BOLVEDERE: A SCALABLE NETWORK FLOW THREAT ANALYSIS SYSTEM

Submitted in fulfilment
of the requirements of the degree of

DOCTOR OF PHILOSOPHY OF SCIENCE

of Rhodes University

Alan Herbert

Grahamstown, South Africa
December 2017

Abstract

Since the advent of the Internet, and its public availability in the late 90's, there have been significant advancements to network technologies and thus a significant increase of the bandwidth available to network users, both human and automated. Although this growth is of great value to network users, it has led to an increase in malicious network-based activities and it is theorized that, as more services become available on the Internet, the volume of such activities will continue to grow. Because of this, there is a need to monitor, comprehend, discern, understand and (where needed) respond to events on networks worldwide. Although this line of thought is simple in its reasoning, undertaking such a task is no small feat.

Full packet analysis is a method of network surveillance that seeks out specific characteristics within network traffic that may tell of malicious activity or anomalies in regular network usage. It is carried out within firewalls and implemented through packet classification. In the context of the networks that make up the Internet, this form of packet analysis has become infeasible, as the volume of traffic introduced onto these networks every day is so large that there are simply not enough processing resources to perform such a task on every packet in real time. One could combat this problem by performing post-incident forensics; archiving packets and processing them later. However, as one cannot process all incoming packets, the archive will eventually run out of space. Full packet analysis is also hindered by the fact that some existing, commonly-used solutions are designed around a single host and single thread of execution, an outdated approach that is far slower than necessary on current computing technology.

This research explores the conceptual design and implementation of a scalable network traffic analysis system named Bolvedere. Analysis performed by Bolvedere simply asks whether the existence of a connection, coupled with its associated metadata, is enough to conclude something meaningful about that connection. This idea draws away from the traditional processing of every single byte in every single packet monitored on a network link (Deep Packet Inspection) through the concept of working with connection flows.

Bolvedere performs its work by leveraging the NetFlow version 9 and IPFIX protocols, but is not limited to these. It is implemented using a modular approach that allows for either complete execution of the system on a single host or the horizontal scaling out of subsystems on multiple hosts. The use of multiple hosts is achieved through the implementation of Zero Message Queue (ZMQ). This allows for Bolvedre to horizontally scale out, which results in an increase in processing resources and thus an increase in analysis throughput. This is due to ease of interprocess communications provided by ZMQ.

Many underlying mechanisms in Bolvedere have been automated. This is intended to make the system more user-friendly, as the user need only tell Bolvedere what information they wish to analyse, and the system will then rebuild itself in order to achieve this required task. Bolvedere has also been hardware-accelerated through the use of Field-Programmable Gate Array (FPGA) technologies, which more than doubled the total throughput of the system.

Acknowledgements

I would like to acknowledge and extend thanks to a number of individuals who gave me support and guidance throughout the duration of this research. Firstly, I would like to give thanks to my supervisor, Professor Barry Irwin. His guidance and support has been key to my successes in this research.

I would like to also extend thanks to my family for their continuous support and care for my general well-being. I pass my gratitude on to my wife, Sindisiwe Herbert, who stood by me throughout this research. Her language contributions helped me to create a more well-rounded document. Praise must also be given to Donna Stevens in this regard who acted as my primary language proofreader throughout this document.

I would like to thank the Computer Science department at Rhodes University for providing me access to the equipment and space required to carry out this research.

This work was undertaken in the Distributed Multimedia Centre of Excellence at Rhodes University, with financial support from Telkom SA, THRIP, Coriant, Easttel and Bright Ideas 39. The author acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Finally, I wish to acknowledge the support of the Council for Scientific and Industrial Research and the Armaments Corporation of South Africa for the financial support for this research.

Contents

List of Tables

List of Figures

Abbreviations

The following list contains the various abbreviations and acronyms used in this document. Citations of research related to these terms can be found in the text body.

ACK	ACKnowledgement
AMQP	Advanced Message Queueing Protocol
ARP	Address Resolution Protocol
AS	Autonomous System
ASIC	Application Specific Integrated Circuit
BGP	Border Gateway Protocol
BPS	Bytes Per Second
CERT	Computer Emergency Response Teams
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
C&C	Command and Control
DDoS	Distributed Denial of Service
DFT	Discrete Fourier Transform
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DoS	Denial of Service
DPI	Deep Packet Inspection
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FIFO	First In, First Out
FIN	FINish
FQDN	Fully Qualified Domain Name
FSID	Flow Source IDentification
Gbps	Gigabits per second
GCC	GNU Compiler Collection
GMII	Gigabit Media-Independent Interface
GNU	GNU's Not Unix
GPGPU	General Purpose Graphical Processing Unit
GPU	Graphical Processing Unit
HDL	Hardware Description Language
IANA	Internet Assigned Numbers Authority
IC	Integrated Circuit

ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IGP	Interior Gateway Protocol
IHL	Internet Header Length
IO	Input/Output
IOC	Indicator Of Compromise
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IP Core	Intellectual Property Core
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISDN	Integrated Switch Digital Network
ISP	Internet Service Provider
PARC	Palo Alto Research Center Incorporated
PC	Personal Computer
PCIe	Peripheral Component Interconnect express
LAN	Local Area Network
LIR	Local Internet Registry
LRU	Least Recently Used
LTS	Long Term Support
LUT	Look-Up Table
Mbps	Megabits per second
MD5	Message Digest 5
MII	Media-Independent Interface
MLP	MultiLayer Perceptron
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIC	Network Interface Controller
NTP	Network Time Protocol
N-ISDN	Narrowband Integrated Switch Digital Network
OpenCL	Open Computing Language
OSPF	Open Shortest Path First
PHY	PHYsical layer
PoPI	Protection of Private Information
POTS	Plain Old Telephone Service
PPS	Packets Per Second

RabbitMQ	Rabbit Message Queue
RAM	Random Access Memory
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RST	Reset
RTL	Register-Transfer Level
SANReN	South African National Research Network
SHA	Secure Hashing Algorithm
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMB	Server Message Block
SNMP	Simple Network Management Protocol
SoC	System on Chip
SQL	Structured Query Language
SSH	Secure SHell
Subnet	Subnetwork
SYN	SYNchronize
TCP	Transmission Control Protocol
ToS	Type of Service
TTL	Time To Live
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
VDSL	Very high bit-rate Digital Subscriber Line
VHDL	Very high speed integrated circuit Hardware Description Language
ZMQ	Zero Message Queue

1

Introduction

THE need for network security is constantly increasing. This demand is caused by the increase in active Internet-capable devices and users, and the increasing integration of infrastructure (in both the public and the private sectors) with the internet. The services and information provided by network users are subject to attack by malicious sources, and to misuse, with the consequence of interrupted services (Furnell *et al.*, 2007).

It cannot be expected that every Internet user be aware of every security flaw or risk that leaves the user and their devices vulnerable to exploitation. Furthermore, misuse of resources such as bandwidth or allocated data quotas by users, whether intentional or not, also serves to affect the quality of service on a network. Because of this, automated systems need to be put in place in key locations within core networks to monitor transactions that occur on large-scale networks (Ioannidis *et al.*, 2000; Qin *et al.*, 2012). These large-scale networks, more often than not, differ in configuration and topology, which adds to the difficulty of threat monitoring within these networks.

For this reason, application-specific hardware and software need to be designed to suit the specific network for which they are intended. Alternatively, mechanisms need to be put into place that can translate occurrences in each network type to a common language. Protocols

that act as such a mechanism and provide a common communication base do exist. One family of such protocols is NetFlow. NetFlow is implemented by pairing unique identifiers with the data fields used to transfer data on networks (Claise, 2004). It focuses on the existence of a connection between endpoints rather than on every detail of the connection, and logs only the metadata of a network flow. Following this protocol helps to add meaning to a log file, irrespective of the source of that log.

Failure to properly monitor and mitigate threats results in problems ranging from network slow-downs (due to bandwidth use by malicious sources) to identity theft (through the collection of relevant information from databases or directly from users of these large networks). Recent events include Conficker (Microsoft, 2009), Mirai (Kolias *et al.*, 2017), WannaCry (Mohurle and Patil, 2017) and Petya (Gordon *et al.*, 2017).

1.1 Problem Statement

Underlying security systems that monitor network traffic and both identify and mitigate threats are an ever-increasing requirement in modern, large-scale networks. The user protection offered by such systems is a basic requirement of a secure network. If a network is unable to ensure that its systems are safe from malicious activity, systems on the network could be exploited and used within botnets to proxy malicious data or other malicious activities on the Internet (Abu Rajab *et al.*, 2006). As a whole, this is detrimental to data flow on the Internet, and can lead to the disruption of secure networks attached to the Internet.

Due to the large volume of network traffic generated on modern networks[1], automated security systems are essential for effective network traffic monitoring. These systems should be able to provide feedback to concurrent systems in order to better mitigate malicious activities targeted at the network, or within the network itself. Moreover, developers of these systems must ensure that the overheads generated by these automated systems are not detrimental to the bandwidth available within the network.

[1] Cisco estimates an average of 122 EB per month of data transfer on the Internet (Cisco, 2017).

1.2 Research Goals

This research explores five goals: three primary and two secondary.

1. **Primary Goals**

   (a) The use of multiple hosts is a must if this research is to tackle large-scale problems. These larger problems, in the context of this research, relate to networks that see growing network throughput, as well as networks with a growing userbase. Compounding these two workloads is the growing amount of bandwidth available to users on a network as time progresses. This will naturally increase the network traffic produced by existing users, as well as users added in the future. This research intends to design, implement and execute a NetFlow v9 and IPFIX processing framework named Bolvedere. Bolvedere is intended to be able to scale out horizontally in order to tackle these larger problems through parallel execution.

   (b) The overarching goal of this research is an investigation of the benefits of implementing both a NetFlow v9 and IPFIX processing framework. Processing of NetFlow records means the discernment and restructuring of a NetFlow record into a common, predefined form to promote ease of analysis. The results produced by Bolvedere must also be consistent and reliable, as inconsistencies introduced into stateful methods of analysis will affect later runtime results. This system should also mitigate Central Processing Unit (CPU) resource bottlenecks created by Deep Packet Inspection (DPI) systems through use of its flow-based protocol, as NetFlow records inherently contain less raw data to process than full packet captures (Doyen *et al.*, 2013).

   (c) Interfacing to external and internal hardware- and software-based processing modules will be a key element of effective data analysis in this system. There is little point in processing NetFlow data into a common, usable form if there is no functionality with which to analyse this data. In this case, the system would only serve to transcribe the data, without doing anything with it. Consequently, analysis processor modules will be developed for Bolvedere that can run either in parallel or disjointly from one another and from the NetFlow processing system. A user will be able to add and remove these modules as needed. The overheads introduced through this design will be recorded and analysed in order to determine its capability.

2. **Secondary Goals**

   (a) The system will be tested using both synthetic data (generated according to known real-world events) and actual data collected from NetFlow collectors on real networks. The synthetic testing is useful for gaining a general idea of how a system will perform in a real-world situation. However, the use of real data is a better determiner of the system's performance.

   (b) The NetFlow discernment subsystem of this implementation can be optimized through discerning each record in the NetFlow in parallel with one another. This is possible because each record is disjointed from any other, as they are self-contained by definition; any value held by a record does not affect any other record stored in a NetFlow data packet. For this reason, the final goal of this research is to optimize Bolvedere through the development and fabrication of a hardware implementation of the NetFlow discernment subsystem on an FPGA chip.

1.3 Scope and Limitations

The system implemented in this research is simply a generic log processor; at its core, it is a stream processor of data bytes. This means that it can process any form of data that is logged with a corresponding template to indicate the content and intended interpretation of each field[2]. However, as this research is intended for use in networks and security space, the focus of the research will remain within this context. Furthermore, as this research is intended to work as a log processor with accompanying templates for log discernment, NetFlow will be the only protocol used in this research, and deviation from this in the context of system operation will not occur.

This research aims to remain within the scope of an easily scalable and accessible NetFlow processing and analysis system. Consequently, the hardware required to run the system correctly should be easily acquired and readily available. For this reason, Bolvedere is intended for execution on the x86 and x64 architectures commonly used by commodity- and server-grade systems.

As this research aims to develop a new technology from the ground up, the focus of the results produced may be biased towards the determination of whether the system works correctly and

[2] A data log is of varying size and is described by its related template, which is defined in a 32 bit structure. The leading 16 bits of this structure hold the field identification number, and the second 16 bits hold the size in bytes of the field in the relevant data log. A template can hold as many of these fields as required by the user.

produces meaningful results. There will be discussions of the results acquired from the analysis of actual data; however, these will focus on the correct functionality of the system as a whole, rather than analysing the details of the information presented by these results.

One should also note that, because this system is developed from new beginnings, the aim is to implement it on a single family operating system. The family chosen for this implementation was Ubuntu Linux[3]. In doing so this leaves space for future work in porting this research to other operating systems.

1.4 Document Conventions

In this text, certain words, segments of text and numbers are represented with different fonts, typefaces, sizes and formatting. The use of these is completely systematic and is intended to be consistent throughout this text. They indicate that a piece of text falls under a category other than regular writing. An example of each of these categories is given below.

Equations:

Mathematical equations are indicated by the use of italics. Major equations are centred and given equation numbers in the right-hand margin. For example:

$$f(x) = mx + c \tag{1.1}$$

Listings:

Listings are presented in a single-lined block and in a monospaced font. Each line is numbered to aid ease of reference. All code in a listing is presented as it is found in the original, and as such follows the complete structure of the coding language used. (Ellipsis is used to represent text leading into or continuing from a text snippet within a listing.) Listing numbers are provided for reference as a part of the heading above the listing. An example is given below.

Listing 1.1: Example Code Listing

```
1 ...
2 for(int i = 0; i < 100; i++)
3 {
4     printf("Value = %d\n", i);     // simply display i as a string
5 }
6 ...
```

[3] https://www.ubuntu.com

Byte Fields:

Byte fields in this text are placed inside of a figure wrapper, using the floating default style. This means that the byte field follows all behaviours of that of a figure, including heading style. Byte fields are used to better diagram network packet headers and data structures. An example is given in Figure 1.1.

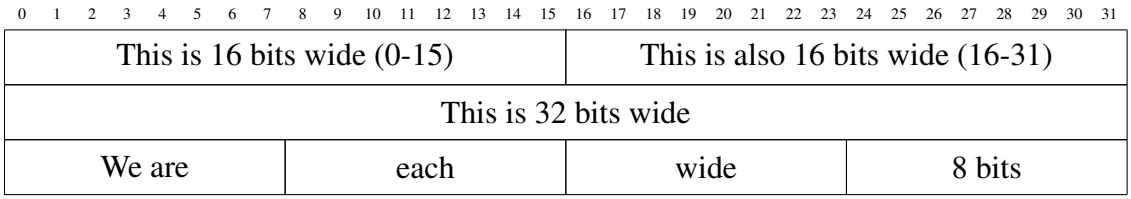

Figure 1.1: Example Byte Field Figure

Number Formatting:

Numbers in this document are rounded to three significant figures after the decimal point. The thousands separator used is a comma (,) and decimals are given after a period (.). For example:

123,456,789.012

Units of measurement relating to a number are separated from the number by a space. For example:

12,751.662 Mbps

1.5 Document Structure

The remainder of this document adheres to the following structure: **Chapter 2** begins with a literature review. The intention of this literature review is to give the reader a fresh understanding of the background material upon which this research is based. As there is hardware involved in this research, the chapter examines the foundations of how networks work. This leads into a discussion of common protocols for Ethernet, including NetFlow and how NetFlow has been used in this project to detect network events. The chapter explores the idea of malicious network events and how these events are categorised, and reviews the methods used to

detect and mitigate iconic network attacks. Finally, Chapter 2 details the history, application and implementation of FPGAs and ASICs, the hardware used for acceleration of Bolvedere.

Chapter 3 discusses the design and implementation of Bolvedere. It proposes a "structure" for the logical flow of data through Bolvedere, and describes how the project design intends to achieve the goals set out in Section 1.2. From this point, the chapter discusses which technologies or new approaches could be possibly used to achieve the intended functionality of each part of the system. The chapter closes with a review of all proposed solutions to each of the subsystems that make up Bolvedere, before drawing its final conclusion as to why the proposed implementation will yield the best results.

Since Bolvedere as a whole is intended to both initially process NetFlow data records, and use selectable modules to analyse these processed NetFlow data records, **Chapter 5** focusses on the algorithms implemented within these analysis modules. These modules apply anomaly detection through geolocation, neural networks as well as mathematical transforms in order to pick out specific events within a network. Each module is discussed in detail: the algorithm used, how it is implemented, technologies used and how and where it is executed.

Testing then occurs in **Chapter 6**. This testing is broken up into sections based on each subsystem of Bolvedere, and then each analysis module that uses processed NetFlow data records from Bolvedere's NetFlow processing subsystems. As each analysis module is a system in itself, depending on the complexity of the module, further testing of the module's subsystems may occur as subsections within **Chapter 6**.

Bolvedere's relevance to today's physical networks is then explored through real-world data analysis in **Chapter 7**. This chapter takes blinded NetFlow data records[4] and streams them through Bolvedere at near real-time[5] in order to test Bolvedere's capability in real-world situations. The results gathered in this chapter are due to Bolvedere receiving NetFlow data records from NetFlow sources on real-world physical networks.

In **Chapter 8**, optimisations are made to Bolvedere's collector subsystem in the form of custom hardware, designed and implemented from the ground up using FPGAs. This chapter describes the hardware in terms of functional components, and goes on to show how each of these functional components interacts with the others to achieve the equivalent input-to-output relationship of the software implementation of the Bolvedere collector subsystem. This leads

[4]The IP addresses associated with hosts in the NetFlow logs are re-assigned so as to blind access to private information in terms of any individual activities on a network (Section 7.1.1).

[5]Real-time is not possible in the context of the NetFlow protocol, as data records at NetFlow sources are held for a period of time and amalgamated before being passed to a NetFlow sink for processing.

into **Chapter 9**, which shows that the hardware and software implementations are functionally equivalent. **Chapter 9** also draws out statistics to quantify the performance enhancement provided by the dedicated custom hardware solution.

Finally, this document concludes in **Chapter 10** by tying the results achieved by this implementation (in the contexts of both testing and application in data analysis) back to the goals set out at the start of this research. The success of this research is examined, leading into a final discussion on the future development and progress of Bolvedere.

2

Literature Review

THIS chapter gives the reader access to the background knowledge required to better understand this research. As this research delves into implementing some of itself on hardware, this literature review begins at the bottom of the network stack with an explanation of Ethernet in **Section 2.1**. This section leads onto a common protocol built on top of Ethernet in **Section 2.2** known as the Internet Protocol (IP). Once IP is defined, this chapter looks towards understanding the NetFlow protocol which is built on top of IP in **Section 2.3**. Some history on these protocols as well as how each of them work and where they are used will be covered in these sections.

The idea of a network event, and more specifically malicious network events, is outlined and then discussed in **Section 2.4**. This section touches on different kinds of malicious network activity as well as the methods used in performing these attacks. This section then leads into **Section 2.5** which discusses some well known malware that makes use of previously discussed methods in order to penetrate systems. The effects of and further distribution of these malware are each discussed in further detail in the relevant subsections and indicators of compromise are addressed directly in **Section 2.6**.

Methods in defending against these malicious network events is then targeted for discussion in

Section 2.7. This section focusses on Distributed Denial of Service (DDoS) attacks, what they are and how one can best mitigate them. There is much research into detection of this network event and the most common forms can be found in this section and **Section 2.8**. The theme of defence is continued in **Section 2.9** with detection of spoofed source packets based on the IP protocol. This method discusses the use of geolocation based on the source IP address of a packet, the known physical location of the monitoring node and LIRs.

A short introduction to what a botnet is and what it can do is given in **Section 2.10** before discussing an implementation of a botnet detection engine. Leading on from this **Section 2.11** discusses well know platforms for detection of network events; be it malicious or not. These applications include Bro (Paxson, 1999) and Snort (Roesch, 1999) and discuss how they process network traffic into events and how these can be used to mitigate network attacks.

As noted prior in **Section 1.2**, this research focuses on the development of some of its subsystems in dedicated hardware. This however is used as optimization after the system was proven to initially work and so the section that discusses such matters regarding the FPGA's implementation of certain subsystems in this research is left until last. **Section 2.12** details what an FPGA is, how it works and how it can later be used in development and fabrication of ASICs. The closing subsections of this section discusses different fabrication processes used in development of ASICs and their advantages and disadvantages.

2.1 Ethernet

The ALOHAnet (Abramson, 1985) inspired development of Ethernet was performed at Xerox Palo Alto Research Center Incorporated (PARC) between the years of 1973 and 1974 by Robert Metcalfe, David Boggs, Chuck Thacker and Butler Lampson. This standard was published on September 30th, 1980 as "The Ethernet, A Local Area Network. Data Link Layer and Physical Layer Specifications". The standard for Ethernet II was published two years later in November 1982 (Digital Equipment Corporation, Intel Corporation and Xerox Corporation, 1982). Formal standardization efforts that occurred at the same time resulted in the publication of Institute of Electrical and Electronic Engineers (IEEE) 802.3 on June 23, 1983 (Healey, 1983).

2.1.1 Network Link Speeds

Network link speed refers to the speed at which data can be transmitted over a network link. More precisely, it is the rate in which bits can be changed on a link between two end-points such

Table 2.1: Common Network Link Speeds and their Common Names (Nikkel, 2013)

Network Link Speed	Common Name
300 bps - 56.7 kbps	Analogue POTS Modem
64 kbps - 128 kbps	N-ISDN
1.5 Mbps - 52 Mbps	VDSL
10 Mbps	Ethernet 10Base-x
100 Mbps	Fast Ethernet 100Base-x
1 Gbps	Gigabit Ethernet 1000Base-x
10 Gbps	10 Gigabit Ethernet LAN

that the receiving end-point receives all transmitted bits. These link speeds are also synchronized between two end-points. This makes sense as if one end-point was presenting bits to the link at a different rate to what the receiving end-point is reading the link at, data loss or repeated data would arrive at the destination end-point. Common link speeds with their common link names can be referred to in Table 2.1.

