostList*)malloc(sizeof(HostList));
414   if(!host_list)
415     {
416       return 1;
417     }
418   memset(host_list, 0, sizeof(HostList));
419
420   return 0;
421 }
422
423 int cleanup_net(void)
424 {
425   HostList *host, *temp;
426   host = host_list;
427
428   while(host)
429     {
430       temp = host;
431       host = host->next;
432       free(temp);
433     }
434
435   close_socket(socket_send);
436   close_socket(socket_recv);
437   return cleanup_sockets();
```

```
438 }
439
440 int send_to_host(char *ip_address, void *message, size_t length)
441 {
442   struct sockaddr_in remote_addr;
443
444   remote_addr.sin_family = AF_INET;
445   remote_addr.sin_addr.s_addr = inet_addr(ip_address);
446   remote_addr.sin_port = htons(PORT_RECV);
447
448   return send_to_socket(socket_send, message, length, 0, remote_addr);
449 }
450
451 int send_advertisement_message(AdvertisementMessage *message)
452 {
453   char buffer[11];
454   struct in_addr addr;
455   int status;
456
457   buffer[0] = message->type;
458   buffer[1] = message->hops;
459   buffer[2] = message->advertisement_type;
460
461   status = inet_pton(AF_INET, message->source_addr, &addr);
462   if(status == 0)
463     {
464       return 1;
465     }
466   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
467
468   status = inet_pton(AF_INET, message->next_addr, &addr);
469   if(status == 0)
470     {
471       return 1;
472     }
473   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
474
475   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
476 }
477
478 int send_to_all_advertisement_message(AdvertisementMessage *message)
479 {
480   HostList *host;
481   host = host_list;
482   while(host)
483     {
484       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
485       send_advertisement_message(message);
486       host = host->next;
487     }
488   return 0;
489 }
490
491 int recv_advertisement_message(void *buffer)
492 {
493   AdvertisementMessage message;
494   char *char_buffer;
495   struct sockaddr_in target, source;
496
497   char_buffer = (char*)buffer;
498   message.type = char_buffer[0];
499   message.hops = char_buffer[1];
500   message.advertisement_type = char_buffer[2];
501
502   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
503   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
504   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
505   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
506
507   if(strncmp(local_address, message.source_addr, INET_ADDRSTRLEN) == 0)
508     {
509       return 0;
510     }
```

```
511
512   switch(message.advertisement_type)
513     {
514     case BROADCAST:
515       recv_advertisement_broadcast(&message);
516       break;
517     case ACK:
518       recv_advertisement_ack(&message);
519       break;
520     }
521
522   return 0;
523 }
524
525 int recv_advertisement_broadcast(AdvertisementMessage *message)
526 {
527   AdvertisementMessage new_message;
528
529   if(!message)
530     {
531       return 1;
532     }
533
534   /* If new host, add */
535   if(!check_host_exists(message->source_addr))
536     {
537       add_host(message->source_addr);
538       new_message.type = ADVERTISEMENT;
539       new_message.hops = 0;
540       new_message.advertisement_type = ACK;
541       strncpy(new_message.source_addr, local_address, INET_ADDRSTRLEN);
542       strncpy(new_message.target_addr, message->source_addr, INET_ADDRSTRLEN);
543
544       /* Send ACK */
545       send_to_all_advertisement_message(&new_message);
546     }
547
548   /* If under max hop count, forward to all hosts */
549   if(message->hops < MAX_ADVERTISEMENT_HOPS)
550     {
551       memcpy(&new_message, message, sizeof(AdvertisementMessage));
552       new_message.hops++;
553       send_to_all_advertisement_message(&new_message);
554     }
555
556   return 0;
557 }
558
559 int recv_advertisement_ack(AdvertisementMessage* message)
560 {
561   HostList *host;
562
563   if(!message)
564     {
565       return 1;
566     }
567
568   /* Check if we are the intended recipient */
569   if(strncmp(local_address, message->target_addr, INET_ADDRSTRLEN) == 0
570      && !check_host_exists(message->source_addr))
571     {
572       add_host(message->source_addr);
573     }
574   else
575     {
576       if(message->hops < MAX_ADVERTISEMENT_HOPS)
577   {
578     host = host_list;
579     message->hops++;
580
581     while(host)
582       {
583         strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
```