Issues in this link speed synchronisation occur when one considers clock drift (Welch and Lynch, 1988). This is the occurrence of slight variation in period lengths of two or more clocks rated at the same frequency. This means that over time the two clocks will fall out of sync. These slight variations in period length are mainly due to physical imperfections in the clocks and heat. This is overcome in networks through a voting process to determine which entity on the link has a "good" clock. It is assumed that a "good" clock would be contained within a certain confidence level and "bad" clocks would fall out of this confidence interval. With this in mind, clocks that do not fall within a set confidence level are excluded from the voting process. Once the best clock has been voted on, all network devices on the link synchronise to that clock in order to mitigate the issue of clock drift (Mills, 1985).

2.2 Internet Protocol

1974 saw the IEEE publish a paper entitled "A Protocol for Packet Network Intercommunication" authored by Vint Cerf and Bob Kahn Cerf and Kahn (2005). The paper described a protocol that was intended to solve the issue of resource sharing using packet switching among network nodes. At first the entire protocol was a monolithic structure named the Transmission Control Program but later was broken down into two modular protocols namely the Transmission Control Protocol (TCP) and IP.

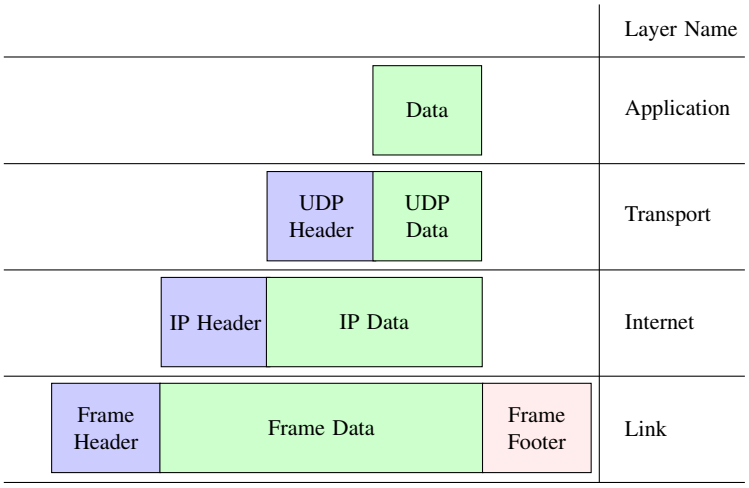

Figure 2.1: Layers of an Ethernet Frame

The specification for Internet Protocol version 4 (IPv4) is dominantly used in current networks and is defined by RFC 791 (Postel, 1981a). This protocol describes headers and fields that are available for use on modern devices that are IPv4 compliant. IPv4 is also used to transport datagrams within the Internet and typically follows the TCP/IP Five Layer Model to transfer data between applications (Kozierok, 2005), an overview of this can be referred to in Figure 2.1.

2.2.1 Routing

In its simplest form, to route a packet means to move a packet from a source host to a destination host through a network (Leighton *et al.*, 1994). What is required to route a packet through a network may very depending on the network and its physical infrastructure. There may be a direct connection to the destined host from the source host, or the packet may need to undergo a series of transfers between routers, hubs and even load balancing systems[1].

There are three primary methods of routing. The first method is interior gateway routing through link states. This method keeps connection and session state to determine the link in which a packet is forwarded within an Autonomous System (AS). The second method is interior gateway routing through use of path vectors or distance vectors, which is based on weights. These weights are defined by the volume of traffic on a link compared to the traffic it can handle and the distance between source and destination hosts. This method also serves to route packets

[1] System in place to distribute client requests among multiple servers.

within an AS. The last method is named the exterior gateway protocol and makes use of tree-like topologies to determine the route which a packet takes between ASs (Mills, 1984).

Border Gateway Protocol (BGP) makes use of path vectors to best determine the path of a packet. This protocol works within an AS between nodes referred to as peers. The paths these peers use to transfer packets within the AS is determined by predefined routes as manually configured by the system administrator (Rekhter *et al.*, 2006).

Interior Gateway Protocol (IGP) is used for exchanging routes and other routing data between gateways inside an AS. This protocol can be broken into two major categories, these being distance-vector routing and link-state routing. Routing Information Protocol (RIP) falls into the former category of exchanging routing information between gateways and makes use of distance values in order to determine the best link for a packet to be forwarded on. This works in such a way that each node within the AS, usually a router, advertises the distance values that it calculates to other nodes and receives their calculated distance values. From here each node updates its routing tables and then reiterates the initial step. This continues until distance values between nodes settle and converge (Malkin, 1998).

To touch on the latter IGP category, one can look to the Open Shortest Path First (OSPF) protocol. This protocol relies on a core node within the network which holds all routing paths within the entire AS. In larger subdivided systems it holds routing paths common to that subdivision. This protocol also makes use of link costs which are used in decision making when it comes to forwarding a packet on a link. These costs can be drawn from a number of factors, from the distance of the link, to the reliability and throughput of the considered link (Moy, 1998; Coltun *et al.*, 2008).

Although these protocols were put in place to route packets, this is not their only purpose. These protocols serve to prevent loops from forming within the routing tables of a network. These protocols also attempt to optimize routes within a network to reduce the time which a packet takes to route from source host to destination host. This also reduces the number of packets within a network at any given time and thus allows for more effective packets in a network for routing (Mills, 1984).

2.2.2 IPv4

This version of IP is well known for its widespread use. This protocol fits in at the Internet layer of network frame and fields within its header can be viewed in Figure 2.2. A description of each header can be found in RFC 791 (Postel, 1981a).

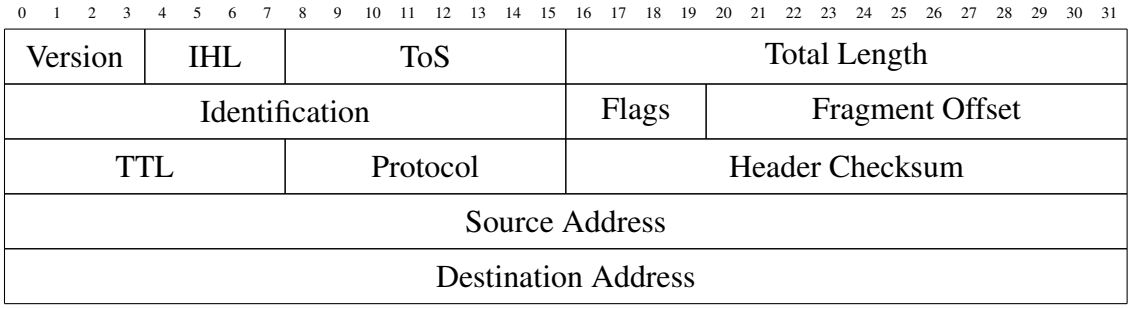

Figure 2.2: Example Byte Field Figure

A major limiting factor of IPv4 is the fact that its source and destination address length is only 32-bits long. This means that there are only 2^{32} available[2] available addresses. This is less than 1 for every human on Earth before automated systems are even considered. At the time of defining the IPv4 standard the thought of more than 4 billion network capable devices, let alone a network of more than 4 billion devices seemed implausible and so a 32-bit address space seemed more than ample for the needs of the world (Postel, 1981a), this however was not the case as most IPv4 addresses had been allocated by 2014 (Huston, 2014)

There are however methods to counteract IPv4 address space exhaustion and prolong the inevitable complete exhaustion of IPv4. The most common method is to create a Network Address Translated (NAT) network (Egevang and Francis, 1994). This method works through the creation of an internal and external network with a host (referred to as "NAT Host") sitting between the two. This NAT Host holds 2 IPs, one for the internal network and one for the external network. This NAT Host serves to translate IP addresses from the internal network and make requests to an external network on behalf of the clients found on the internal network using its own external network IP address; this is performed by use of a NAT table and is typically performed by a commodity network router (Cisco, 2015). Internal clients find this host using the NAT Host's internal network IP address, this IP address is commonly referred to as a default gateway.

A working example for this form of implementation would be best described by its use in the Internet. When one joins the Internet it can simply be assigned one IP address for use (typically assigned to one's router) and have the device which this address is assigned to then treat the Internet as an external network and then create its own internal network through implementing a NAT.

This benefits the Internet by allowing multiple devices to transmit traffic to it and have systems on the Internet transmit back to these NATed devices through the use of a single IP address.

[2] commonly referred to as around 4 billion.

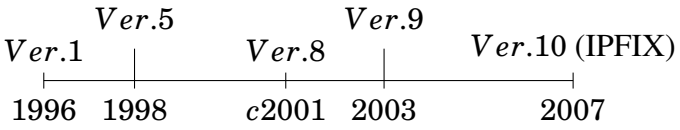

Figure 2.3: NetFlow Version Timeline

This helps to save IP addresses assigned to devices on the Internet, thus combating IP address exhaustion on the Internet.

2.3 NetFlow

The NetFlow protocol is best described as a means of logging network flows that pass through a flow monitoring device in a communication pair's route. A flow is defined by a connection and communication between a host and any other host, multicast group, or broadcast domain in the form of a sequence of packets (Kerr and Bruins, 1996). A flow monitor using the NetFlow protocol can collect information out of these network flows and store them in data fields for transmission to a logging host for analysis and/or storage (Claise, 2004). As NetFlow was initially introduced by Cisco in 1996 (Kerr and Bruins, 1996), there have been multiple versions of NetFlow with wide-spread support over multiple firewall and routing devices.

The need to update the NetFlow protocol over the years arose from multiple factors. First, the addition of Internet Protocol version 6 (IPv6) (Deering, 1998) that was brought about by the IP address exhaustion (Huston, 2014) of the IPv4 space required amendments to be added to the NetFlow protocol; this addition in itself obsoleted NetFlow version 1 due to its exclusive support of IPv4. Furthermore, the need for better use of network resources grew as the amount of traffic passing through flow monitor points increased. Finally, the requirement to adapt these records to one's needs gave way to overhauling the NetFlow protocol to allow users the ability to break out of the predefined logging fields determined by older versions of NetFlow. This overhaul allowed users to define templates containing fields that they required in any order they specified that could then be used to discern data records at a later point (Claise, 2004). This also allowed users to log any protocols defined at a later stage through allocating a user defined template ID to support the new protocol through defining what the new fields are in the NetFlow sink nodes.

To give background on updates that have occurred since the original NetFlow standard one can first observe the additions introduced by NetFlow version 5. This version included the addition

of new record fields and this furthered the standardisation to NetFlow (See Appendix A for a feel as to what a NetFlow record contains) by including logging of subnet masks and AS numbers (Huston, 2006a). Version 8's two main updates were the ability to aggregate data records that were defined in version 5 (Huston, 2006b) and also the ability to custom define extra fields of the user's choosing at the end of the data record log. More recently, version 9 continued to build on the freedom brought forth by version 8's custom appended fields through the addition of the template packet to the NetFlow protocol. This template packet that resulted in an overhaul of the NetFlow protocol allowed one to fully define the fields to be logged in a data record and the order in which they are logged from a network flow (Claise, 2004). Figure 2.3 shows a timeline for release dates of NetFlow version standards.

These templates are coupled with template identification numbers that allows for multiple templates to be used and are defined by the 2 byte long template identification field within the NetFlow protocol. Although IDs 0 through to 255 are reserved for use by specific flow templates, template identification numbers 256 through to 65535 are available for public use; a fairly generous template count. This template count further extends the memory requirements of these devices and most devices supporting NetFlow version 9 limit the number of templates that can be stored to count far less than the available 65535. Allowing for templates to define what should be logged and the order in which it should be done does have drawbacks. These drawbacks are found in memory and performance as the device receiving and using the template has to store the received templates for later use in identification of data records and then expend processing time discerning the related data records (Cisco, 2003a).

JFlow

The Juniper Flow[3] is functionally equivalent to NetFlow, and is completely interoperable with any NetFlow supporting sink (Juniper Networks, 2011). The main distinguishing point is that where NetFlow is developed by Cisco (Claise, 2004), JFlow is developed by Juniper Flow.

sFlow

Developed by HP[4], Sampled Flow differs from NetFlow and JFlow in that it focusses on a statistical approach to monitoring networks (Phaal *et al.*, 2001). The primary benefit of this network flow monitoring protocol is its ability to scale through sampling of every N'th packet.

[3] https://www.juniper.net/us/en/

[4] http://www.hp.com/

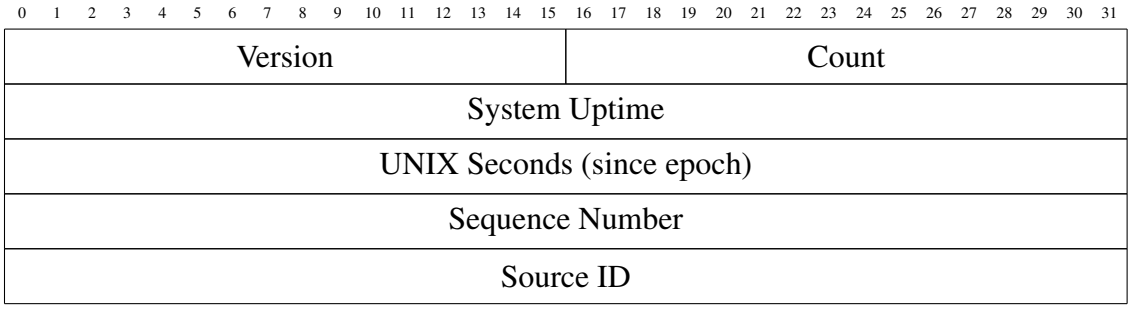

Figure 2.4: NetFlow Version 9 Packet Header Format

This means that a host using sFlow to monitor a network can scale itself to a load it can manage by adjust the gap between packet samples.

The downside of this is that relatively small network connections, when compared to other connection activity on the monitored network, can be completely missed. Furthermore, accurate timestamping of network flows is also traded for this aforementioned scalability, as the end of a network flow may not fall on an N'th sample (Phaal *et al.*, 2001).

2.3.1 NetFlow Version 9 Template Records at a Lower Level

To understand how NetFlow version 9 works one can simply look at how its packets are built[5]. One should note the single pass nature of the NetFlow protocol as this means that no future field in a NetFlow packet is required to process any field before it. This means that stream processing can be performed on a NetFlow packet by a network enabled device. Also it should be noted that the frame, Ethernet, Internet and transport layers are omitted in this explanation as they only exist to ensure the NetFlow packet arrives at its exported destination.

The first header of every NetFlow version 9 packet is 20 bytes long and always contains the information in the order shown in Figure 2.4. This header ensures that any NetFlow version 9 collector knows basic information about the NetFlow source node and whether any NetFlow packets have been missed during collection. To further detail each of these fields one can refer to the list below:

Version: The NetFlow protocol version implemented in the transmitted NetFlow packet.

Count: The number of records (both templates and data) contained in the transmitted NetFlow packet.

[5]The construction of an IPFIX packet is the same as that of a NetFlow version 9 one.

System Uptime: A count of the number of milliseconds since the NetFlow source node was started.

UNIX Seconds: Seconds that have elapsed since 0000 UTC 1970 (Unix Epoch).

Sequence Number: A count of all transmitted NetFlow packets by a NetFlow source node.

Source ID: This is the NetFlow exporter's unique ID number used to identify which NetFlow exporter the NetFlow packet was sourced from.

The reason for the requirement of at least the information contained in this header is due to the requirement to know what version of the NetFlow protocol is being used, if any NetFlow packets had been missed and where the NetFlow packets are being sourced from. The manner in which a missed packet is detected is simply by comparing the last sequence number received with the next one, if it is not what is expected according to the last NetFlow packet received, possible packet loss or corruption could have occurred. Another note is that if one knows where a NetFlow exporter was placed and what source ID was assigned to it, one could effectively say what network flow is occurring from that network without the need to look at the source IP address.

A FlowSet header always follows the NetFlow header and can either contain data records (referring to Figure 2.5) or template records (referring to Figure 2.6[6]). The total records held within all FlowSets in a NetFlow packet cannot exceed 30, however these 30 records can be any distribution of data or template records. One must note that each of these FlowSets are rounded to the fourth byte and thus leads to the requirement of padding at the end of a data record FlowSet. It is also notable that each FlowSet has the same header containing the FlowSet ID and the length of the FlowSet.

The template FlowSet header and template data follows the format in Figure 2.5. The FlowSet ID is used to notify a NetFlow collector of which template to use when discerning NetFlow data records. In the case that there is a zero in the FlowSet ID, it symbolizes to the NetFlow collector that this FlowSet contains template data. Each field following this is 2 bytes in length and so is kept in pairs as to always align on the 4 byte boundary removing the need for padding within this type of FlowSet. Following a FlowSet header symbolizing that it is carrying template data, the FlowSet can contain multiple templates. The length in the within the FlowSet header is intended to notify a NetFlow collector of where the FlowSet ends. Where a template record ends within the Flowset is handled by its own header.

[6]Assumes all records are size 2 bytes for display purposes, however these do vary in size.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
FlowSet ID = 0	Length
Template ID	Field Count
Field 1 Type	Field 1 Length
Field 2 Type	Field 2 Length
...	
Field N Type	Field N Length
Template ID	Field Count
Field 1 Type	Field 1 Length
Field 2 Type	Field 2 Length
...	
Field N Type	Field N Length

Figure 2.5: NetFlow Version 9 Template FlowSet Format

Each template record within the template Flowset contains its own two field header stating what ID this template will be referred to by incoming data record FlowSets, and the number of fields within the template record. As stated, each template field is made up of a pair of 2 byte fields. This means that the field count multiplied by 4 will give the location of the start of the next template record and, if the FlowSet is correct, the addition of all the template record lengths multiplied by 4 will equal that of the template length plus 4 for the Flowset header.

To define a field within a template a type and length is required. The type is defined by a predefined ID number as specified in the NetFlow version 9 and IPFIX protocol documentation. There is also reserved space for custom types for prototyping or user specific data collection. The length field is required as some predefined types are of variable length and this information is required in order to read a data record correctly.

Data record FlowSets cannot be interpreted without the collector having received a template record FlowSet containing the Template ID specified by the FlowSet header in a data record FlowSet. This is because the data record FlowSet, other than the FlowSet ID and length fields, contains no information as to where a record starts, stops, or what each field contains; this information is trivial to work out with the relevant template.

If the template is not received, the task of skipping over the entire FlowSet is performed through use of the length field contained in the FlowSet header. If the template is possessed by the NetFlow collector the length of each data record can be calculated from this point by adding the

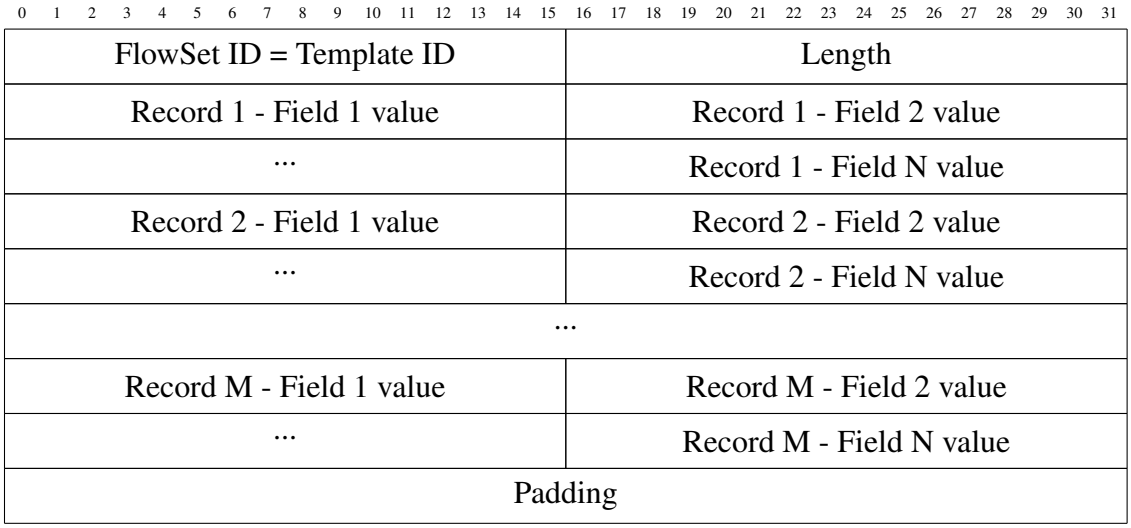

Figure 2.6: NetFlow Version 9 Data FlowSet Format

length of all record length fields. Furthermore the start of every record as well as field within each record within the data record FlowSet can be calculated using the record's template.

After every data record in the FlowSet has been processed, padding is used to align the FlowSet to the 4 byte boundary. A FlowSet may not align on this boundary due to the variance in field size within records in the data record FlowSet.

2.4 Malicious Internet Activity

A malicious event acted upon something or someone is simply defined as an action with intent to do harm to that entity. Within the Internet these entities are commonly found to be end-point hosts. These hosts are typically targeted in order to collect information or prevent other entities on the Internet from accessing the information or services provided by the host, be it private or public.

This section begins with a brief summary of what it means to hash a string and then leads into some methods in which these attacks are performed, whom these targets are, and what these attacks aim to do. Furthermore, this section looks at some of the more well known attacks and what their purposes were, as well as some interesting discoveries that came about because of their implementation on the Internet.

2.4.1 Brute-Force

This form of an attack is applicable to multiple forms of attack and is defined by having little to no intelligence to the attack methodology (Leslie, 2014). The method of this attack simply tries every combination from start to finish of the input range until a solution is found. As this attack intends to attempt every combination as input to a system in order to gain access to information held by the system, this approach is time consuming and thus attempts to leverage high throughput hardware in order to achieve this task (Paar and Pelzl, 2009). This hardware includes the Graphical Processing Unit (GPU) through the use of Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL).

2.4.2 Port Scanning Techniques

A port scan is a method of sending specific packets at a target in order to determine which network ports are open, closed or filtered by a firewall on the target. This information leads to acquiring knowledge of what services are running on a system. The reason this information can be collected is that set ports are reserved for a specific service. This information can help a potential attacker better aim their attack at a service they know is vulnerable or avoid wasting time on a system they know they can not penetrate (Lyon, 2015).

There are multiple ways of performing a port scan on a host with different goals. Some of these methods and the data they aim to collect are listed below:

TCP SYN: A simple method of sending a Synchronize (SYN) packet to a port on a host, if the port is open and waiting connection it will respond with a Synchronize-Acknowledge (SYN-ACK) packet. If the port is closed it will respond with relevant Internet Control Message Protocol (ICMP) packet stating that the port is closed. If no response is received from the SYN request, it is assumed that the port is filtered by a firewall. At this point the communication is closed by the scanning host as the information required is collected and the next port is probed.

Connect(): This method is used when the system performing the scan does not have access to raw sockets in order to send packets out. Instead this scan uses the underlying operating system's own TCP network handles in order to send the SYN packet to a target. If the operating system returns from the function call with success, it means that the port was open in which the connection attempted to connect to. If the function responds with port

closed, the port was indeed closed. If the function responds with a connection timed out, it is assumed that the port is filtered.

ACK: This scan's purpose is to determine whether a firewall exists on a target, and in the event that it does exist, it determines whether the firewall software is stateful or not. It is an interesting point to raise that this form of scan cannot actually determine whether a port is open or closed. This characteristic is due to the fact that an Acknowledge (ACK) packet arriving at an open or closed port warrants the same response, this being a TCP Reset (RST) packet. The only option left is the event that no response is received, this means that the port is filtered by a firewall that is most likely stateful.

Window: This scan, like the ACK scan, is implemented by transmitting ACK packets at the target and determines results using the same method outlined by the ACK scan. The difference is in a detail in the TCP RST response packet from the target that some target systems include. This detail is the window size of the packet. When the out of order ACK packet is received and the RST packet is transmitted back to the scanning host, some systems will respond with a positive window size if the port is open and a window size of zero if the port is closed. This detail can be used to determine whether a port on a system is open or closed on a target system with this vulnerability.

Maimon: Named after its discoverer Uriel Maimon, the scan makes use of the Finish (FIN) and ACK flags in a TCP packet. This scan technique is used against BSD systems to resolve whether a port is open or not. It was observed that this packet arriving at a closed port on a BSD system yields a TCP RST packet to be sent in response where an open port simply drops the packet and does not respond at all. This can be used to tell whether a port is open or closed on a BSD host (Zhang *et al.*, 2008).

2.4.3 Vulnerability Exploitation

Humans are not perfect and as hardware and software are developed by people, there are imperfections introduced into the systems. These imperfections lead way to unexpected behaviour that when acted upon leads to results that fall outside of the systems intended operation. These results can range from simple errors in output to code being remotely uploaded and executed on the system or system being shut down completely.

Malware such as Blaster Worm (Dougherty *et al.*, 2003), Conficker (Microsoft, 2009) and SQL Slammer (Shannon and Moore, 2004) use these vulnerabilities to upload and execute them-

selves upon remote hosts. These vulnerabilities exploited by these malware include MS03-026 (Microsoft, 2003b), MS03-039 (Microsoft, 2003c), MS08-067 (Microsoft, 2008) and the Microsoft SQL Server Resolution Service (Shannon and Moore, 2004). Even though these are well known vulnerabilities that have long since been fixed, there are still many existing systems on the Internet that are still vulnerable to these attacks due to improper maintenance of said systems. These iconic attack methods were chosen to show their age (dating back to January 2003 in the case of Blaster) and further exclaim the neglect shown by system administrators.

2.4.4 Availability of Attacks

The large number of malicious attacks occurring on the Internet on daily basis is due to two reasons. The first is the ease in which one can perform these attacks. For the most part someone with little to no knowledge of how a vulnerability is exploited on a network can simply get hold of existing tools and scripts to run at the press of a button, that all one needs do is point at a target. The second is that many malicious attacks are automated, be it via botnet (Gu *et al.*, 2008) or by a malicious preconfigured system.

One of the reasons for the popularity of these attacks is that there is a market for these systems, botnets and zero day attacks (a vulnerability that is yet to be exploited and is unknown to the vendor) and these can fetch a high price depending on the capabilities of the exploit. There are also freely available tools for configuring and performing malicious exploits of systems on a network (Bilge and Dumitras, 2012) for the purpose of "testing" a network's security or because the exploit is no longer deemed current. These exploits are for the most part well known and fixed and for a target to fall victim to these forms of exploitation is due to their own negligence in terms of keeping their system up to date. This section will deal with the two common ways exploits are performed in a legal penetration testing environment.

Metasploit

This software suite is directed at penetration testers to test the security of networks and users on it (O'Gorman *et al.*, 2011). It houses a wide variety of tools that allow for assessment of software and systems running on a network as well as the awareness of the users on a network through features like the generation of phishing campaigns to test users on a network. Coupled with these features is a database of fully functional exploits that are known to the penetration testing community. This means that one can install Metasploit, which is free, and launch these

attacks on a target host on a network at one's leisure (Maynor, 2011). This can of course be for Metasploit's intended purpose, that being security consciousness, or for malicious reasons.

For this reason, platforms such as Kali Linux[7] exist as a dedicated penetration testing operating system distribution that comes pre-installed with Linux-based software that one would use for penetration testing of a system (Muniz, 2013). Furthermore, the Metasploit community supports this Linux distribution and thus regular updating of the distribution and tools on it is freely available.

Reuse of Code in the Wild

There is a need to understand existing malware on the Internet and because of this collecting and monitoring the redeployment of malware in a safe environment can yield information that can be invaluable in combating it (Perdisci *et al.*, 2008). There are multiple methods of capturing malware and the class of software that typically performs this action is referred to as a "honeypot"; although this is not the only way to capture a piece of malware. A honeypot acts as a vulnerable system in order to attract malicious attacks (Provos, 2004). Any attacks that are targeted at the honeypot are then logged and any uploaded data is stored for later review. From this point one can set up an environment such as a virtual network, or if one has the resources, a live environment in which to rerun these malware and analyse their characteristics.

Other methods of malware collection range from malware sharing communities to collecting the remnants of uploaded scripts and programs to a server that may have resulted in a failed or successful attack. Either way deletion of these malware are a loss to the community trying to combat these forms of malicious attack and one should attempt to pass on the malware to a party that has a use for it (preferably not malicious in nature).

2.4.5 Distributed Denial of Service Attack

The goal of a DDoS attack is the same as that of a Denial of Service (DoS) attack, in that they both aim to bring down a service that exists on a network. The key difference between a DDoS attack and a DoS attack is that a DDoS attack uses multiple physical source hosts rather than a singular host. These hosts usually exist within a botnet (Gu *et al.*, 2008). Note that a DoS attack can appear to be a DDoS attack through the spoofing of multiple IPs, which makes detecting a DDoS attack difficult.

[7] https://www.kali.org/

Motivators

Motivation for denying a service is varied and at times can have no motive at all. The reason for the latter is that clients can unknowingly perform a DoS attack, or even inherent structures of a network lead to the occurrence of DoS; broadcast and multicast storms are examples of this.

In short, multicasting means that a host is sending a packet which will be received by multiple destination addresses (Deering, 1997). This allows for a reduction in the overheads of broadcasting a message to multiple clients, as the server no longer has to send copies of the same packet to each individual host. The problem arises when a request requires additional information from a non-existent entity within the network. If the client on the system then broadcasts to all other clients requesting the location of this service, it can lead to all clients that receive the broadcasted message to repeat the request to every other clients, in order to locate the missing service; this logic would eventually snowball and flood the network with these requests (Tseng *et al.*, 2002).

This effect is hard limited by the bandwidth of the network. Once the network bandwidth limit is reached, excess packets will be dropped, however the bandwidth will still be saturated. The problem with this is that no other services on the network can function as expected, as there are little to no network resources for these services to acquire to perform their communications; the network is now suffering from DoS.

Reasons for a deliberate DoS attack can be varied. One of these attacks can be launched on a target service, or set of services, based on nothing more than a way of protesting against what the service provider stands for or ransoming a network's bandwidth for money (Halpin, 2012). On the other hand, there have been cases of DDoS occurring to online stores due to servers not able to handle the load of all clients at a given time. These mostly occur during special offers when a flood of customers aim to purchase a product and even though they know that the provided service is under strain, they continue to attempt to gain access to the site through repeated requests of the resource in order to get their share of the product (Jensen and Gruschka, 2008).

Perspective

Mirai (Biggs, 2016) and BASHLITE (Gallagher, 2016) are two malware that seek to infect hosts running a Linux-based OS. Once infected these hosts could be controlled remotely to perform tasks on behalf of a third-party (in other words these hosts were included in a botnet).

This remote control was used to generate more than 620 Gbps of network traffic directed at the website "Krebs on Security"[8] as well as over 1 Tbps at web host OVH[9], . This amount of network traffic was enough to bring these web services offline. More information on the Mirai malware can be found later in Section 2.5.6.

Prevention

As the types of DDoS attacks vary, one cannot expect one solution to cover all angles of a DDoS attack, instead one has to rely on multiple defensive measures in order to secure their resources from such occurrences. These defences include use of firewall ruling, network switch, router configuration and software-based solutions (Mirkovic and Reiher, 2004). Firewall implementations exist in both software (Bishop, 2003) and hardware (Cho *et al.*, 2002) and both aim to provide defence through a rule-based system. These rules can be as broad as blocking all traffic from a range of IP addresses, to looking for a specific byte in a specific location within a packet of set protocol.

Some switches and routers come with filtering capabilities that allow for DoS prevention. This includes rate limiting and traffic shaping. These all deal with the throughput of connections and aim to limit them to allow other connections the opportunity to have their packets served (Raghavan *et al.*, 2007). Further functionality such as packet inspection and bogon filtering use the data within the packet to decide whether to process a packet or ignore it.

2.5 Selected Historic Malware

This section touches on some of the more common and major malware in existence. This allows the reader to further understand the reason why malware exists, be it for personal gain or to make a statement, and the means which software is exploited in order to infect a system. The information provided here is given to the reader to help better understand how such infections occur and why it is still possible to infect a host today.

[8] https://krebsonsecurity.com/
[9] https://www.ovh.ie/

```
offset    0  1  2  3   4  5  6  7   8  9  a  b   c  d  e  f  0123456789abcdef
00001a00  00 30 40 00  3c 31 40 00  00 80 00 00  00 00 00 00  .0@.<1@.........
00001a10  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00001a20  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00001a30  00 00 00 00  00 00 00 00  00 00 00 00  6d 73 62 6c  ............msbl
00001a40  61 73 74 2e  65 78 65 00  49 20 6a 75  73 74 20 77  ast.exe.I just w
00001a50  61 6e 74 20  74 6f 20 73  61 79 20 4c  4f 56 45<20> ant to say LOVE
00001a60  59 4f 55 20  53 41 4e 21  21 00 62 69  6c 6c 79 20  YOU SAN!!.billy
00001a70  67 61 74 65  73 20 77 68  79 20 64 6f  20 79 6f 75  gates why do you
00001a80  20 6d 61 6b  65 20 74 68  69 73 20 70  6f 73 73 69   make this possi
00001a90  62 6c 65 20  3f 20 53 74  6f 70 20 6d  61 6b 69 6e  ble ? Stop makin
00001aa0  67 20 6d 6f  6e 65 79 20  61 6e 64 20  66 69 78 20  g money and fix
00001ab0  79 6f 75 72  20 73 6f 66  74 77 61 72  65 21 21 00  your software!!.
00001ac0  05 00 0b 03  10 00 00 00  48 00 00 00  7f 00 00 00  ........H.......
00001ad0  d0 16 d0 16  00 00 00 00  01 00 00 00  01 00 01 00  Đ.Đ.............
00001ae0  a0 01 00 00  00 00 00 00  c0 00 00 00  00 00 00 46  .......À......F
00001af0  00 00 00 00  04 5d 88 8a  eb 1c c9 11  9f e8 08 00  .....]..ë.É..è..
                                                             10,78  Command
```

Figure 2.7: MSBlaster Binary Viewed in Hex Viewer

2.5.1 Blaster Worm

This worm that exploits the Remote Procedure Call (RPC) service in systems running Microsoft Windows XP was first observed on the 11th of August, 2003. More details regarding this exploitation is detailed by the technical documents MS03-026 (Microsoft, 2003b) and MS03-039 (Microsoft, 2003c). Exploitation of these vulnerabilities typically involves sending a specific malformed packet to a RPC port numbered 135, 139, 445 or 593 depending on the system's configuration. This worm comes in four major variants with as many as twelve more that existed on the Internet at the peak of this worm's life cycle.

The specifics of this worm are amusing when one looks into the hand written parts of the code. Embedded in the code are two strings that only serve the purpose of calling out Bill Gates, co-founder of Microsoft, and declaring love for a San; be it person or object. These messages can be referred to in the compiled code's binary in Figure 2.7.

Other than relaying a message this worm attempts to disable Microsoft's update servers by having every machine successfully infected by the Blaster worm to SYN flood port 80 of the Fully Qualified Domain Name (FQDN) windowsupdate.com; this on a large scale would lead to a DDoS attack targeted at this FQDN. This worm would also lead to the development of the Welchia worm (Section 2.5.2) which would use the same means of infection as Blaster (Dougherty *et al.*, 2003).

An interesting notes on the side effects of this worm is that infection would cause the RPC service to crash causing Windows to schedule a reboot. This was often the first sign given by

this worm that it may have infected one's computer (Trend Micro, 2003). This crash however was not a definite as the later developed Sasser worm also caused the same symptom to occur.

2.5.2 Welchia

Also known as the Nachia worm, this worm was developed as a countermeasure to the Blaster worm. Welchia would use the same methods of infection as Blaster did to infect a system, this being the MS03-026 (Microsoft, 2003b) and MS03-039 (Microsoft, 2003c) vulnerabilities found in Microsoft Windows XP systems. After infection the Welchia worm would download and then execute code that would search for and disable the Blaster worm's functionality. This was achieved by attempting to delete MSBLAST.EXE; the Blaster worm's binary. Along side this directive the Welchia worm also attempted to update the OS to the latest version and apply patches to prevent infection or reinfection of the Blaster worm.

As this worm's intent was not malicious, given that its goal was to remove a malicious software, this worm was classified as a helpful worm. At first when this worm was discovered, virus scanner companies determined this worm as a high priority threat but after realisation of what its intent was the threat level priority was dropped to low.

This worm first appeared on the Internet on the 18th of August 2003 and was programmed to self-remove after a 120 days of operation or after the date of the 1st of January 2004. This means that the worm in theory should no longer exist on the Internet. This worm was stated by Microsoft to have achieve wide spread success, although there were cases of the worm not being able to apply updates and patches to prevent reinfection due to lack of connectivity. Microsoft followed up this worm's countermeasure with the release of a stand alone Blaster worm removal kit twelve days after the theoretical end life of the Welchia worm (13th of January 2004) that would allow one to manually remove the Blaster worm from a system (Brasford, 2004).

2.5.3 Storm Worm

Storm worm goes by many names such as Small Dam, Peacomm and Nuwar and is the reason for the existence of the Storm botnet. This trojan is typically distributed via executables attached to an email or download off a remote server that promises access to data or functionality that it does not. Once executed on a Microsoft OS it proceeds to install itself on the host and turn it into a zombie on the Storm botnet (Porras *et al.*, 2007).

A zombie is simply defined as an Internet enabled system that has been compromised in such a way that it can have code executed on it remotely without the system's administrator's consent. The most common tasks these zombies perform is spam email, DDoS attacks, click fraud (automatic clicking of advert links on sites to bring in revenue on pay per click adverts) and site hosting (Geng and Whinston, 2000).

Storm Worm was first observed on the 19th of January, 2007, and the primary sites of infection were Europe and the United States, this is most likely due to most of the baiting emails being in English. The reason for such success in the bait used is due to the interest perked in the subject lines of the bait emails sent out and believable enough executable files that were attached to these bait mails that people would then download and run (Symantec, 2007a). One can refer to these subjects and file names in Appendix D.

Storm Worm's survivability is due to two key factors, this being a method of Domain Name System (DNS) name resolution called Fast Flux and the legal allocation of these FQDNs used to host this botnet. Fast Flux is a technique used in which multiple IPs are associated with a single domain name and are purposefully set up to have a short life span. The IPs to which these hosts belong take turns in registering and de-registering themselves with the domain name. This increases the difficulty in pinpointing an IP as the changing IP obscures the associated Command and Control (C&C) network traffic. Furthermore, these hosts are usually just infected hosts in the botnet and serve little more than as a proxy for the botnet's traffic, thus allowing for a further line of obfuscation (Nazario and Holz, 2008). On top of this is the fact that most domain name registers would rather abstain from removal of such domain names due to the fact that the owners of these domains are legal paying customers. This protects the owner of these domain names from having their domains taken down or being denied traffic to their rotating IPs.

2.5.4 Conficker

This computer worm was first discovered in November 2008, infected computers over 190 countries and took over 5 different revisions in its life cycle. The recorded attack vectors used by this worm, enabling its penetration into systems, was exploitation of the MS08-067 vulnerability prevalent in Microsoft Windows 2000, XP and Server 2003 operating systems, and transfer via removable media through creation of an AutoRun script on the media device (Microsoft, 2008). The MS08-067 vulnerability attacks netapi32.dll typically through Server Message Block (SMB) on port 445 by using a specially crafted packet to cause a buffer overflow

and inject and execute code remotely[10].

Infection from AutoRun is due to a feature in Windows 2000, XP, Server 2003, Vista and Server 2008 where an autorun.ini file would be searched for on load of the removable media and code executed according to the script in the autorun.ini file. This means that executable code and an autorun.ini could be placed onto a removable device by Conficker, and when inserted into another system, would automatically run and infect that system.

Other methods of infection used by this worm were use of unsecured shared folders on a network and also exploitation of users with weak usernames and passwords. The use of multiple attack vectors used by Conficker made it difficult to counteract, and infections ranged from home users through to private sectors and even government.

Originally Conficker did not have any self-preservation methods built into itself. However as revisions were made to it, so was functionality to help keep it on a system. This functionality began with blocking DNS lookups of sites that could potentially give a user access to tools or information to remove Conficker. It also disabled Windows AutoUpdate which prevented installation of a patch released by Microsoft that fixed the MS08-067 vulnerability. Later functionality included disabling of Windows Safe Mode, in memory patching of dnsapi.dll to include further prevention of sites that could potentially remove Conficker from a system and a mechanism that would terminate anti-virus software when discovered and would rescan every second to check whether the anti-virus software had rebooted and repeat the process if it had. Each version of Conficker also included its own updating mechanism which would automatically update itself to the latest released version of itself further increasing its survivability (Microsoft, 2009).

The goals of Conficker were to gain access to user's personal information and to grow its botnet (this being performed through infection of more systems connected to an infected system). Personal information included banking information, credit card information, passwords and identity theft in the form of IP addresses.

2.5.5 SQL Slammer

A simple UDP-based worm that in its entirety occupied a total of 376 bytes and exploited a buffer overflow in Microsoft SQL Server Resolution Service (running on port 1434) in order to upload and execute its payload. Once running this worm simply starts resending itself at randomly generated IP addresses. If a random IP happens to be running a vulnerable version of

[10]http://www.speedguide.net/port.php?port=445

Microsoft SQwarlords of draenorL Server, it too will become infected and repeat the process of the attacker (Shannon and Moore, 2004).

David Litchfield, the discoverer of the Microsoft SQL Server vulnerability and creator of the original code base for this worm, stated that the code is simple to the point that it doesn't have the ability to write to disk (Leyden, 2003). Instead the code sits in memory and executes until the system or service is rebooted; an action that would deallocate the worm's memory thus terminating its process. He further states that even though this would disinfect the system, it is likely to get reinfected soon after this process is complete as the system would still be vulnerable to infection.

Once this worm was released onto the Internet it saw an initial infection count of 75,000 systems within its first ten minutes (Symantec, 2007b). This is due to the method in which the payload was sent. As mentioned before this worm uses the UDP protocol in order to transport its payload to a randomly selected target. As UDP is a stateless protocol, the packets can be sent off and forgotten about as there is no reason to set up a connection and maintain it as is protocol with TCP connections. This vastly reduces the overheads in transmitting the worm's payload and also the supporting code to handle connection management expected by TCP, thus reducing its size to the aforementioned 376 bytes.

The intention behind this worm was little more than to slow down general system access to the Internet through a wide-spread DDoS attack that would see the Internet's bandwidth being allocated to transmission of this worm to other targets. This was due to the fact that the worm was set to retransmit its payload as fast as it could. Coupling this with infected systems that are allocated high bandwidth connections to the Internet, one would see a slow down as the Internet's infrastructure is now tasked with processing a vast amount of unnecessary network traffic.

2.5.6 Mirai (The Future)

The name Mirai is Japanese for "The Future" and is derived from the anime series *Mirai Nikki* (Future Diary) (Krebs, 2017). The languages this malware was written in was C [11] and GO [12]. The Mirai botnet, this is the botnet formed from hosts infected by the Mirai malware, was first discovered in August 2016 (New Jersey Cybersecurity and Communications Integration Cell, 2016).