```
584         send_advertisement_message(message);
585         host = host->next;
586       }
587   }
588     }
589
590   return 0;
591 }
592
593 int send_consensus_message(ConsensusMessage *message)
594 {
595   char buffer[11 + SHA256_DIGEST_LENGTH];
596   char hash_string[SHA256_STRING_LENGTH + 1];
597   struct in_addr addr;
598   int status;
599
600   buffer[0] = message->type;
601   buffer[1] = message->hops;
602   buffer[2] = message->consensus_type;
603
604   status = inet_pton(AF_INET, message->source_addr, &addr);
605   if(status == 0)
606     {
607       return 1;
608     }
609   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
610
611   status = inet_pton(AF_INET, message->target_addr, &addr);
612   if(status == 0)
613     {
614       return 1;
615     }
616   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
617
618
619   memcpy(buffer + 11, message->last_block_hash, SHA256_DIGEST_LENGTH);
620   get_hash_string(hash_string, message->last_block_hash,
621       SHA256_STRING_LENGTH + 1);
622
623   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
624 }
625
626 int send_to_all_consensus_message(ConsensusMessage *message)
627 {
628   HostList *host;
629   host = host_list;
630   while(host && strlen(host->addr) > 0)
631     {
632       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
633       send_consensus_message(message);
634       host = host->next;
635     }
636   return 0;
637 }
638
639 int recv_consensus_message(void *buffer)
640 {
641   ConsensusMessage message;
642   char *char_buffer;
643   struct sockaddr_in target, source;
644
645   char_buffer = (char*)buffer;
646   message.type = char_buffer[0];
647   message.hops = char_buffer[1];
648   message.consensus_type = char_buffer[2];
649
650   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
651   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
652   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
653   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
654
655   memcpy(message.last_block_hash, char_buffer + 11, SHA256_DIGEST_LENGTH);
656
```

```
657   switch(message.consensus_type)
658     {
659     case BROADCAST:
660       recv_consensus_broadcast(&message);
661       break;
662     case ACK:
663       recv_consensus_ack(&message);
664       break;
665     }
666
667   return 0;
668 }
669
670 int recv_consensus_broadcast(ConsensusMessage *message)
671 {
672   ConsensusMessage new_message;
673   unsigned char last_hash[SHA256_DIGEST_LENGTH];
674   char hash_string[SHA256_STRING_LENGTH + 1];
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   }
707
708       /* Send ACK */
709       send_to_all_consensus_message(&new_message);
710     }
711   else
712     {
713       printf("[ CONS ] Received consensus message with mismatched hash\n");
714     }
715
716   /* If under max hop count, forward to all hosts */
717   if(message->hops < MAX_CONSENSUS_HOPS)
718     {
719     message->hops++;
720     send_to_all_consensus_message(message);
721     }
722
723   return 0;
724 }
725
726 int recv_consensus_ack(ConsensusMessage *message)
727 {
728   HostList *host;
729
```

```
730   if(!message)
731     {
732       return 1;
733     }
734
735   /* Check if we are the intended recipient */
736   if(strncmp(local_address, message->target_addr, INET_ADDRSTRLEN) == 0)
737     {
738       set_ack(message->source_addr);
739     }
740   else
741     {
742       if(message->hops < MAX_CONSENSUS_HOPS)
743   {
744     host = host_list;
745     message->hops++;
746
747     while(host)
748       {
749         strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
750         send_consensus_message(message);
751         host = host->next;
752       }
753   }
754     }
755
756   return 0;
757 }
758
759 int send_rule_message(RuleMessage *message)
760 {
761   char buffer[11 + 4 + 4 + 2 + 2 + 4];
762   struct in_addr addr;
763   int status;
764
765   buffer[0] = message->type;
766   buffer[1] = message->hops;
767   buffer[2] = message->rule_type;
768
769   status = inet_pton(AF_INET, message->source_addr, &addr);
770   if(status == 0)
771     {
772       return 1;
773     }
774   memcpy(buffer + 3, &addr.s_addr, sizeof(addr.s_addr));
775
776   status = inet_pton(AF_INET, message->target_addr, &addr);
777   if(status == 0)
778     {
779       return 1;
780     }
781   memcpy(buffer + 7, &addr.s_addr, sizeof(addr.s_addr));
782
783   status = inet_pton(AF_INET, (void*)&message->rule.source_addr, &addr);
784   if(status == 0)
785     {
786       return 1;
787     }
788   memcpy(buffer + 11, &addr.s_addr, sizeof(addr.s_addr));
789
790   status = inet_pton(AF_INET, (void*)&message->rule.dest_addr, &addr);
791   if(status == 0)
792     {
793       return 1;
794     }
795   memcpy(buffer + 15, &addr.s_addr, sizeof(addr.s_addr));
796
797   memcpy(buffer + 19, (void*)&message->rule.source_port, 2);
798   memcpy(buffer + 21, (void*)&message->rule.dest_port, 2);
799   memcpy(buffer + 23, (void*)&message->rule.action, 4);
800
801   return send_to_host(message->next_addr, (void*)buffer, sizeof(buffer));
802 }
```