[11] https://isocpp.org/
[12] https://golang.org/

The Mirai malware is simple in operation. It targets a running Linux-based OS using a list of commonly used usernames and passwords (this list also contains default usernames and passwords) in order to attempt login into the system. This means that Mirai simply attempts to gain access to a host that has a poorly chosen login credentials or default credentials. Once access is gained the malware then installs itself on the host.

What is particularly interesting about this malware is the inclusion of a subnet exclusion table within the malware itself. This exclusion table contains the subnets that the malware should not attempt to infect. A full list of these excluded subnets can be found in Listing E.

This exclusion was most likely included to prevent serious legal ramification due to the devices that this malware intended to infect. The Mirai malware is most interested in infecting devices that can be classified as part of the Internet of Things (IoT). This includes any Internet accessible devices such as IP cameras, smart TVs, Internet connected Blu-ray and DVD players (Paganini, 2016). If one of these IP cameras happened to be part of a military installation and confidential information was gathered from it, this would be a cause for a lot more man power being put into bringing this malware offline than originally predicted.

The lifespan of a host infected by Mirai malware is until reboot of the system. The malware itself does not implement any starting mechanisms for itself at the boot time of a host. However one should note that if the rebooted host has its credentials left unchanged it will most likely be reinfected soon after reboot has occurred.

Another point on infection of a host by Mirai is that upon initial infection the Mirai malware attempts to shutdown all other malware running on the host (Herzberg *et al.*, 2016). Furthermore, Mirai attempts to lock down ports 22 (SSH), 23 (telnet) and 80 (HTTP) to prohibit remote connection to the device and thus prevent remote reboot to clear the device[13].

As mentioned prior, the Mirai botnet has been used primarily in DDoS attacks around the world.

2.6 Indicators of Compromise (IOC)

Simply put, an indicators of compromise is an observed event that strongly indicates an intrusion on a system or host (Gragido, 2012). Some IOC are listed below:

[13]This is done by stopping all services that utilize these ports, and preventing allocation of services to these ports (Herzberg *et al.*, 2016).

Differing Hashes: Hash for original file differs from the file on a system (usually performed through MD5 (Rivest, 1992) or SHA1 (Eastlake 3rd and Jones, 2001) hashing).

File Inspection: If a hash does not exist for the original file, simple inspection for a sequence instructions known to be used for a malware in a file may indicate malware.

IP Address Inspection: Comparison of incoming and outgoing network traffic to known malicious IPs can strongly indicate an intrusion.

URL Inspection: Comparison of outgoing requests to an Uniform Resource Locator (URL) known to belong to a malicious entity is a strong indicator of intrusion.

Virus Signature Detection: A task typically performed by virus scanners, detection of a virus signatures on a system involves file inspection, registry inspection, as well as runtime resource (files, network, memory, etc.) request inspection.

Typically IOC leads to easier detection of re-occurrences of malware intrusion with the eventuality of complete prevention mechanisms being put in place. To help speed up the time taken from a malware first being detected to full system immunity, reporting standards have been put in place. The Incident Object Description Exchange Format seeks to standardize the format of an incident report allowing for easier cross system application development to counter system intrusion (Danyliw *et al.*, 2012).

2.7 DDoS Attack Detection Approaches

A DoS attack can be performed trivially through means of transmitting enough packets at a target IP address in order to fully saturate its network link. The more systems are involved in transmitting these packets, the less each needs to send in order to saturate the target's network link (Mirkovic and Reiher, 2004). Although there are smarter methods of denying a user access to a system on a network, such as causing a server to exceed its physical resource limits and thus disabling its ability to serve any more requests, the form described here through bandwidth saturation has a very easy-to-spot signature; this being more packets than usual destined to a network IP.

This form of attack is also easy to record and easy to replicate, and so makes a good starting point for implementing and testing anomaly detection systems, or to train potential network security candidates. Although it is difficult to detect whether a DDoS attack has malicious

intent or not without context, the ease of replication makes this form of attack a popular one to analyse and so there is much research and literature around detection and classification of this network event.

2.7.1 MULTOPS

MUlti-Level Tree for Online Packet Statistics, or MULTOPS for short, is an application that uses a tree of nodes to store packet rate statistics at a subnet level of the connections being made through the node in which MULTOPS in running on. This tree is designed to expand and contract during the runtime as to only contain the data relevant to detection of a Distributed Denial of Service attack (Gil and Poletto, 2001), this also makes it memory effective. Although outdated by today's means, at the time this research was conducted it was able to process in excess of 300,000 packets per second on a 700 Mhz Intel Pentium III.

This implementation is intended for application on routers or packet monitoring devices which have low CPU and memory resources in comparison to that of fully fledged computers. The device running MULTOPS would then gain the ability to be able to detect fluctuations in bandwidth requirements to a destination and terminate the connection. This termination is implemented through simply dropping all subsequent packets from the source that are involved in the connection destined to the affected destination address.

2.7.2 Learning through Neural Network using sFlow as Learning Data

The approach taken through the combined efforts of Ewha Woman's University and the Electronics and Telecomunications Research Institute, both situated in South Korea, show promise into DDoS attack detection using flow records as training input for a Distributed Denial of Service detecting neural network (Rochester *et al.*, 1956). Their work looks into using sFlow[14], a competing standard against NetFlow, to monitor networks' connections and generate the flow records for the neural network (Kim *et al.*, 2004).

The research performed by Ewha Woman's University and the Electronics and Telecomuunications Research Institute used data records known to be involved with a DDoS attack as well as data records known to not be involved in a DDoS attack. This was then given to the neural network in order for it to train and feedback could be given directly from the known results of

[14]http://www.sflow.org/

the training set. After this, the neural network was used in an attempt to accurately determine whether unknown sFlow data records were part of a DDoS attack or not. The error rate of this neural network was rated at 2.898%.

2.7.3 Attack Classification through Use of NetFlow

The Samara State Aerospace University of Russia takes the approach of classifying attacks through controlled environments and then analysing the flows that are generated from such attacks. The research started out primarily targeted at the successful consistent detection of single source IP DoS attacks. Once it was confirmed that detection of a DoS attack could be performed this research scaled out and reapplied the DoS detection mechanism to detection of DDoS attacks (Galtsev and Sukhov, 2011).

This research then continued to build on its detection engine to include the ability to detect port scans which is usually an indicator of a potential attacker looking for a vulnerability in the services hosted on a system. This research then proposed a locking system that if a DoS, DDoS or port scan is detected, would then lock out all subsequent traffic from that source until conditions are met to unlock the IP. These conditions include no inbound traffic for a set period of time from that source or a drastic change in behavioural patterns that suggest the IP has been reallocated.

This research has continued to develop the original detection engine to extend into the detection of known worms such as W32.Blaster (Dougherty *et al.*, 2003) and Red Worm (Microsoft, 2003a) and classification of these worms traffic has been performed. Following these results they have stated that there is promise in the ability to classify these worms through the use of NetFlow logs.

2.7.4 TOPAZ

The Traffic flOw and Packet Analysis System, or TOPAZ for short, places its focus on real time event analysis. Attacks that occur on systems within networks should be detected and mitigated as fast as possible and this is exactly what this research aims to achieve. The generated flow data (NetFlow v9 is used in this implementation) is fed directly into a preprocessor that identifies key attributes from within the NetFlow data records. This key data is then passed on to a separate system that is dedicated to the running of this research's detection algorithm that uses these key attributes to successfully identify and analyse network traffic (Munz and Carle, 2007).

The system is also designed to be smart enough to identify flow data that warrants concern through its classification engine. This further frees up resources to allow the host system to respond to a threat quickly at a hardware level. The TOPAZ system also allows for adaptation by allowing filtering of specific protocols. This allows TOPAZ to target a specific search range according to what anomalies one is trying to detect. This again speeds up the system as it deals with less data.

Incoming NetFlow data is divided up into three main categories in order to help speed up TOPAZ's filtering algorithms. These categories are listed below:

Short term flows: This category is reserved for flows that have been newly reported; flows that have been seen for the first time.

Long term flows: This category contains flows that have been seen in records more than once and consistently appear as records in consecutive NetFlow logs.

Sporadic flows: This category contains flows that have been seen in records more than once but do not consistently appear as records in consecutive NetFlow logs.

These categories are used against known characteristics of network events that exhibit short term, long term or sporadic characteristics. This helps filter out what the flow is definitely not, based on what can be expected in each category, thus further speeding up TOPAZ's analysis engine.

2.7.5 Statistical Approach to DDoS Detection

The Boeing Company and Network Associates Laboratories propose the use of statistical analysis through formulaic algorithms and mathematical modelling to detect malicious attacks on networks. This is performed through gaining samples of the network under standard working conditions and storing it for comparison at a later point. This analysis is performed through observation of timings of each network IP passing through a node that generates the flow data for analysis (Feinstein *et al.*, 2003).

After the results from standard working conditions are produced, comparisons can be made to detect behaviours that are not common on the network. At runtime, standard operation of this system allows for adaptability through aggregation of further network results. This allows for a running average to better adapt slight changes in the network over its lifetime. This system also incorporates a time of day detection system that allows for fitting network characteristics based on a time of day, week, or month to allow for further accuracy and fidelity in attack detection.

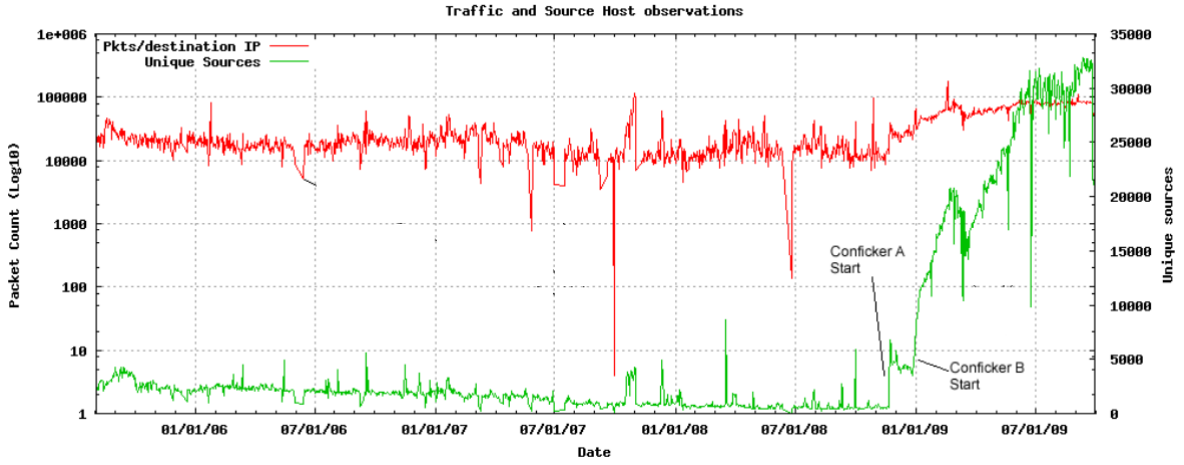

Figure 2.8: Network Telescope Traffic Increase at Conficker Start (Irwin, 2011)

Basic testing of this system is through detection of DDoS attacks and loss of connectivity, this being the other extreme, in order to ensure the system runs as expected. The system achieves detection of both cases successfully.

2.8 Internet Background Radiation

This name refers to a known network address space that is unallocated which is purposefully observed to see what traffic is destined to this address space. Because this address space is unallocated, all traffic that is destined to it is inherently suspicious. Other names for this network address space used in this way includes black hole, darknet and Internet motion sensor (Harrop and Armitage, 2005).

Suspicious network traffic that is destined to these network telescopes is mostly attributed to random port scans, DDoS backscatter and misconfigured systems. These port scans are due to malware, misconfigured software/hardware, or users with malicious intent scanning ranges of IPs searching for vulnerable Internet facing IPs. Network traffic can also arrive to this network space due to legitimate responses to spoofed IP sources (Pang *et al.*, 2004). This is common when considering seemingly legitimate TCP SYN packets or attempts to access a closed network port on a system; these would both warrant a response to the source IP in the request.

Major malware events have been detected and recorded with this method of Internet observation. As an example, during the peak of the Conficker Worm Barry Irwin's research observed and recorded finger printing techniques used by the Conficker Worm in its attempts to locate

new targets on the Internet. His work shows this through depiction of spikes of fingerprinting network traffic received at his network telescopes at the times of which new versions of the Conficker Worm were released. The start of the first version of conficker can be observed in Figure 2.8 at point "Conficker A Start" and the start of Conficker B can also be observed on the same figure.

Furthermore, he shows the result of a bug that was introduced into the worm in which the most significant bit of the destination IP address used by the Conficker worm is never set. This is due to the fact that 32-bit integers were used in the random generation of destination IP addresses by Conficker. However, the integer was signed thus omitting the highest bit for a sign bit. This led to the destination IP addresses used by the Conficker Worm to be between 0.0.0.0 and 127.255.255.255 (Irwin, 2011, 2012, 2013).

2.9 Packet Source IP Geolocation and Dealing with Spoofing

IP addresses on the Internet are assigned by blocks called subnets. These subnets are allocated to LIRs in the world, and thus one can use the source IP address of a network flow in order to place where a network flow originated from in the world through IP address comparison to these registries (Housely *et al.*, 2013). This information can be used to regulate network connections through generation of firewall rules in order to allow, prevent or modify connections to certain IP addresses (King, 2010).

Blocking of an IP address or range of IP addresses may seem like an appropriate way in which to mitigate an attack from a known source IP address. However, it is not (Johnson *et al.*, 2007). Commodity PCs can generate network packets in which all bytes within it are defined by the user. This is referred to as packet spoofing, and is performed by modification of the IP source address bytes of a network packet to represent that of an IP address not belonging to that system. Furthermore, one can randomly generate these address for every packet that system transmits. This means that one cannot just block an IP address or set of IP addresses and be done with it; this is a very tedious problem to solve. Spoofing in this manner is typically used in the spectrum of DoS attacks (Needham, 1993).

There are defensive mechanisms in place for dealing with attacks that use IP address spoofing. The first and most inconvenient is to change one's systems static IP address/addresses. If the malicious system is set to attack a specified IP address, or works on the system of resolve a FQDN once and then target the given IP address, then this would allow the target to mitigate

this form of implemented attack (Douligeris and Mitrokotsa, 2004). This will work until the attacking system updates the IP in which it is targeting through administration, or by resolving the target FQDN again.

Simply knowing who should be connecting to a system can be invaluable information to prevent random IP source address attacks. Creation of a whitelist containing only the subnets and IP addresses that should have access to your system is a method that can filter out a lot of unwanted connections. This also makes sense in business planning as there are many businesses that only do business with a set part of the world (Srivastava and Giffin, 2008). This means that one can include these subnet whitelists into the business model of one's company.

Another consideration one should make is for stateless protocols such as ICMP (Postel, 1981b) or UDP (Postel, 1980). Stateless protocols like this do not require a connection to be set up beforehand, and thus allows an attacker to generate packets and send them at a target with no overhead required on the attackers side.

If one does not have a deep understanding of how networks work, such as a small company that can not afford on-site system administrators, one can employ a third-party company in which one's service is relayed through. This third-party company, like CloudFlare (CloudFlare, Inc., 2016), can provide mitigation of malicious network activity on your behalf in order to keep one's services available and secure.

2.9.1 Use of NetFlow Node Source ID in Packet Spoofing Detection

As part of the NetFlow version 9 and IPFIX protocols (Cisco, 2003a), each NetFlow source node is allocated an ID number. If one were to ensure these IDs were unique and keep a record of where each node is physically located, one would now have a method of detecting where certain types of network traffic are sourced from, be it within a private network or a public network, without the need for an IP address (Cisco, 2003b). This is useful in locating where a network flow is physically sourced from when considering malicious activity that uses IP address spoofing as discussed in Section 2.9.

To elaborate with an example, consider a simple DoS attack on a target system. If the DoS attack was created in such a way that it spoofed traffic with sources from around the world, in order to prevent communications from the attack one would have to effectively block communication from the world; not very useful at all. Now if one had access to NetFlow sources from around the world and knew which NetFlow source ID numbers were allocated to each specific location in

which a NetFlow source node was placed, one could use this information to better mitigate this example attack. From the logs generated from these NetFlow source nodes, one could identify which flows were destined to their service. Coupling this information with the NetFlow source ID number which each NetFlow log is received with, and assuming one can detect which logged flows are associated with the DoS attack, one could tell which locations in the world the attack is being generated from. From this point one could perform more effective attack mitigation.

2.10 Botnet Detection

In more detail, a botnet is a name given to a set of systems infected by a malware or set of malware that receives remote commands in order to perform a task which is not authorised by the owner of the system. Bots within a botnet are usually aimed at completing multi-host tasks such as DDoS attacks or proxying of traffic through multiple hops.

Infection of a system can occur like any other malware and there is no special method in which a system can be infected and included in a botnet. One can use methods detailed in Section 2.4 to infect a vulnerable system and execute code that includes the system in a botnet. The affect it has on the infected system varies depending on the infection/infections. The effects range from unnoticeable, which allows for a longer infection time, to full system owner lockout, which usually results in shorter infection time. Some operations an attacker can perform on an infected system are listed below:

DDoS Source: Use of the infected host as a generator of network traffic for inclusion in a DDoS attack.

Key Logger: Logging of key strokes on a host to collect information about a user. This can include usernames, passwords, emails, banking details and any other person information (Matalytski, 2006).

Ransomware: Software used to lockout a user from a host and information on it until they have paid for access rights back to their host and data (O'Gorman and McDonald, 2012).

Spam Bot: Use of a host to send out advertisement material usually in the form of emails (Chiang and Lloyd, 2007).

The affect on networks however is widespread, be it from the point of development or general runtime of the network. To explain, systems have to be developed to deal with malicious request

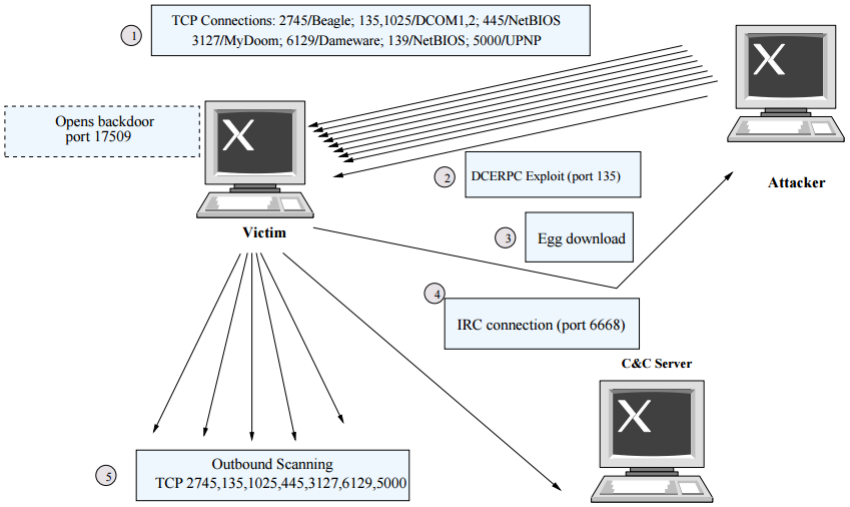

Figure 2.9: Bot Attack Process on Vulnerable System (Gu *et al.*, 2007)

as well as have maintenance team assigned to the system after it goes live for further errors that may occur. This stretches the budget and time of the entity that employs and owns the software. In terms of general network runtime, bandwidth allocated to attacks such as DDoS, or the requirement for network packets to stay on a network longer while it gets proxied through multiple hops, decreases the available bandwidth on the network; this affects all users as a whole.

2.10.1 BotHunter

The perimeter of any system or network is a strategic point in which to implement a protection system. This location allows for monitoring of any inbound and outbound traffic that involves the protected system. BotHunter takes advantage of this location to detect and mitigate attempted intrusion into the protected system. This is performed through analysis of flows generated by network traffic that passes through the node on which BotHunter is running (Gu *et al.*, 2007).

The BotHunter system makes use of the network intrusion detection tool Snort (Roesch, 1999) to apply a detection rule set to traffic passing through the perimeter based monitoring node. Snort is a network monitoring system and further information on this system can be referred to in Subsection 2.11.1. The rule set is based on a five step intrusion model which was discovered in this research and was found to be used by most exploits in order to penetrate a system. This process is provided with a step by step visual aid as seen in Figure 2.9.

1. **External to internal port scan:** This is a port scan from an outside network attempting to fingerprint a host or multiple hosts on the internal network.

2. **External to internal exploit based on scan report:** This is an attempt to execute code remotely via exploitation based on information gained from the external to internal port scan.

3. **Internal to external binary acquisition:** This stage is the response to the remotely executed code that gets the now exploited host to request additional files and resources in order for the exploited host to perform specific tasks that the attacker wishes.

4. **Internal to external command and control communications:** This stage occurs once the attacker has fully uploaded all of the malware's functionality onto the exploited host, thus letting the exploited host run as a stand alone infected system. At this point the infected host requests directives from a controlling host in order to achieve the task of the attacker.

5. **Internal to external port scan:** In most cases this research found that after a system was fully infected it would attempt to infect other systems, thus growing the botnet autonomously. The point to note here is that now the port scan comes from the infected host and thus originates at the internal network.

BotHunter then uses the analysis tools provided by Snort along with the rule sets provided to it to generate confidence ratings as to the likelihood that an incoming connection from an external host, or an outgoing connection from an internal host is due to that of inclusion or attempted inclusion in a botnet. The information then presented to the user on successful detections includes the victim's IP, the attacker's IP and logs for evidence trails leading up to the infection, or showing that a system is infected.

2.11 Attack Detection Through Packet Analysis

The idea of using the presence and general characteristics of a connection, rather than the information sent via the connection's payload, to detect malicious attacks has its advantages and disadvantages. The most obvious advantage is the increased volume of traffic that can be processed, as major parts of a packet can now be ignored. However, this also leads to the greatest disadvantage, which is that in the event of an attack, one has no means by which to accurately detect what parts of the system have actually been affected if that information is contained

within the ignored segments of the communication. This disadvantage does come with a reprieve as an attacker usually aims at common parts of a system (this being payroll systems, private information, private electronic property, among other resources).

Other than the requirement for more processing power required to perform full packet analysis on a connection, one also needs to consider privacy laws. One such law that seeks to protect users privacy is the Protection of Private Information (PoPI) Act (Korb, 2013). As not all private information communicated on a network is encrypted, and thus unreadable, full packet analysis on such a connection could lead to a breach in user privacy. For this reason user privacy must also be taken into consideration when performing full packet analysis.

The decision to use full packet or partial packet analysis should be weighed up against the number of attackable underlying systems within the system, and the time taken to check through these packets versus the quality of service one can provide to clients when full or partial packet analysis is performed at the time of the security system's inception.

2.11.1 Snort

Snort is an intrusion-detection system first developed by Martin Roesch, who later founded the company Sourcefire which is now in charge of Snort's development. Snort can perform protocol analysis, content searching and content matching through use of search trees. These services, coupled with Snort's event trigger capabilities, allow it to ensure quality-of-service and react to events such as intrusions or system failures (Roesch, 1999).

Snort is intended to be configured into 1 of 3 main operating modes: packet sniffer, packet logging or network intrusion detection. Packet sniffer and logging modes are similar in that both read packets from the Network Interface Controller (NIC) and then either display them to console or write them to disk respectively. In intrusion detection mode, Snort is allowed to perform specific actions based on what has been detected from connections made through Snort's host. Triggers can be attached to these intrusion events and when an intrusion is detected, Snort can perform tasks to mitigate the damages of the intrusion. These tasks include notifying system administrators, shutting down connections or just monitoring the link in further detail to allow for in depth analysis of the attack for later use; be it for recovery or new defensive mechanisms.

Pixel Snort is an implementation of Snort that offloads tasks typically performed on the CPU to a GPU in order to leverage increased performance from Snort. This implementation of Snort makes use of the GPU's stream processor capable functionality in order to perform major parts

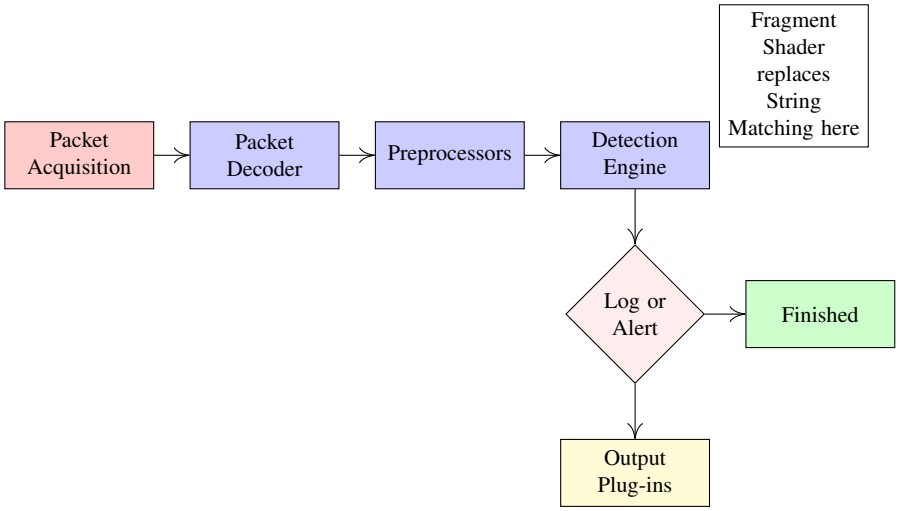

Figure 2.10: Logic Flow of Pixel Snort

of string matching that Snort would normally do on the CPU (Jacob and Brodley, 2006). Motivation for this approach was based on the fact that an estimated third of all processing performed by Snort is string matching, this is noted in Figure 2.10. A further benefit is that while the offloaded task is being performed by the GPU, this would free up the CPU to process something else in parallel, or devote more time to other subsystems of Snort.

Through extensive testing, ranging from testing the processing of expected network traffic to traffic crafted specifically to cause cache misses at the CPU level, this system was found to yield up to a 40% improvement over the conventional CPU-based Snort implementation. Results reflect lower CPU usage, less dropped packets and less time taken to perform a string match on an incoming packet. However, there are points within testing where GPU performance tends towards CPU performance. These points are during low load and high load. At low load, this performance similarity is most likely due to the fact that both of these systems have excess resources in order to process the given task, and so can both do it in the minimum time required.

When one considers the other end of the spectrum, the case of high load, one can most likely account the GPU's performance to that of bottlenecks within the system. The two major bottlenecks within a system is that the CPU still has to delegate tasks to the GPU, and the memory bandwidth restriction imposed by the Peripheral Component Interconnect express (PCIe) bus. Either of these bottlenecks, or combination of both, would result in the GPU not being allowed to fully saturate its hardware resources and thus a reduction in packets processed would occur.

2.11.2 Bro

Like Snort, Bro can be described as an intrusion-detection system. Bro was originally developed by Vern Paxson and is licensed under the BSD license. This software can be divided into two major underlying systems, these being the event engine and policy scripts. The event engine's purpose is to analyse or record live traffic and generate neutral events. From here, the policy scripts take the generated neutral events and analyse them in search of anomalies or specific characteristics. The Bro scripting language that is used to analyse these neutral events generated by the event engine is also Turing complete (Paxson, 1999).

The Turing complete nature of this language makes it more powerful. A system is said to be Turing complete or computationally universal if it has the ability to simulate any single-taped Turing machine (Hodges, 1983). In other words, the Bro scripting language can be used to process data and perform tasks of any other Turing complete language such as C++ (Stroustrup, 1986) or Java (Oracle, 2015).

2.11.3 System for Internet-Level Knowledge

The SiLK toolkit is a collection of software for network traffic collection and analysis. This toolkit developed and are currently maintained by the Computer Emergency Response Teams (CERT) Network Situational Awareness Team (CERT NetSA) of the Carnegie Mellon University. The majority of these tools are implemented in C, Perl and Python and are compatible with major UNIX-like (The Open Group, 2012) operating systems such as Linux[15], Solaris[16], OpenBSD[17], Mac OS X[18] and Cygwin[19].

These tools base their data collection upon IPFIX, NetFlow version 9 and NetFlow version 5. The information within these records are then converted into a known space efficient format and recorded into service-specific binary files. The analysis tools then read in these binary files and perform analysis on this collected data, in order detect anomalies or specific characteristics of the network traffic on the recorded network[20].

[15] https://www.linux.com/
[16] www.oracle.com/Solaris
[17] http://www.openbsd.org/
[18] https://www.apple.com/za/osx/
[19] https://www.cygwin.com/
[20] https://tools.netsa.cert.org/silk/

Table 2.2: NAND Gate Look-Up Table

Input A	Input B	Output
0	0	1
0	1	1
1	0	1
1	1	0

2.12 Field-Programmable Gate Array

The FPGA is an Integrated Circuit (IC) used to simulate and help develop ASICs. This simulation is performed through population and linking of fields within the FPGA to simulate basic logic blocks (such as AND, OR, NAND, XOR, etc) with which one builds up the basic components of the described hardware (Francis *et al.*, 1992). The general inputs and outputs of the described hardware for the most part can be bound to any pin that is marked as an input or output on the used FPGA's package. It is only when specific functionality is required, such as clock input and output pins or differential pairs, that the pin mapping of the described hardware has to be specifically bound to certain pins within the FPGA package used (Xilinx, 2009).

A language used to describe the data flow and logic which the FPGA simulates is referred to as a Hardware Description Language (HDL). Of these languages the most commonly used for development on both Intel FPGAs and Xilinx platforms is VHDL (Computer Society of the IEEE, 1988). This language was developed for the United States Department of Defence as a means to record the logic that was being used in their equipment (Doulos, 2014). As this language was originally developed to document existing ICs, this language can also create them. This makes VHDL a powerful tool as one can first simulate the hardware's logic on an FPGA. Once all errors have been corrected one can then get the described logic fabricated into an Application Specific Integrated Circuit (ASIC).

This also reduces costs in production significantly, as an FPGA can be reprogrammed multiple times. Because of this characteristic, it also allows for rapid iteration development and thus saves on development times, as full fabrication of an ASIC can have up to a 6 month lead time before fabrication begins (Kuon and Rose, 2007).

2.12.1 How It Works

To further explain the inner workings of an FPGA and how the logic blocks are implemented, one needs to first understand what a Look-Up Table (LUT) is and how it works. As one may

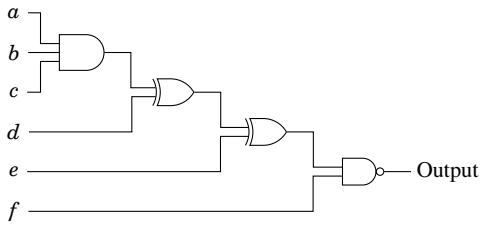

Figure 2.11: Example 6 Input Gate Logic

derive the functionality from the name, the LUT takes an input to look-up a value within a table. This looked up value is then presented as an output from the table (National Instruments, 2015); exactly like a logic table.

The LUT gives the basic building block required to form a logic operation within an FPGA (Enderton and Enderton, 2001). To elaborate, consider the humble NAND gate represented in Table 2.2. A simple look-up table to implement a NAND gate would to simply take in two single bit inputs, input A and input B, and produce a single bit output, as represented in Table 2.2. When input A and input B are presented to the look-up table and its output is enabled, the LUT would then use these inputs to reference the output value and then produce the output bit on the output line from LUT.

In a Xilinx Spartan 6 a LUT allows for up to 6 input values, the means in which this is implemented will be discussed in detail later (Xilinx, 2014). Allowing for this number of inputs into a LUT means that either one can perform larger single logical unit implementations, such as a XOR with 6 inputs, or represent multiple logic units in a single LUT. For example, the logic depicted in Figure 2.11 can be represented by a 6 input LUT as shown in Appendix B.

If one wanted to build a functional system which consisted of more than 1 to 5 logic gates one could look towards combining multiple LUTs within an FPGA in order to achieve this task. Using the outputs of multiple LUTs' allows one to build up essential logical units used in modern computing, such as an adder or even a divider (Parhami, 2009).

It is notable that most of the logic gates used in these aforementioned arithmetic logic only consists of 2 inputs, so why does one need a 6 input LUT. Allowing for a LUT of this size is only beneficial to the users experience. The reason for this is that a look-up in a LUT takes time as the electrons have to propagate through the LUT's logic, and as one connects more and more LUTs together, the time needed to perform the logical operation and produce a result increases. This directly affects the throughput of the logic in the FPGA as the longer it takes to traverse the logic simulated in the FPGA, the less results can be produced in a given time period. To

counteract this one can use the method described above of simulating multiple logic gates in a single LUT in order to reduce the number of LUTs used to produce the logic of a system, thus reducing time taken and maximizing the systems throughput (Omana *et al.*, 2003).

These connections between LUTs are less simple than stating them as wired together, and this further adds to slow downs within the device when using multiple LUTs. Each output needs to be buffered somewhere so it can be used as an input, and this is typically done by a flip-flop (a single bit of memory). The lesser used memory store used to buffer these LUT outputs are latches, these are slower and lead to timing problems within the FPGA when searching for maximal throughput (Embedded Micro, 2015).

2.13 Summary

The topics discussed in this chapter were based on the outline stipulated by this research's goals defined in **Section 1.2**. This chapter started off in **Section 2.1** through definition of what Ethernet is and some details about its workings. As the Internet Protocol is built on top of Ethernet, it logically followed that this protocol is discussed next in **Section 2.2**. **Section 2.3**, then discusses the NetFlow protocol which is used for all data input into the system proposed by this research. Understanding of Ethernet, Internet Protocol and NetFlow is a requirement before design of this system can even begin in **Chapter 3**.

Sections 2.4 through **2.11** seek a better understanding of what malicious activities exist on the Internet and how one can approach mitigating such events. These sections start through explaining what malicious activities are on the Internet by explaining the different classes of these activities, and what characterizes each. With this understanding the text moves into looking at well researched malicious attacks, their behaviours and history to deepen the readers knowledge on this topic. Drawing from this knowledge, prevention methods for the discussed attacks are brought to light and shown where and how they can be applied to counter these malicious activities and protect a network. This knowledge is applied in **Chapter 5** when this system attempts to provided new countermeasures to these malicious activities on the Internet.

This chapter finally draws to a close with an introduction to logic gate level circuit design in **Section 2.12**. This knowledge is then applied in the hardware acceleration of this research in **Chapter 8**.

3

Technology Evaluation

THIS chapter evaluates existing technologies that were considered for use in the implementation of Bolvedere's base system. The base system in this implementation is the part of Bolvedere that processes and distributes NetFlow records for processing. The system will be expanded on in **Chapter 4**.

This chapter starts with **Section 3.1**, which discusses existing hardware architectures, the type of data each is suited to process, and then relates this ability to NetFlow records directly. Following this, **Section 3.2** brings forward methods of inter- and intra-process communication due to the possible requirement for multiple subsystems to communicate within Bolvedere in order to form the system as a whole. Introduction of inter- and intra-process communication is also required if scalability is implemented in terms of concurrency, as this will most likely involve communications between the subsystems within Bolvedere. This chapter concludes with a summary in **Section 3.3**.

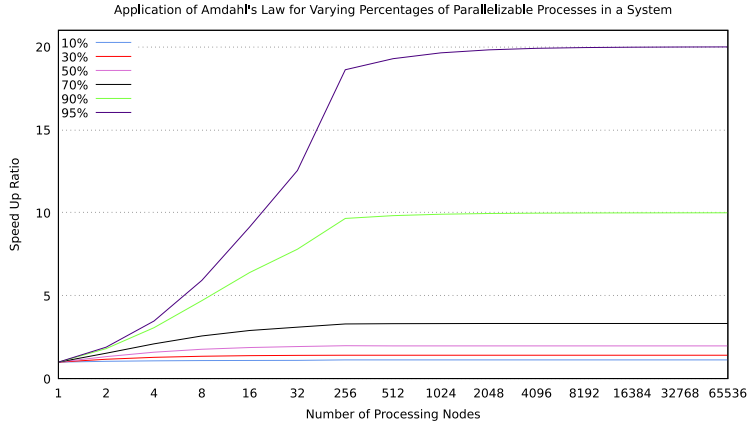

Figure 3.1: Application of Amdahl's Law

3.1 Architectures

A primary goal defined for this system (detailed in Section 1.2) was scalability; however one should take care regarding the manner in which one scales a system. Amdahl's Law shows that increasing the number of processing nodes within a system is not the most effective method of increasing the system's throughput (Amdahl, 2007). Using Amdahl's formula in Equation 3.1, one can calculate the results for the application of Amdahl's Law in Figure 3.1. One can see from these results that even a system that can run 95% of its processes in parallel can only achieve an approximate 20 times speed-up. In addition to this is the fact that to achieve this 20 times speed up, one would require 4,096 processing nodes.

$$SpeedUp(P,S) = \frac{1}{(1-P)+\frac{P}{S}}, \quad \text{where P is the parallelizable percentage of the system and S is the number of processing nodes in the system} \tag{3.1}$$

The cost-to-performance ratio per processing node at the higher end of this spectrum strongly serves to call into question the addition of processing nodes (horizontal scale-out) to the system in order to increase processing speed. For this reason, one must give consideration to selecting the proper tools for the job before implementing a system, as one should not rely on scalability to solve performance issues within a poorly optimized system. Instead, one should use scalability to improve the performance of a well-implemented system, further increasing throughput.

With this in mind, one should consider the best tool for the job, and then scale that implementation. Choosing this best tool is a problem in itself, and to best solve it, one should consider the strengths and weaknesses of currently existing architectures.

3.1.1 CPU

Since mid-2009, AMD and Intel have held over 99% of the global market shares for PC processor manufacturers (Statista, 2015). The commodity processors that both these companies commonly manufacture follow the Single Instruction Single Data (SISD) instruction sets x86 (Intel Corperation, 2015), x64 (Intel Corperation, 2015) and IA64[1] (Intel Corperation, 2010). A SISD instruction set refers to a hardware architecture that performs a single operation upon a single memory location (Michael, 2004). These instruction sets are also designed around a Reduced Instruction Set Computer (RISC) architecture in order to keep instruction simple (Kane and Heinrich, 1992). These simple instructions allow for a relatively small hardware footprint. This small footprint reduces the time taken for an electron to propagate through the relevant hardware, and so more data can be put through that hardware. This is coupled with the fact that fewer instructions mean faster instruction decode times, and so more instructions can be fed into the processing logic.

Optimization through the use of an instruction pipeline also benefits from the use of a SISD-based instruction set using RISC architecture (Quinn, 2004). As there is only ever one memory location and one instruction in consideration at a time, coupled with the instructions available being well-defined and few, it is easier to break down these instructions into common steps (micro-operations). The hardware to perform these common steps is then allocated to sequential instructions as the instruction enters the common step.

If one were to consider this form of logic for a single instruction, one would soon note that, if the instruction was only using a singular part of this common hardware at a time, the rest of the hardware would remain idle while the single part is operating. However, if one were to allow the next instruction to use the hardware just released by the previous instruction as the instruction moved onto the next common step, one could repeat this logic to include multiple instructions at different stages of execution at one time. This is what is referred to earlier as pipelining (Murakami *et al.*, 1989). As an example, consider an instruction set where the instructions share these common features:

1. Decode Instruction
2. Load Data
3. Execute Instruction
4. Store Result

[1] Itanium family of processors (https://www.intel.com/content/www/us/en/products/processors/itanium.html).

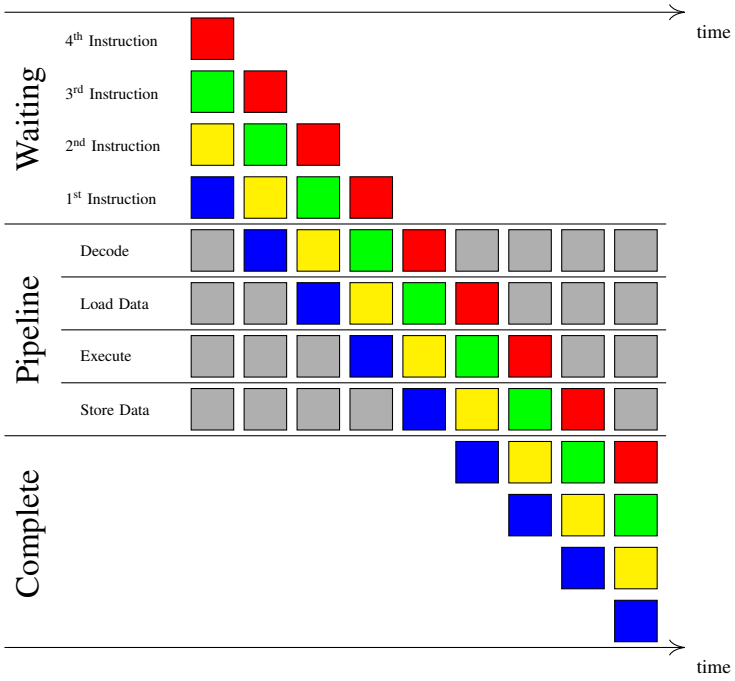

Figure 3.2: Example Pipeline Stages of Execution

In this case one can accept four instructions into the CPU at once with a pipeline that involves these listed stages. This is because the first instruction loaded into the CPU would first be decoded as to what the instruction should do, and would then need to fetch its relative data. Instead of letting the decode instruction hardware go idle, one could then let the next instruction be decoded. To continue this logic, when the first instruction is now being executed on the loaded data, the second instruction can have its data loaded and a new instruction can begin its decoding phase. If there are 4 or more instructions to be executed through this pipeline, other than the initial 3 instructions entering this pipeline, there would not be any idle hardware within the CPU (Hennessy and Patterson, 2011). This example is diagrammed in Figure 3.2.

One must also note that one cannot compute an instruction if one does not know what it is, or what data it is working on. For this reason, the use of a pipeline allows the use of hardware that would otherwise be stalling while waiting for the relevant resources. Another point one should consider is that this form of hardware further reduces the propagation delays of electrons, as there is even less hardware for electrons to pass through during a micro-operation (Boyes, 2002). This allows a CPU to yield higher clock speeds without the concern of clock skew[2].

Within Bolvedere, the use of a sequential processing device, such as a CPU, should not be

[2]Clock skew is the phenomenon in which a signal source arrives at different components at different times. This is more common in larger circuits.

considered as an architecture for NetFlow record discernment. Although discernment on a CPU is possible, this operation deals with memory accesses that cannot be determined before the arrival of NetFlow data and template records. When dealing with incorrect branch prediction, this introduces overheads, which lead to process stalls (Yeh and Patt, 1991). Consider the typical lifetime of a NetFlow data packet as it enters a NetFlow processing system: when it first enters this system, the template identifier has to be fetched from the NetFlow data record. This is a simple and predictable operation.

From this point the process becomes indeterministic, as one has to look up each field before reading the data from the NetFlow data packet and processing the data accordingly. This is a task that cannot be streamlined, as one does not know what NetFlow templates will be received before the runtime of this system. This makes the optimisation of a NetFlow processor for a sequential architecture such as a CPU difficult.

The converse to this indeterministic event is when data arrives in a known order. This means that there will be no overhead in discerning what a certain field is within the received NetFlow data record, as the data will always arrive in the same form. As this form is known, the data can just be processed. This concept will be further built upon in Section 4.2.1.