```
803
804 int send_to_all_rule_message(RuleMessage *message)
805 {
806   HostList *host;
807   host = host_list;
808   while(host)
809     {
810       strncpy(message->next_addr, host->addr, INET_ADDRSTRLEN);
811       send_rule_message(message);
812       host = host->next;
813     }
814   return 0;
815 }
816
817 int recv_rule_message(void *buffer)
818 {
819   RuleMessage message;
820   char *char_buffer;
821   struct sockaddr_in target, source, fw_source, fw_dest;
822
823   memset(&message, '\0', sizeof(RuleMessage));
824   char_buffer = (char*)buffer;
825   message.type = char_buffer[0];
826   message.hops = char_buffer[1];
827   message.rule_type = char_buffer[2];
828
829   source.sin_addr.s_addr = *(int*)(char_buffer + 3);
830   target.sin_addr.s_addr = *(int*)(char_buffer + 7);
831   inet_ntop(AF_INET, &source.sin_addr, message.source_addr, INET_ADDRSTRLEN);
832   inet_ntop(AF_INET, &target.sin_addr, message.target_addr, INET_ADDRSTRLEN);
833
834   fw_source.sin_addr.s_addr = *(int*)(char_buffer + 11);
835   fw_dest.sin_addr.s_addr = *(int*)(char_buffer + 15);
836   inet_ntop(AF_INET, &fw_source.sin_addr, message.rule.source_addr,
837       INET_ADDRSTRLEN);
838   inet_ntop(AF_INET, &fw_dest.sin_addr, message.rule.dest_addr,
839       INET_ADDRSTRLEN);
840
841   memcpy(&message.rule.source_port, (uint16_t*)(char_buffer + 19), 2);
842   memcpy(&message.rule.dest_port, (uint16_t*)(char_buffer + 21), 2);
843   memcpy(&message.rule.action, (int*)(char_buffer + 23), 4);
844
845   switch(message.rule_type)
846     {
847     case BROADCAST:
848       recv_rule_broadcast(&message);
849       break;
850     case ACK:
851       break;
852     }
853
854   return 0;
855 }
856
857 int recv_rule_broadcast(RuleMessage *message)
858 {
859   FirewallBlock new_block;
860   unsigned char last_hash[SHA256_DIGEST_LENGTH];
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       /* TODO: ADD recv_new_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_CONSENSUS_HOPS)
882     {
883       message->hops++;
884       send_to_all_rule_message(message);
885     }
886
887   return 0;
888 }
889
890 int poll_message(void *buffer, size_t length)
891 {
892   int bytes_read;
893
894   bytes_read = recv_from_socket(socket_recv, buffer, length, 0);
895   if(bytes_read <= 0)
896     {
897       return 0;
898     }
899
900   switch(((char*)buffer)[0])
901     {
902     case ADVERTISEMENT:
903       return recv_advertisement_message(buffer);
904       break;
905     case CONSENSUS:
906       return recv_consensus_message(buffer);
907       break;
908     case RULE:
909       return recv_rule_message(buffer);
910       break;
911     default:
912       return bytes_read;
913     }
914 }
```

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.
69  */
70 socket_t create_socket(void);
71
72 /**
73  * @brief Closes a socket.
74  *
75  * Closes the socket using the relevant API for the operating system.
76  * @param sock the socket to close.
77  */
78 void close_socket(socket_t sock);
79
80 /**
81  * @brief Binds a socket to a port.
82  *
83  * Binds the socket to a port, and configures it to use IP and UDP.
84  * @param sock the socket to bind.
85  * @param port the port to bind the socket to.
86  * @return whether the socket was successfully binded. If an error has
87  * occurred, the return value will be -1, otherwise the return value will
88  * be 0.
89  */
90 int bind_socket(socket_t sock, int port);
91
92 /**
93  * @brief Sends a message to a remote socket.
94  *
95  * Sends the data stored within the buffer to a remote socket.
96  * @param sock the local socket.
97  * @param message the data to send.
98  * @param length the length of the data.
99  * @param flags the flags used to configure the sendto operation.
100 * @param dest_addr the destination address
101 * @return how many bytes were successfully sent. If an error has occurred,
102 * a negative value will be returned.
103 */
104 int send_to_socket(socket_t sock, void *message, size_t length, int flags,
```

```
105                     struct sockaddr_in dest_addr);
106
107 /**
108  * @brief Receives a message from a socket.
109  *
110  * Receives a message from a socket.
111  * @param sock the socket.
112  * @param buffer the buffer to read the message into.
113  * @param length the number of bytes to read.
114  * @param flags the flags used to configure the recv operation.
115  * @return how many bytes were successfully read. If an error has occurred,
116  * a negative value will be returned.
117  */
118 int recv_from_socket(socket_t sock, void *buffer, size_t length, int flags);
119
120 #endif
```