3.1.2 GPU

Graphical Processing Units (GPU) are specifically designed to deal with large datasets using a Single Instruction Multiple Data (SIMD) instruction set (Cockshott and Renfrew, 2004). The reason for this is that they are primarily used for graphics processing (as the name suggests), which entails placement and texturization of objects on a display (Sanford, 2007). Typically, a graphic object is made of a model or a sprite. These objects often have the same operations performed on them, be it a transform of the actual model or sprite, a texture placement or shading. For this reason, having the ability to perform the same operation on multiple objects at once speeds up the overall throughput of the graphic system (Patterson and Hennessey, 1998). A high-level side-by-side comparison between a SISD and a SIMD addition instruction can be referred to in Figure 3.3

To better understand the computational ability of a SIMD instruction set when compared to a SISD instruction set, consider a room full of cups in 3D space with a singular lighting point. The cups' response to the light is to reflect it, making a seemingly smooth and shiny surface. The math is simply represented by knowledge of the light, where a cup is located, the angle of the cup's surface to the light and the position of the camera. Knowing this, one can tell which

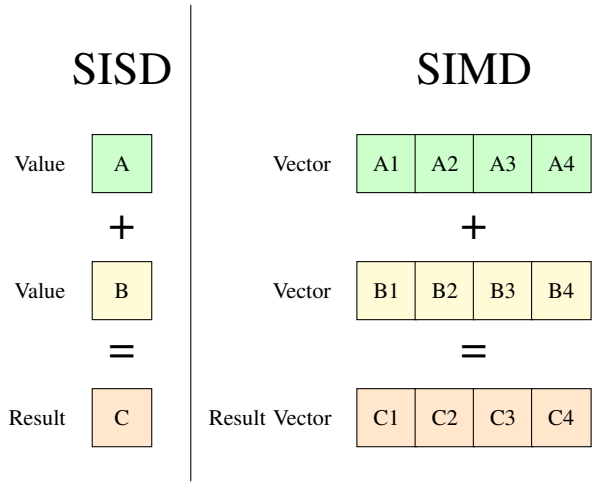

Figure 3.3: SISD and SIMD Addition Comparison

part of the cup reflects light towards the camera and which doesn't, and can shade the cup's surface appropriately from this point. However, doing this same calculation for every cup and every surface on the cup is repetitive and resource intensive under a SISD architecture.

However, under a SIMD architecture, one can load the location of multiple cups and perform the same calculation on each, based on the location of the light and the camera in parallel. This allows one to process repetitive calculations quickly and in a more effective manner, thus taking less time to complete and increasing the throughput of the system as a whole (Patterson and Hennessey, 1998).

General Purpose Graphical Processing Unit (GPGPU) refers to a method of using a GPU's graphical instruction set for performing more general calculations in parallel (Lee *et al.*, 2009). One could consider a water simulation as an example for this form of computation. Even though water simulation has nothing to do with graphics, one can use a GPU's SIMD architecture when considering each point within the simulation. One can load every point within the simulation into a GPU, along with the relevant calculation, and process multiple points at once for the simulation, as it is a repetitive calculation. This allows one to increase the rate at which the simulated information is processed, and thus allows for a higher throughput of data within the simulation. This leads to allowing for a more detailed or larger simulation, as one can now better deal with a larger collection of data within the simulation.

Although using a GPU in a GPGPU context is extremely effective, GPUs using CUDA do have a failing point, seen when delving into the inner mechanisms of their architecture. A process that is compiled into instructions for a GPU is called a kernel and is run in lockstep. Lockstep is a process by which every core inside a GPU assigned to run a kernel has to be at the same

stage of execution as every other core running that assigned kernel. This makes programming conditional statements a difficult task to achieve (Pai *et al.*, 2013). Because of this, GPUs are programmed to complete many repetitive calculations. These results are then passed to another system, usually running on attached hardware, to apply conditions to them in a GPGPU context.

In short, GPUs perform well when a set path of logic can be defined; follow steps A to B and never stray, just yield the results. Because of this logic, designing a system with which a user can interact becomes very tricky, and quickly turns one away from using GPGPU as the tool for tasks such as designing and running of a user interface. This is not to say that GPUs are useless in this regard - GPUs can aid in displaying or performing complex calculations onto user input.

For this reason, a GPU's best fit is in the domain of heavy, repetitive calculation (and thus processing), and not in the domain of NetFlow discernment. This is due to the lockstep nature of GPUs, rendering them not able to support multiple paths of execution concurrently in a single kernel (Pai *et al.*, 2013; Collingbourne *et al.*, 2013).

3.1.3 FPGA

As discussed in Section 2.12, an FPGA is used to aid design and implementation of ICs through the emulated hardware environment it provides. This means one can design hardware specifically to solve a problem, and is referred to as hardware acceleration (Oxford English Dictionary, 2015). The two general paths which hardware acceleration implementation take in order to increase the throughput of a system is to either parallelize the problem, or to create new hardware logic that is designed specifically to perform the required computation for the problem (Zemcik, 2002).

This versatility allows for the FPGA to be used at any point of this platform; however, an FPGA would be most important at the point of NetFlow collection. The reason for this is that NetFlow record discernment requires random access to field orderings as determined by a NetFlow template (Claise, 2004).

Dedicated hardware in this situation could buffer the entire NetFlow data packet into memory (Kumar *et al.*, 2002). Once buffered, the use of dedicated hardware allows the program to look up the template associated with the NetFlow data packet, discern the entire NetFlow packet according to this matching template, and then place the fields in a known order for a CPU or GPU to further process. This would remove the indeterministic process from a CPU or GPU, and allowing for programs to be written for them that can be optimized for a specific data form.

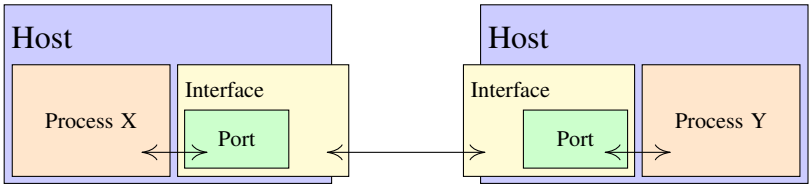

Figure 3.4: Multi-Host IPC Using Point-to-Point Communications

This transformation can be done in parallel within a hardware device designed specifically for this task. Furthermore, any special operations required for look-ups, comparisons or copying of data can have specific hardware designed to complete them too.

Using each of these architectures at different stages of this system to solve specific problems for which each was designed would yield the best implementation of this system. Furthermore, each technology brought to light in this section can be designed to run at scale, and thus further increase performance. One now needs to define the best manner in which each of these subsystems communicate, to prevent bottle-necking from occurring.

3.2 Inter-Process Communication (IPC)

Systems are usually made up of smaller subsystems or subroutines that run in parallel or in sequence with each other and pass information between them in order to achieve a larger goal (Oxford University Press, 2015). These subsystems and subroutines can be as simple as functions within a program, or other applications running in separate threads, processes or even external to the physical system (be it hardware- or software-based). The manner in which these threads, processes and functions communicate is called Inter- and Intra-Process Communication.

This section brings to light some of these methods of concurrency and where they can be best applied.

3.2.1 Network Sockets

This method of communication makes use of the network and is particularly versatile as it can be used on an individual system, or to communicate with systems on a connected network. The most simple form of this communication method is on a point-to-point basis. To achieve this, one would create a listening server on one end and a connecting client on the other through

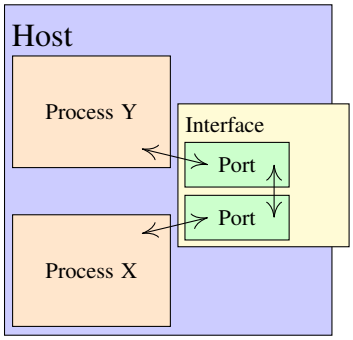

Figure 3.5: Single-Host IPC Using Point-to-Point Communications

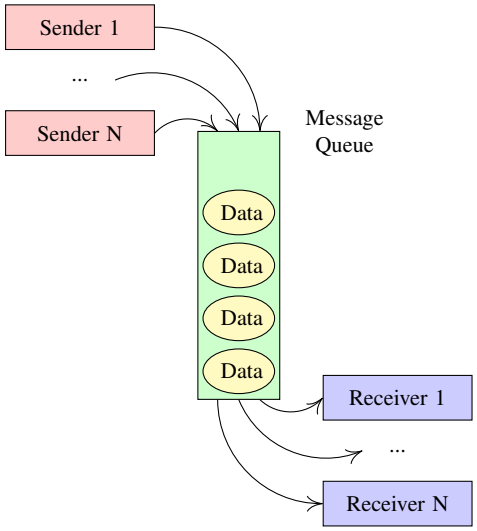

Figure 3.6: A Simple Message Queue

standard application of sockets (Winett, 1971). The client would then attempt to establish a connection with the server's listen socket and, if successful, data could then be passed between the pair. An overview of this is shown in Figure 3.4, where the arrows depict the data flow.

If one wishes to pass data between processes on a single host, one can specify 'localhost'[3] for the address the socket should attempt a connection with (Postel, 1981a; Deering, 1998). This form of communication is depicted by Figure 3.5. If the process or system one wishes to connect to is external to the host, then one can specify the IP address in which that host resides. If both are connected to the same network, connection should occur as expected and data can be passed.

An improvement on and extrapolation from the basic network socket is shown in Figure 3.6, and is an interface that builds upon this very same socket called the message queue. This interface

[3]IPv4 address 127.0.0.1

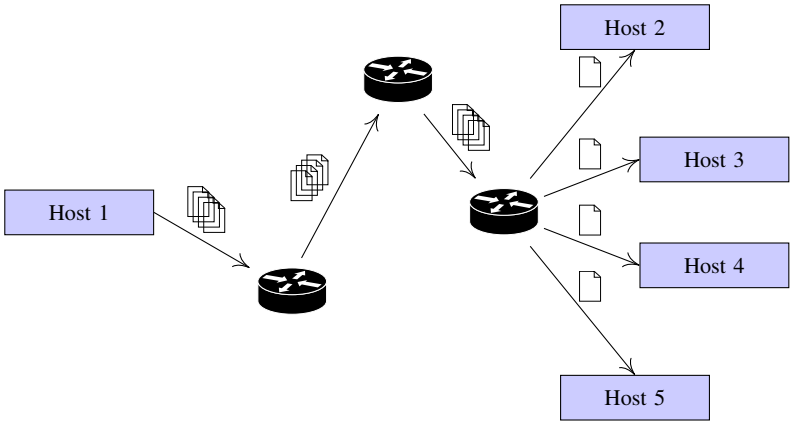

Figure 3.7: A Host Broadcasting Network Packets to Other Hosts

allows for data to be placed into a queue without the need for immediate action on the placed data. The data can then be fetched when resources are available to handle the data in question. What is more, no system that reads and/or writes to the message queue need know of any other system that reads and/or writes to the message queue. Further, one can dedicate a system to act as the message queue, thus removing the overhead in maintaining this interface from the system that processes the data within the message queue (Passint *et al.*, 1996).

The final form of socket communications in this subsection is the use of broadcasting and a high-level overview of this process is depicted in Figure 3.7. A broadcast is simply the action of sending out a single message to multiple listeners. In terms of network sockets, one can connect to a broadcast group. From this point, any packets that are sent to the broadcast group are received by all participants of the group. A number of technologies exist to achieve this function, such as multicast (Cain, 2006) and geocast (Navas and Imielinski, 1997). This removes the overhead of sending the same message to each participant of the group individually (Fiat and Naor, 1994). The use of this concept is discussed further in Section 3.2.5 by detailing ZMQ (Zero Message Queue), which is built on top of multicast networks (Hintjens, 2013).

3.2.2 Shared Memory

This method of data access allows multiple processes on a singular host to access the same piece of memory, thus only allowing for intra-host communication. Access to this memory can be through reads and/or writes, and can occur simultaneously. This method of memory access helps to prevent data redundancy within a host (Dagum and Enon, 1998). Overheads do occur in this method of memory access when two or more processes scheduled for physically

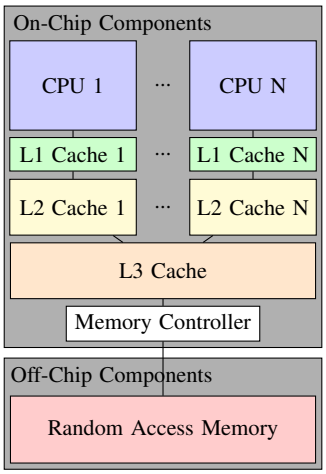

Figure 3.8: An Overview of a Processor's Memory Hierarchy

different processor cores try to access the same memory location. This is due to the structure of a processor (Openstax CNX, 2015).

Typically, a core within a processor has its own cache (or memory). A high-level overview of this can be seen in Figure 3.8. The data needs to be loaded to a core's local cache before it can be processed. At time of writing, there does not exist a commodity processor that allows separate cores to access another core's local memory directly. This means that if a modification is made to the data, it would then get marked as dirty to signify its modification.

In order for another core to continue to work on this data, it would have to wait for the data to be written out from the core which modified the data's local cache, to the public cache (memory that is accessible by all cores). This would then have to be read to the new core's local cache in order for it to be processed. This can lead to stalls in the system, as halts occur while waiting for data to be written to the public cache and read into a different core's local cache (Zhuravlev *et al.*, 2010).

Another point to note is that one should implement memory management variables that keep track of who has access to a set of data, or when a process is intending to write to the dataset (i.e. to mark a modified data as dirty, as mentioned previously). The reason for this is that if two processes write at the same time without any form of notification to each other as they write, they could end up overwriting each other's results (Hansen, 2013). Consider this simple example that explains this occurrence. Process X and process Y each take a time T to complete. If process X reads the data and is processing it when process Y reads the same original dataset to perform its process that takes time T, it is obvious that process X would complete first and write its results to the dataset. After this occurs, process Y would complete and write its results

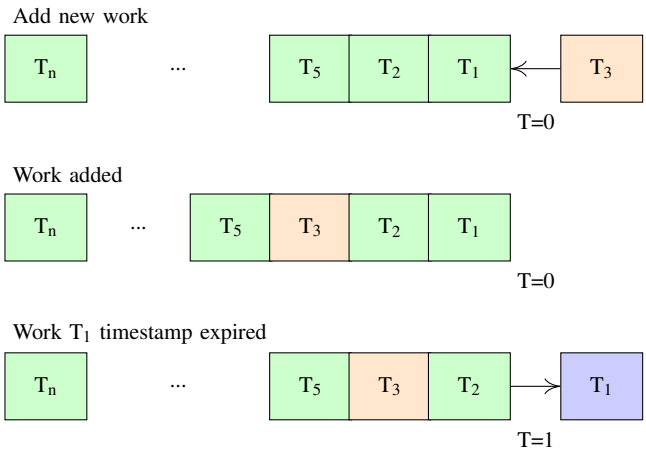

Figure 3.9: Simple Workings of a Work Queue

to the same dataset. The problem is that process Y's results are based on the original dataset and not process X's results to the dataset. At this point, process X's results are completely discarded, leading to a malformed result.

3.2.3 Threads and Work Queues

A thread belongs to a process and currently makes up the smallest work unit of execution that can be independently scheduled by an OS (Ramanathan, 2006). As threads run within the same process, they inherently share all memory within the process. This means that one no longer has to manage shared memory; however, all the overheads in resource contention still need to be dealt with within a process. As threads can be managed by the process and not just the underlying operating system, work flow can be more tightly managed by the process itself (Dice *et al.*, 2006).

Another motivator for threads is that a process running multiple threads can run its threads in parallel on a host with a multi-core processor; this helps to further optimize a process. In short, threads should be considered as a replacement to forked processes making use of shared memory (Reinders, 2007).

Work queues are built on top of threads, and help in better scheduling work flow within a process. The basic workings of a work queue can be viewed in Figure 3.9. The simplest use of a work queue is to assign a function or subroutine within a program that is to be called when a specified time has passed (Zheng and Thain, 2015). This defined work is then placed in the

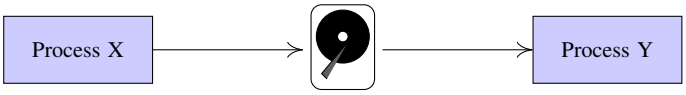

Figure 3.10: Two Processes Communicating Through a File on Hard Drive

scheduler, and when the time in which the work is set to start has passed, the process then wakes up the work queue handler to spawn a thread on the given work with the specified subroutine. Further options that can be specified are datasets to be passed to the function which gets called for the work, and which processor core the work should start on within the host.

3.2.4 Other Forms of Interprocess Communication

Other methods of passing data between processes can be performed through use of a shared resource. This resource could be a system such as a database (Silberschatz *et al.*, 1997) or simply a file that resides on a hard disk. A very simple depiction of this can be seen in Figure 3.10. Simply putting data into a database to have those contents read later, or performing the same action with a file on disk, would constitute a viable method of interprocess communication.

One can even go as far as to make use of the Linux kernel to either act as a pass-through, or to communicate with a kernel module directly using procfs (Oracle, 2007). The Linux procfs works through creation of file handlers that, when an action is performed upon it (be it read, write or any other file operations), calls predefined functions to run and process the relevant data for the command (Wang *et al.*, 2010). For example, if a write is requested, the data being written to the file actually gets stored in memory, and the whereabouts of this data is then passed to the defined function so it can locate and process this data. A read expects a result to be returned which is created by the function paired with the read handler. This result is produced and written to memory, and the results returned hold the location of these results in memory, and how large the buffer is that contains them.

3.2.5 Zero Message Queue

The Zero Message Queue[4] (ZMQ) library gives access to a distributed networking model that makes use of sockets and broadcasting to allow for increased concurrency within a system. ZMQ is also designed for high throughput and low latency scenarios where event response

[4] http://zeromq.org/

turnaround is a key factor in the life cycle of a system. Furthermore, these transmitted messages ensure atomicity over the entire broadcast group. The ZMQ transport layer can be set to in-process, inter-process, TCP and multicast modes, depending on whether the communications are happening within a process or between processes on a single host, or between processes on separate hosts (Hintjens, 2013).

This library also allows for multiple forms of connection patterns that allow for N-to-M (where N is a natural number and M is a whole number) connection using topologies such as fan-out, pub-sub, task distribution and request-reply.

Fan-out: This method evenly distributes the data containing messages to all available clients in the fan.

Fan-in: This method evenly pulls data containing messages from all available clients in the fan.

Pub-sub: This method has a publisher which produces messages containing data. A subscriber subscribes to a publisher and from this point receives all messages the publisher publishes. A publisher can have multiple subscribers and a subscriber may connect to multiple publishers.

Request-reply: This method is made up of a client and server, and for every request made by the client to the server, the server expects a response from the client.

This library also allows for an asynchronous Input/Output (IO) model to allow for data to be handled when resources are available to deal with it (Hintjens, 2013). Importantly, this library also supports a multitude of programming languages and OSs, allowing one to use the language and platform that is best suited to process the data at hand.

3.2.6 Rabbit Message Queue (RabbitMQ)

Rabbit Message Queue[5] (RabbitMQ), although functionally equivalent to ZMQ[6], focuses on ease of deployment and ease of use. Out of the box, it supports advanced routing scenarios, load balancing and persistent message queuing (Videla and Williams, 2012). However, this focus causes restrictions in terms of throughput, as supporting these features causes RabbitMQ's headers to be larger, and thus take longer to transmit and interpret. Another factor that adds latency to RabbitMQ's protocol is that it heavily relies on the Advanced Message Queueing Protocol (AMQP) and thus has a centralized server (Vinoski, 2006). This prevents client-to-client communications, and instead clients need to communicate through a centralized server.

[5] http://www.rabbitmq.com/

[6] In terms of ability to broadcast a message

Table 3.1: Inter-Process Communication High-Level Overview

IPC	Strengths	Weaknesses
Network Sockets	Easy-to-use	Slow
Shared Memory	Fast Multi-process	Single-host execution Requires memory management
Threads and Work Queues	Fast	Single-host execution Requires memory management
ZMQ	Easy-to-use Multi-language support Network scalable	Small scale is slow
RabbitMQ	Easy-to-use Network scalable	Small scale is slow Large header size
Nanomsg	No dependencies Network scalable	Small scale is slow C implementation only

3.2.7 Nanomsg

Nano Message[7] is another option for message queue-based interprocess communication. This library is based in C, and again features many of the routing scenarios, which it names "scalability protocols", that both ZMQ and RabbitMQ support. The selling point of this library is that, as it is implemented completely in C, it has no dependencies at build time or runtime of a system. Other than this fact, this library is functionally equivalent to ZMQ and RabbitMQ with respect to Bolvedere.

3.3 Summary

This chapter explored current existing technologies that were considered to form the foundation on which Bolvedere was built. This chapter began with an evaluation of existing architectures in **Section 3.1**. This section brought CPU, GPU and FPGA technology to the forefront, and in evaluating each, found that the best implementation of this system would be an amalgamation of these technologies.

It became apparent at this point that, in order to gain the benefits of each architecture, an effective mechanism for communication between each architecture would be required. Methods of communication between these subsystems were evaluated in **Section 3.2**. Continuing from this point, **Chapter 4** designs the Bolvedere system before comparing which technologies explored

[7] http://nanomsg.org/

in this chapter are best suited for each subsystem of Bolvedere. A table summarizing each technologies strengths and weaknesses with respect to Bolvedere's requirements is found in Table 3.1.

4

Design and Implementation of the Base System

THIS chapter introduces the implementation of the Bolvedere system. The base system is specifically detailed in this chapter, whereas processor module designs, which specifically deal with the analysis of NetFlow records, are detailed in **Chapter 5**.

This chapter reasons and produces the fundamental design and implementation of Bolvedere as a whole. This chapter starts with the foundational flow of logic being reasoned out in **Section 4.1**. Once the overarching logic flow is produced, **Section 4.2** proceededs to design Bolvedere in greater detail, and implementation decisions are made. Finally, this chapter closes with **Section 4.3** defining the manner in which Bolvedere is configured.

4.1 Implementation Goals

This research ultimately aims to produce a scalable distributed NetFlow v9 (Claise, 2004) and IPFIX (Quittek *et al.*, 2004) (also referred to as NetFlow v10) analysis platform for detection

of malicious and unintentional events upon a system from their external network link. The decision to use NetFlow over sFlow (Phaal *et al.*, 2001) was made due to the accuracy that NetFlow offers when compared to sFlow, the availability of Cisco hardware, and the support for reciept of JFlow records (Juniper Networks, 2011). The problem when dealing with NetFlow v9 records is that what is monitored, as well as the order in which monitored fields are reported, are unknown until runtime. To solve this problem, the following flow of logic is proposed for this base system:

1. NetFlow records are received by a collector. For all purposes, this collector will act as a NetFlow record sink.
2. NetFlow records in the collector are discerned as to whether they are template records, or data records.
3. If the NetFlow record is a template, the template is stored to be used for later discernment of data recrods.
4. If the NetFlow record contains data, and the template is held for the data record, the data is discerned.
5. The discerned data record is now re-ordered into a known format so as to allow optimisation of all subsequent operations that are executed on CPU and GPU.
6. At this point, the discerned and re-ordered data record is distributed to NetFlow record processors for analysis.

4.2 System Design

It is clear from the proposed flow of logic for this system that there are three overarching operations that need to be performed. These are:

1. The collection and discernment of NetFlow records.
2. The distribution of these discerned NetFlow records.
3. The analysis of the NetFlow records.

Collection and discernment of these records is performed by following the NetFlow specification (Claise, 2004). Analysis of these discerned records is done on a per analyser basis, and the operations performed are based on what is being analysed. This leaves only the manner in which distribution of discerned records to analysers takes place to be considered.

One of the goals of this research, as previously detailed in Section 1.2, is scalability. This must be taken into consideration when defining how NetFlow records are distributed to analysis processes. With this goal in mind, the ideal distribution method to analysis processes would be in the form of one-to-many, which would allow multiple processes to receive the same message. This would reduce the need to have a collection and discernment mechanism for each process performing analysis.

Considering the technologies evaluated in in Chapter 3, this form of distribution can be achieved in several ways. A message queue system, configured to use a publish-subscribe routing scenario (Eugster *et al.*, 2003), would best suit record distribution. The reasons for this are now discussed.

The first advantage of using a message queue system is that it is a network-based protocol. This means that messages (in this case the NetFlow records) can be distributed to hosts outside of the host in which the message queue system is running. This gives an immediate advantage in terms of scalability when comparison is made to shared memory or file on disk communications. This also naturally trumps multi-threaded processes (Rumelhart *et al.*, 1987).

Other network technologies, such as socket-based communications and network groups, also have the ability to distribute data between hosts. Unfortunately, using a direct socket-based system to distribute a message to multiple hosts requires that system to retransmit a copy of the message for every receiving host; this is very resource-inefficient. The solution of broadcast groups is effectively functionally equivalent to the aforesuggested publish-subscribe routing scenario (Banavar *et al.*, 1999).

At this point one should have noticed that there will be multiple NetFlow record processors receiving a distributed message from a single publisher to which they are subscribed. The main point to note is that each analysis processor that is developed will need to interface with a publisher. The overheads in achieving this task on the developer's end should be kept to a minimum. For this reason, a message queue in the publish-subscribe configuration was chosen, as it would be easier to configure than a broadcast group.

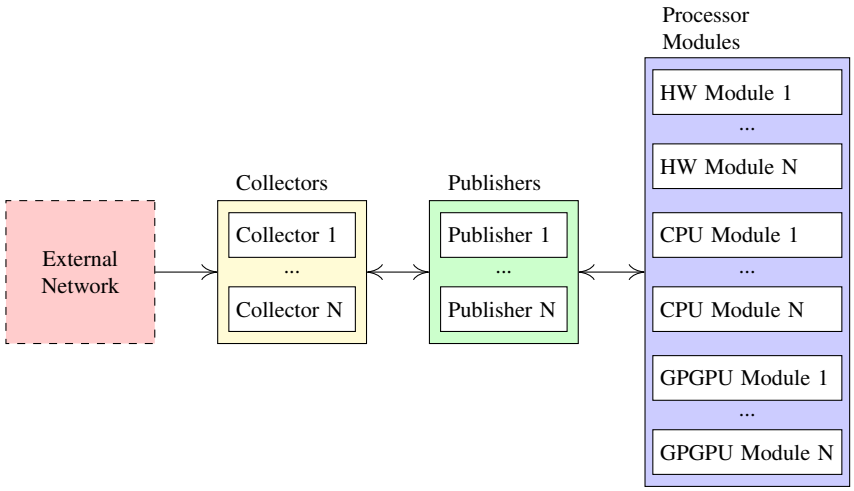

Figure 4.1: High-Level System Overview

4.2.1 System Components and Flow of Logic

Section 4.2 identified three major operations that need to be performed in Bolvedere; namely the collection of NetFlow records, distribution of these records, and analysis. Other operations that were defined in Section 4.1 were that of NetFlow data record discernment performed by storing template records for later use, and re-ordering of NetFlow data record fields in order to enable optimization of sequential architecture processors at further points in Bolvedere's flow of logic.

The best place for this re-ordering to occur would be at receipt of a data record. Once the data record is discerned, it should be re-ordered into a known form as required by a sequential subsystem of Bolvedere. The message queue system in Bolvedere is positioned at a point from which it can communicate with both analysis processes and the collector. This logically places this message queue publisher in the middle of the system. For this reason, control of the order in which NetFlow data records should be put was given to the publisher.

This breaks Bolvedere down into three major components: the *collector*, *publisher*, and subscribing *processor modules*. A high-level overview of Bolvedere, and how logic flows between each of these components, can be viewed in Figure 4.1. More detailed descriptions of each of these components follows:

Collector: Receives configuration from publisher as to which NetFlow fields the publisher wants to receive, as well as the order in which it wants to receive them in. It then collects NetFlow records containing both templates and data. It is then discerned which NetFlow

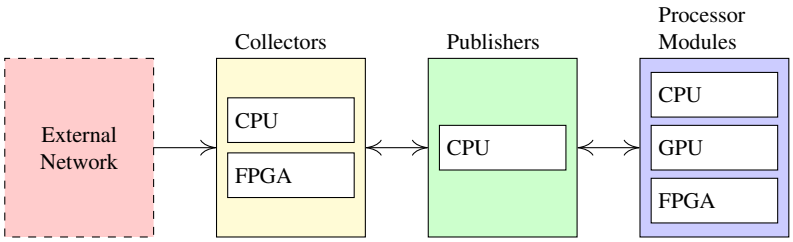

Figure 4.2: Proposed Host Architecture for Each Bolvedere Subsystem

template these data records apply to, and the records are re-ordered and filtered into the form that the publisher's configuration requested before being passed on to the publisher.

Publisher: The publisher receives its name from its functionality - it publishes processed data to the back-end containing all processor modules. The publisher is the central point of the system and knows what it wants to publish. Because of this, it knows what it requires from the collector in order to produce and create the information which it publishes. Using this knowledge, the publisher configures the collector, and once data is received from the collector and filtered by the publisher, the publisher then publishes this information via a broadcasting method to the back-end where processor modules reside.

Processor Module: This is where all major data and information analysis, processing and enrichment occurs. A processor module's role is to listen to information published by a publisher and perform analysis on it in order to determine something about the flows being collected by the system. As publishers work in a broadcasting-like method, multiple processor modules can listen to a single publisher, allowing for parallel computation.

4.2.2 Architecture

Which architecture to use when building each component of Bolvedere is now considered. The three major architectures considered are listed below:

CPU: High single-data throughput, commonplace as primary data processor in most systems, well-defined instruction set with relatively small architectural changes from revision to revision.

GPU: High multi-data throughput, not as common as CPUs in systems. With standardization of CUDA and OpenCL, the instruction sets for these devices see little change to existing instructions.

FPGA: High parallelism, uncommon, used for design of hardware for solving specific problems, fast for solutions where no prior hardware processor exists.

Starting with the collector, it was decided for this system to make use of an FPGA; however, a software-based implementation would initially be created to prove that this subsystem would work, and for deployment where dedicated FPGA hardware is not available. A CPU-based software implementation of this system, although not perfect in terms of available instructions, can get past that barrier with the sheer number of instructions it can perform in the time it takes an FPGA to do one (Underwood, 2004). CPUs are also commonplace, so this would reduce costs by not having to create dedicated hardware to achieve the task.

This being said, the use of FPGAs allows for future-proofing of this system. An FPGA design can be used as the basis to fabricate ASICs that perform the same task as that of the FPGA but at much higher speeds (Markovic *et al.*, 2007; Kuon and Rose, 2007). This will allow one to fabricate and use a processing unit designed with hardware support for instructions specifically designed to perform the job of the collector, and run at speeds equivalent to that of a CPU. It is clear that this implementation would out-perform a CPU; however, the downside is the barrier to entry, this being the cost.

The task of the publisher of this system is to configure the collector as to the order of how it wants to receive data, and then apply simple filters before publishing the data to processors. As the data is of a known form when received from the collector, it can be worked through from top to bottom when serializing it for transmission. The platform decided on to achieve the main processing of this task was the CPU. This is because CPUs work well through data in a sequential format (Intel Corperation, 2016), which the data received by the publisher from the collector is.

In the big picture, this means that the CPU never has to look up what a field is before applying filters. Consider this example to better explain this logic. If one were to apply the following filter "*if not port 80 then don't publish*" to a publisher, the CPU would have to first look up if the relevant field/s exists[1] in the NetFlow log, and then go and fetch it with a specific offset that has first has to be calculated on a log-by-log basis. This would drastically slow down the system (Smith, 1985; Zukowski *et al.*, 2006). Having received the data in a known, requested format from the collector, accessing the data is as simple as using a predefined offset and then applying the filter.

Processor modules in this system are a bit of a mixed bag. These are subsystems designed to receive filtered logs of a known format from a publisher or multiple publishers in order to

[1] In this case both source and destination ports.

process and determine something about the received logs. Depending on what that something is, the underlying architecture in which the relevant process is best suited for varies. For this reason, anything goes at this layer of the system. The idea is that the best tool for the job should be used, and if that requires a simple piece of software or a dedicated piece of hardware, it should be allowed without question.

4.2.3 Interprocess Communication

As the processor module subsystems within this system use different platforms to achieve their tasks, they need a common way in which to communicate. Listed below are the communication methods considered for use in this system:

Network Sockets: Hardware is commonplace in computers, well-supported standard.

Shared Memory: Faster than network sockets, consideration needs to be taken for shared memory access, cannot distribute shared memory resources to an external host (Dagum and Enon, 1998).

Threads and Work Queues: Can truly run processes in parallel if host has multiple processors (Tullsen *et al.*, 1995), similar memory considerations need to be taken as shared memory, sharing memory is easier to configure as this is implicit, cannot share memory outside of a process.

Secondary Memory: Slow as disk access has to be performed for most data access, wears out mechanical parts, good for long-term storage of data.

Zero Message Queue: A message queue implementation that allows for same features as network sockets, handles broadcast groups silently with little to no overhead, supported by over 30 languages (Hintjens, 2013), is well-defined protocol.

RabbitMQ: A message queue implementation with ease-of-use at its core. Like ZMQ, it supports handling of broadcast groups with little overhead and support across multiple languages (Videla and Williams, 2012), response latency for RabbitMQ is typically slower than ZMQ.

Nanomsg: A message queue implementation that was implemented in C to be able to executed without relying on any dependencies.

ZMQ was selected for this system as it is network-based, and also allows for acceptable ease-of-use, with a trade-off for lower overheads in communications. Furthermore, ZMQ is available

and supported across multiple programming languages, as well as being easy enough to implement on a language that does not support ZMQ, due to its well-defined protocol.

This inter-process communication interface also bridges the gap on data distribution to multiple processor modules. This is achieved through the built-in support of broadcasting through the publish-subscribe routing scenario. This allows a single packet to be transmitted and received by multiple hosts. This means that a single message can be sent by a publisher and received by all subscribed processor modules executing locally on, or remotely from, the publisher.

4.2.4 Configuration

The primary distributor of this system's configuration to other subsystems is the publisher. This decision was made due to the placement and role the publisher plays in this system's communications path. The publisher is attached to both collector and processor module parts of Bolvedere, and so can easily communicate the relevant configuration to each. This can be seen in Figure 4.1. If the subsystem that held the configuration is either a collector or processor module, there will be extra overhead required in passing this information through the publisher.

This configuration is communicated at the boot of each collector or processor module, and a publisher's role with respect to configuration is to simply listen for configuration requests. Based on this, an outline of the flow of logic for each subsystem is presented below.

The timeline for a publisher is as follows:

1. Compile with use of configuration and template files.
2. At boot, create user interface to report feedback.
3. Attempt to start collector side listen socket.
4. Attempt to start processor side ZMQ publisher socket.
5. Start thread to wait for configuration requests.
6. Start thread to receive re-ordered NetFlow logs.
7. Start thread to publish filtered logs using ZMQ handle.

The timeline for a collector is as follows:

1. Create socket for communication with publisher.
2. Request configuration from publisher as to what NetFlow fields the publisher requires and the order the publisher requires these fields in.
3. Once a configuration response is received, store this for later use.
4. Create socket to receive NetFlow logs and templates from NetFlow sources.
5. Use appropriate process mechanisms according to type of NetFlow data received defined by NetFlow standard.

The timeline for a processor module is as follows:

1. Perform steps required to start process in chosen language.
2. Import language-specific ZMQ library.
3. Check what publishers exist on the network.
4. Attempt to subscribe to selected publisher or publishers.
5. Use this ZMQ socket to receive filtered NetFlow logs from the publisher in main process loop.
6. Process received data.

One should note that, in order for the publisher to know the configuration for each part of the system, the configuration is generated at compile time of the publisher subsystem. The reason for this is that the publisher is CPU-based, and receiving a dynamic configuration at runtime will incur a CPU bottleneck in discernment of the configuration to transmit to collectors and processor modules.

4.3 Build Time and Run Time Configuration

Ease-of-use, configuration and code re-usability can break or make an application intended for use by any user base. For this reason, careful consideration had to be taken into account when

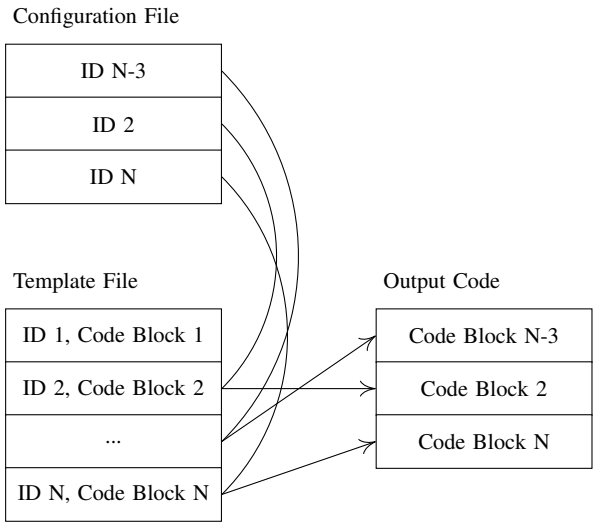

Figure 4.3: Configuration File Selecting Code to Build Publisher from Template File

decisions were made as to how this system as a whole would be set up and interfaced to. This section focuses on two major points of this system; the collector being the collector and the publisher. The system was designed with the intent of the collectors belonging to publishers that processor modules would in turn subscribe to.

The reason for focus on the collector and publisher (also referred to collectively as the base) aspect of this system is that processor modules are intended to be third-party, and thus detract from the core platform of Bolvedere.

4.3.1 Build-time Configuration Templates and Configuration

Configuration for this system was designed with code re-usability in mind, and so makes use of two reusable files: a template file and a configuration file. A configuration file relies on a template file, as the configuration file looks up what code needs to be built into the publisher based on the selected template file for configuration. A template file on the other hand is where all functional code is stored on a modular basis. This is better represented in Figure 4.3, where one can view a configuration file being used to fetch requested code from a template file in order to build the code base for the publisher.

The simplest form of physical configuration is considered in this example, in which a single collector is directly attached to a single publisher, which then has a single processor module subscribed to it. At boot time of this system the first subsystem to come online is the publisher,

due to the fact that the publisher knows what it wants to publish, and thus knows what it requires from a collector in order to publish that data. The next subsystem to come online is the collector, and it notifies the publisher of this fact.

The publisher then connects to the collector and transmits a configuration as to what information the publisher wants and the order in which it expects it. Finally, the underlying mechanism which allows for processor modules to connect is brought online. The functionality of each subsystem, and the order in which the system boots up into a running state, was seen to be the most logical and easiest to implement at time of design. Listing 4.1 shows how a template file is constructed. First of all, any text after a '#' character is treated as a comment and thus ignored at build time. There are two optional headings that can appear anywhere within this template file; the INCLUDE_HEADER and GLOBAL_HEADER. The INCLUDE_HEADER allows for one to include their own C code external to the publisher's already included headers, thus allowing one to build up their own tool set and further increase code re-usability. To define a new header, one adds where the header is located in conformance with the C standards, without the need for a #include directive.

The GLOBAL_HEADER heading allows one to define persistent global variables within the configuration. All variables that can be declared in C, as well as creation of structs and #defines, can be declared here. This enables a user to perform simple tasks such as keeping state over multiple received NetFlow logs, which, without a persistent variable, would not be possible.

These modules defined in the template file are written in C and require a name, the NetFlow field ID on which the module is intended to work, and the size of this field in the NetFlow data record. Regarding the pointers "*relevant_data*" and "*zmq_buffer*" in the template file, the "*relevant_data*" pointer always points to the field specified by the template module in the received NetFlow log. This is the reason for the specification of the NetFlow field ID and the fields size with the name of the module, as this allows the publisher builder to discern where "*relevant_data*" pointer should point to in memory at build time for run time.

The "*zmq_buffer*" pointer is far simpler in its workings. It is a user-modifiable pointer to the publisher's ZMQ data buffer that it publishes to all subscribed plug-ins. The user can move this pointer around at will, and write to any location within this buffer. Once every record in a NetFlow log is processed, this buffer is automatically handed to the ZMQ handler to broadcast the data in the buffer to all subscribed processor modules.

One can define as many modules as they desire within a template; however, the use of duplicate names is not allowed. Because of this restriction, it follows that one can only define a single INCLUDE_HEADER and GLOBAL_HEADER. The use of C in these template files is due to the

Listing 4.1: Template File

```
# Any comment starts with a hash character

# This will insert these C headers at build time of the publisher
INCLUDE_HEADER
{
"../path/to/library.h"
<stdio.h>
}

# This will instantiate global variables in the publisher accessible
# by all selected templates
GLOBAL_HEADER
{
    uint8_t generic_variable = 0;
    uint8_t *generic_char_array;
    int32_t generic_return_value = -1;
}

# This is an example of a module with code
# Format: module name, relevant NetFlow field ID,
#         size of NetFlow field
EXAMPLE_MODULE_1, 1, 4
{
    *zmq_buffer = *(relevant_data + 0); zmq_buffer++;
    *zmq_buffer = *(relevant_data + 1); zmq_buffer++;
    *zmq_buffer = *(relevant_data + 2); zmq_buffer++;
    *zmq_buffer = *(relevant_data + 3); zmq_buffer++;
    zmq_buffer_len += 4;
}

# This is another example module
EXAMPLE_MODULE_2, 7, 2
{
    printf("Layer 4 Source Port is %04X\n",
           ntohs(*(uint16_t *)relevant_data))
}

...
```

Listing 4.2: Configuration File

```
1 IPV4_SRC_ADDR     # Modules requested to build publisher in order
2 IPV4_DST_ADDR     # that will be requested from collector.
3 L4_SRC_PORT
4 L4_DST_PORT
5 IN_BYTES
6 IN_PKTS
```

Table 4.1: Configuration Template Code Symbol Replacement

Symbol Name	Code Replaced With
INCLUDES	Replaced with the modules definitions definitions in the INCLUDE_HEADER heading.
GLOBALS	replaced with the global variables that are defined in GLOBAL_HEADER.
PROCESS	replaced by the code defined in the selected modules and is placed in the order the configuration file listed these modules in.
OFFSETS	This symbol is used for configuration of the collector and is further discussed in Section 4.3.2.

fact that C is a well-defined language that compiles to native byte codes on the host system, allowing for maximal performance. Exactly how this C code is used to build the publisher will now be discussed. A configuration file would look like the example given in Listing 4.2, and is nothing more than a list of modules to use within a template file. The order of this file matters, as it determines the order in which modules defined in the template file should be executed at the runtime of the publisher.

At build time, a template file and configuration file are passed into the publisher builder. At this point, the configuration file is read and all modules which the configuration file is intended to build into the publisher are read. The next step is to check whether each of these listed modules exist within the template file. If these modules exist, the build process starts.

In Appendix C, one can view the template file in which the module's C code is loaded at the points marked between symbols '>>>' and '<<<'. What each of these are substituted for can be found in Table 4.1.

After the C code defined by the template file has been placed into correct points defined in the C template, compilation can occur. The publisher builder at this point simply compiles the publisher[2]. After this has succeeded, the publisher builder terminates. The compiled executable publisher can be found in the same folder the build command was run in.

[2]This was performed using GNU (GNU's Not Unix) Compiler Collection (GCC) (https://gcc.gnu.org/).