Listing A.11: socket.c

```
1  /**
2   * @file socket.c
3   * @brief Cross-pflatform socket interface.
4   * @author Adam Bruce
5   * @date 15 Dec 2020
6   */
7
8  #include "socket.h"
9  #include <stdio.h>
10 #include <string.h>
11
12 #ifdef _WIN32
13 #ifndef _WIN32_WINNT
14 #define _WIN32_WINNT 0x0501
15 #endif
16
17 #include <io.h>
18 #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

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <errno.h>
5
6  #include "../ipc.h"
7
8  int main(void)
9  {
10   IPCMessage m;
11   char buffer[100];
12   int running = 1;
13
14   init_ipc_client();
15   printf("*********************************************************************\n");
16   printf("*             Decentralised Firewall IPC Interface                  *\n");
17   printf("*                          by Adam Bruce                            *\n");
18   printf("*********************************************************************\n");
19
20   printf("\nAvailable commands:\n");
21   printf("  enable   :  Enables communication over the network.\n");
22   printf("  disable  :  Disables communication over the network.\n");
23   printf("  rule     :  Generates a new block containing the rule,\n");
24   printf("              and broadcasts the block for consensus.\n");
25   printf("  shutdown :  Terminates the framework.\n");
26   printf("  quit     :  Quits this program.\n\n");
27   while(running)
28     {
29       printf("dfw>");
30       memset(buffer, 0, 100);
31       memset(&m, 0, sizeof(IPCMessage));
32       scanf("%s", buffer);
33
34       if(strlen(buffer) >= 4 && strncmp(buffer, "quit", 4) == 0)
35   {
36     running = 0;
37   }
38       else if(strlen(buffer) >= 4 && strncmp(buffer, "rule", 4) == 0)
39   {
40     scanf("%s", m.rule.source_addr);
41     scanf("%hd", &m.rule.source_port);
42     scanf("%s", m.rule.dest_addr);
43     scanf("%hd", &m.rule.dest_port);
44     scanf("%s", buffer);
45
46     m.rule.action = DENY;
47     if(strncmp(buffer, "ALLOW", 5) == 0 ||
48        strncmp(buffer, "allow", 5) == 0)
49       {
50         m.rule.action = ALLOW;
51       }
52
```

```
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_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_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, otherwuse the return value will be 0.

4.1.2.7 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.

4.1.2.8 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.

4.1.2.9 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.

4.1.2.10 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.

4.1.2.11 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_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_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, otherwuse the return value will be 0.

4.2.2.7 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.

4.2.2.8 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.

4.2.2.9 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.

4.2.2.10 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.

4.2.2.11 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 }

  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

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_ADVERTISEMENT_HOPS 5

  The maximum number of network advertisement hops.
- #define MAX_CONSENSUS_HOPS 5

  The maximum number of hops before a message is destroyed.

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.

Index