Listing 4.3: Configuration Byte Sequence for Collector

```
1 uint8_t conf[] = {0x00, 0x08, 0x00, 0x04, // IPV4_SRC_ADDR Size 4
2                   0x00, 0x0c, 0x00, 0x04, // IPV4_DST_PORT Size 4
3                   0x00, 0x07, 0x00, 0x02, // L4_SRC_PORT   Size 2
4                   0x00, 0x0b, 0x00, 0x02, // L4_DST_PORT   Size 2
5                   0x00, 0x02, 0x00, 0x04, // IN_PKTS       Size 4
6                   0x00, 0x01, 0x00, 0x04  // IN_BYTES      Size 4
7                  };
```

4.3.2 Runtime Collector Configuration

As stated in the Section 4.2.1, the collector's job is to re-order NetFlow logs into the format that the publisher expects. This is so it can most effectively perform its step in the system's process. The reason for this is due to NetFlow version 9 and newer allowing for customisation of NetFlow logs through the use of templates, as explained in Section 2.3. The collector keeps track of NetFlow templates and uses these to discern received NetFlow logs, before filtering and re-ordering them into the format that the publisher requires. The storage mechanism for templates will be further explained in Section 4.3.3.

The configuration required to achieve the task of re-ordering is received from the publisher, as the publisher knows what it needs to produce to its subscribers (the processor modules). At publisher build time, when all known modules have been read from a template file containing the modules' codes, as requested by the configuration file, the publisher builder uses the NetFlow field ID and field size information provided for each module in the template file in order to build a configuration byte sequence for a collector. Listing 4.3 shows the line of C code produced by the publisher builder that contains the byte sequence sent to the collector in order to configure it. As the NetFlow template standard uses 16 bits for the field ID and 16 bits for field size, this system was designed to hold to this standard itself and uses 16 bits to represent both the field ID and size respectively. With this in mind, interpretation of the byte sequence follows the form of first 16 bits NetFlow field, and following 16 bits states the size of the field in bytes. In Listing 4.3, one can see what the ID of the first 16 bits represents, followed by the size represented in the following 16 bits in the comment at the end of each configuration line.

As 16 bits is equivalent to 2 bytes, and the type *uint8_t* is equivalent to a single byte within C, the array in Listing 4.3 can be read in 4 byte blocks. This means that the first 2, bytes when read in little-endian, represents the NetFlow field ID, and the second 2 bytes of the 4 byte block represents the field size in little-endian. Where these values are collected can be seen in Listing 4.1 when interpreting a module header as defined in Section 4.3.1. Furthermore, these collected values are also in order of the module acquisition as defined by the configuration file used. From

the collector's point of view, this configuration information is all that is required in order to find any field required by the publisher.

One should note that this configuration buffer shown in Listing 4.3 is built directly from the configuration file shown in Listing 4.2. As noted before, the order of the template modules supplied in the configuration file is the same as that in the generated configuration buffer. This will result in the a collector receiving this configuration passing data fields in this same order to the publisher.

Missing Data Record Fields

It is worth noting that any NetFlow log received which, once interpreted with the correct NetFlow template, is found to not contain a field required by the publisher, is populated with every bit in the field sent to the publisher set to 1. Although it is not impossible for a required field to represent correct data as this binary value, it was thought to be highly unlikely, and thus used in this implementation. When the publisher receives this information from the collector it can then be dealt with in an appropriate manner as defined by the publisher's module configuration for that field.

Runtime Collector Configuration Request

Lastly, the finer details as to where this configuration byte sequence is stored and how the collector signals the publisher in order to receive this byte sequence needs to be addressed. In Table 4.1, an explaination of ">>>OFFSETS<<<" was given, but its functionality was not directly explained in Section 4.3.1. This symbol is replaced by the configuration buffer (example shown in Listing 4.3) at build time and is what is passed to a collector for configuration when requested.

In order for the collector to receive this configuration byte sequence, the collector has to first signal the publisher that it is ready to receive the configuration byte sequence. The collector performs this by sending the message "*I'm alive*" to the open socket on the publisher in which the publisher expects to receive collector information. Once this message is received by the publisher, the publisher responds with a packet containing the configuration byte sequence. After this, the collector can configure itself and start to transmit the information the publisher wants to the publisher's collector socket.

Defining a configuration byte sequence and keeping the collector separate from the publisher, even though this task of collection is something that can be performed by the publisher, was intentional. The task of looking for a field within a NetFlow log is disjoint from any other field search, and is thus easily parallelizable. As such, this is not a task that should be expected to perform best on a sequential operation architecture host, such as the one the publisher is expected to execute on, as discussed in Section 3.1.1. Because of this, it was decided to separate the collector from the publisher in order to replace the collector at a future point for a hardware platform that is more suited to the task (such as a GPU or ASIC). A well-defined, easy to parse configuration byte sequence was aimed to aid in ease of replacement of this subsystem.

4.3.3 NetFlow Template Store and Re-Ordering

Although the subject of NetFlow data record re-ordering has already been discussed in Sections 3.1 and 4.3.2, a more in-depth explanation of this process is detailed here. This process undergoes 3 major steps:

1. NetFlow template record storage.
2. NetFlow data record discernment.
3. NetFlow data record re-arrangement.

Template storage is already explained in Sections 3.1 and 4.3.2. Briefly, this refers to the process of storing templates that arrive at the collectors interface that are generated periodically by NetFlow record sources. Without these templates, interpreting the information of all NetFlow records sent by that NetFlow source to a collector would yield no meaning, as the collector would have no way in which to interpret the log. For this reason, templates must have unique IDs, and received records must state the template ID required to discern the data records.

When a data record is first received, it is simply seen as a NetFlow header followed by a sequence of bytes. The template ID is read from the NetFlow record, and the relevant template is then used to determine what the sequence of bytes means. Once which fields the NetFlow record contains, and what value is in each field, is known, the collector can move onto the next step, which is re-ordering.

Re-ordering takes the fields required by the publisher from the discerned NetFlow data record and builds a new byte sequence of these fields in the order the publisher wants the fields. Any

fields that the publisher requires that are not in the discerned data record are populated with every bit set to 1.

This method also helps remove overheads required by the publisher to process headers, as the NetFlow headers used for data record interpretation are no longer needed, and are thus removed. This saves 20 bytes on the NetFlow header and 8 bytes per FlowSet. Given that a NetFlow packet can contain up to 30 records, this is a saving of up to 260 bytes that can be used to carry other metadata without incurring a bottleneck on the system's data path.

4.4 Summary

Section 4.1 started this chapter by defining the major operations that needed to be performed in order to received and process a NetFlow data record. This flow of logic was then built upon in **Section 4.2**, which defined major components of Bolvedere and their roles. Architectures for each component were then decided upon, before defining the manner in which component communicated was specified. This chapter then closed with **Section 4.3.1** defining how Bolvedere would be configured and run.

With Bolvedere's base system specified, this document now details processor modules in **Chapter 5**. This chapter will detail analysis algorithms used to detect network anomalies from data received from a Bolvedere publisher.

5

Processor Modules

GIVEN the design completed in the previous chapter, this chapter will focus on algorithms implemented as processor modules for Bolvedere. These processor modules receive the processed NetFlow data records from the publisher subsystem via ZMQ. As the application of each of these processor modules can vary widely, each implemented algorithm will be addressed to the theory it is based on, how it was implemented, and what is expected at the runtime of the module. It must also be noted that this chapter points out the ability to use the best tool for the job through implementations using best programming language and hardware.

Section 5.1 starts with a look at the detection of DDoS attacks. The approach taken to detecting this form of attack in this research is via implementation of a Neural Network using R. Following this, **Section 5.2** takes a look into the application of Fourier Analysis on NetFlow data when compared to full packet analysis techniques.

Detection of port scans through examination of NetFlow data records is then targeted in **Section 5.3**. This utilizes network flow direction and time outs to detect port scans. **Section 5.4** deals with sudden changes in throughput on individual ports on a per IPs basis, how these are detected and why this type of activity should be of concern.

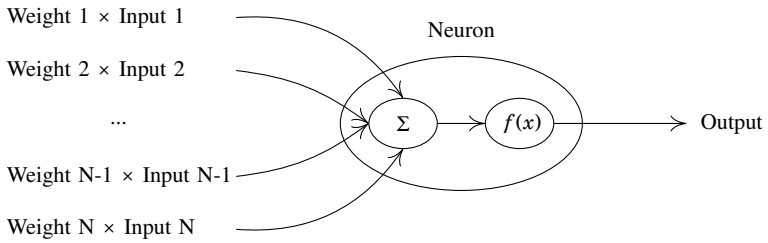

Figure 5.1: Basic Overview of a MLP Class Neuron

Section 5.5 then discusses the use of the NetFlow source nodes' unique IDs to geolocate a network flow instead of the use of the source IP which could be spoofed. This module is supported in **Section 5.6** which implements anomalous source IP detection by comparing the source IP of network flows to known subnetworks through an interface on the NetFlow source node.

The final module is detailed in **Section 5.7**, which implements a network-based exploit detection system that uses network fingerprints based on NetFlow data records from controlled malicious exploits to attempt to further detect reimplementation of these malicious events on a network.

As a note to the reader, this chapter aims to define how processor modules work with respect to their implementation and configuration. **Chapter 6** follows this chapter with testing of these modules.

5.1 DDoS Detection Through Use of Neural Networks

The R Project (The R Foundation, 2016) as a whole is an environment for statistical computation used through a well-defined programming language called R[1]. This language and environment is used for anything from simple statistical analysis such as the chi-square test (Moore, 1976), to more complex calculations such as fluid dynamics or rocket flight patterns.

The use of R in this algorithm is for something in-between these diverse goals, this being a neural network. A neural network tries to mimic the brain's learning ability through the representation of a brain by neurons connected to each other through a weighting system; although this is not an accurate representation of the brain (Hall *et al.*, 1992). A neuron simply has an input value, a weighting function for the input, a normalization function for the case of multiple inputs, and an output derived from these inputs. Figure 5.1 shows a graphical representation

[1] https://www.r-project.org/about.html

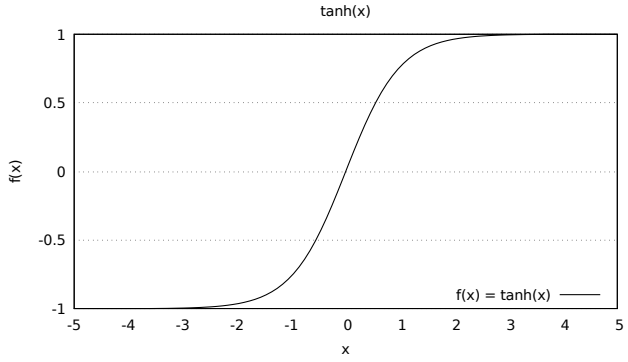

Figure 5.2: Graph produced by f(x) = tanh(x)

of such a neuron taking in weighted inputs that are summed before being used as input to an activation function.

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} = tanh(x) \tag{5.1}$$

One such activation function is the Hyperbolic Tangent and is generated by the equation found in Equation 5.1. This produces the graph found in Figure 5.2. At this point, one should note that the larger the summed weighted input gets, the closer to the value 1 the output of the neuron becomes. This normalization function is characterized by its initial steep slope leading into an asymptote towards the maximal normalized value 1.

Neurons are then arranged into layers which can only communicate to the next layer or subsequent layers (Hagan *et al.*, 1996); this ensures signals propagate forward from start to finish. These layers have specific roles and are classified as follows:

Input Layer: This layer receives the initial input information.

Hidden Layer: This layer receives its input from the input layer and relays its message forward to the output layer.

Output Layer: This layer receives its input from the hidden layer and outputs a final result out of its output.

The hidden layer that lies between the input and output layer may in reality consist of multiple layers. These layers are usually collectively named simply as hidden layer and an incrementing number. These hidden layers follow the same rules of all layers in which they are only allowed to connect their output to an input of a subsequent layer. This information can all be referred

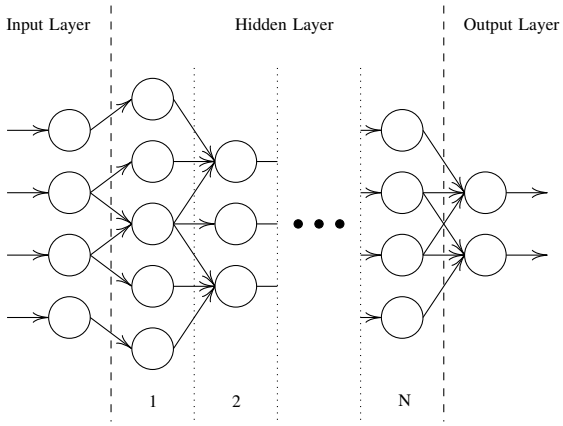

Figure 5.3: Representation of a Neural Network

to in Figure 5.3. This form of neural network falls into the MLP class of neural networks and requires at least 3 layers. This being: 1 input layer, 1 hidden layer, and 1 output layer.

Detection of DoS and DDoS attacks is typically done through use of the packet count and byte count fields of the NetFlow log. When considering a recurring pattern of byte and packet count with close average bytes per packet coupled with near maximum bandwidth use on a link, this is usually a good indication as to a form of DoS attack (Estan *et al.*, 2004; Zhenqi and Xinyu, 2008). This research took the approach of training and then using a neural network in order to perform detection of DoS and DDoS attacks using these aforementioned characteristics.

5.1.1 Data Representation and Training

Training of a neural network can take a substantial amount of time[2] and is due to two factors. The first is acquisition of a training dataset. This dataset consists of a set of known outputs for given inputs (Setiono and Liu, 1997). This allows one to pass inputs to a neural network and judge the accuracy of its results using the known outputs.

The second factor is the time taken to train a neural network, and this is mainly due to how a neural network learns. A neural network learns through a method called backpropagation. This is performed during training and after a result's accuracy is judged through comparison to the known expected output. Once the accuracy of a result is known, one then works backwards from the output layer to the input layer, adjusting the weighting of which each neuron applies to

[2] weeks and even months.

its inputs in order to generate an output (Hecht-Nielsen, 1989). This is done for each neuron on a per-neuron basis as to the backwards propagation technique selected, and as such takes time[3].

The initial problem in neural network training is acquiring this known data used for training (the training set) which contains both known good results, and known bad results; this is so the neural networks can learn to tell right from wrong. There are situations where the information is easy to get hold of, such as the case of facial recognition data. For this one simply needs pictures of faces and an identifier to say which picture or set of pictures belongs to each person. Facebook[4] implements a feature in which uploaded images are checked for faces and suggested tagging on of individuals is then given to the user. The recognition and suggestion of these tags is based on a profile built for a person on previous photos they have been tagged in or uploaded themselves (Facebook, 2016). Gaining the known data for training a neural network in this case is relatively straight forward, however for the application of neural networks it is not.

The main issue is discrimination of valid network traffic against traffic that is involved in a DDoS according legitimate systems' traffic (Yu *et al.*, 2012). For example, a DDoS attack utilizing the full packet's Maximum Transmission Unit[5] (MTU) and spoofing a single source IP may look the same at a glance as another system which also utilizes the full MTU on a network in order to achieve its task. However, observation of the packet's Time To Live (TTL) and flags fields may provide extra insight into the objective of the packet. For this reason expert knowledge about a network is required before generating a learning set for a neural network, and a new learning set should be considered for each new network that the neural network will be applied to.

The next problem is the types of DoS and DDoS attacks (Sections 2.4.5 and 2.7). Depending on what technique is used, the traffic seen can vary drastically, and coupled with this is the fact that there are new types of attacks being implemented on a daily basis. This means that not only does the neural network have to catch up to be able to detect current DoS and DDoS techniques, it also has to keep current, and so requires continuous training during its application life cycle. The issue that arises in this context is that the size of a network packet is limited, and when multiple entities start to occupy the same space they start to get mistaken for each other when that space starts to become full. This discernment will be the biggest hurdle the neural network has to face (Zhang *et al.*, 2001).

All is not grim in the discernment of malicious traffic though, as neural networks do have an inherent feature that can help better indicate the certainty of an attack. The nature of this

[3]Backpropagation is heavily CPU bound (Hecht-Nielsen, 1989).
[4]https://www.facebook.com
[5]Typically 1500 bytes on Ethernet.

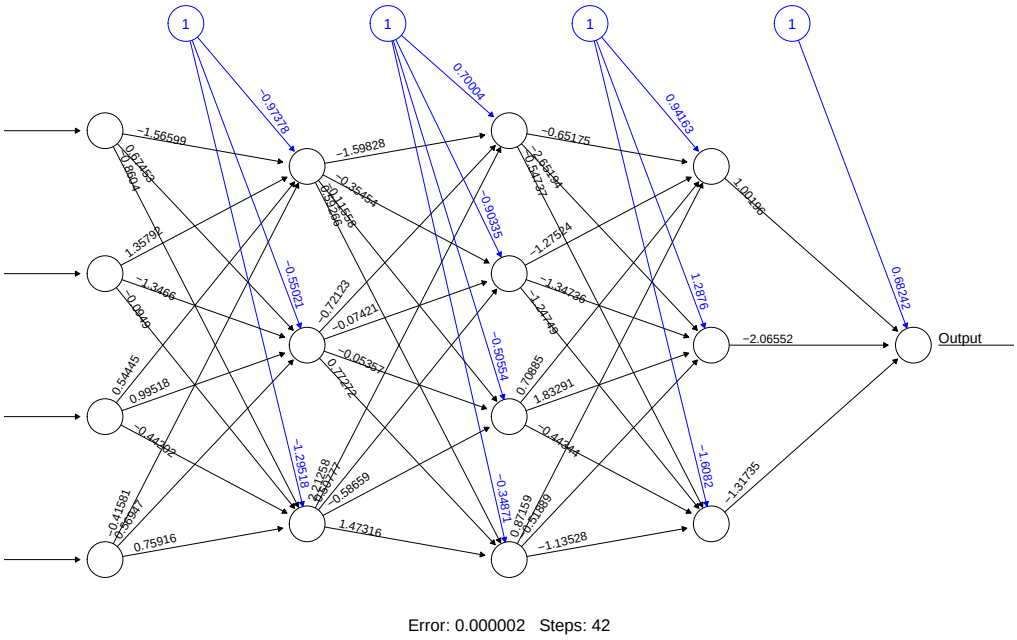

Figure 5.4: Example R Representation of Neural Network

weighted system means that results are never represented as a simple yes or no; a ‘1’ or ‘0’ to speak computer binary. This means that if the neural network is not 100% sure it can output a value to represent its confidence in its result, however this does mean that human intervention may be required in finalizing a decision on a set of traffic (Kosko, 1992).

Considering the aforementioned characteristics used in detection of a DoS or DDoS attack, this being packet count and byte count, NetFlow log fields used for training and later detection of a DoS or DDoS attack for this neural network needs to be decided on. Another consideration that has to be made is the dynamic allocation of IPs on networks and the Internet. As the user of an IP can change, an indefinite suspension of receipt of packets from an IP is not a fair judgement. This means that a forgetting mechanism based on a time-out, or a window-based system, should be put in place to mitigate absolute blocking of traffic from an IP.

Another factor that should be considered is what does an ‘IP’ actually mean. As IP addresses are dynamically allocated[6] on the Internet for most home, broadband and wifi consumers, the question of what does a specific IP actually mean if it does not belong to a known server should be asked. The short answer to this is: not a whole lot, not enough to be meaningful at least without other external inputs such as geolocation or time of day. Because of this, it was decided

[6]Not the case for most servers that exist on the Internet as this would make DNS resolution difficult.

that only the number of unique source IPs seen in the designated time window needs to be known. This allows the neural network to work in a more general case rather than having to be reapplied for each network it is deployed on. With this in mind the following fields available from the NetFlow logs were decided on for the neural networks training and anomaly detection in a specified time window:

- Number of unique source IPs.
- Number of unique source ports.
- Packet count over all logs.
- Average packet size over all logs.
- Most common protocol.

These fields are grouped based on the destination IP as one needs to consider the target that is receiving the potential DoS attack. Collection of training data, training and testing is discussed later in Section 6.2. An R representation of a neural network is represented in Figure 5.4. One should note that R uses external weighting factors to better weight neurons' inputs and these are represented as blue inputs to a layer.

5.1.2 Supporting Neural Network Feeder Program

Once the neural network has been trained it can now be put to work processing filtered NetFlow logs from the publisher and producing meaningful results. This is done through a feeder program programmed in R that uses the *rzmq*[7] library to handle ZMQ sockets. Pseudocode for this process can be viewed in Algorithm 1. At program start, the program first checks to see if the neural network exists and is trained. If not, this program undertakes to ask for a training set to train a new neural network for this processor module before continuing to the next step.

[7] https://cran.rstudio.com/web/packages/rzmq/

```
Input: neural network, incoming processed NetFlow logs
Output: classification of NetFlow log's involvement in a DoS/DDoS attack

if neural network does not exists then
    halt;
end
connect to publisher/s using ZMQ;
while halt signal not received do
    receive NetFlow log;
    store source IP, source port, packet count and byte count;
    lookup port-IP pair buffer;
    if port-IP record buffer does not exist then
        create new port-IP record buffer;
        store new record in new buffer;
    else
        if record buffer contains 5 records then
            eject oldest record buffer;
            store new record record buffer;
        else
            store new record in record buffer;
        end
    end
    if record buffer contains 5 records then
        send record buffer to neural network for analysis;
        output results to user;
    end
end
```

Algorithm 1: Neural Network Feeder Algorithm

After the existence of a valid neural network is confirmed, the next step is to connect to a publisher using ZMQ. Once connected to a publisher this program enters its main loop which is to receive logs and store the source IP, source port, packet count and byte count according to the destination IP which acts as the key. Once the number of logs for a destination IP reaches the user-defined threshold (the default is 5), the logs are then processed into the expected form as defined by the training set in Section 5.1.1[8]. The raw form of this data (stored as a table in R) can be viewed in Listing 5.1.

Once the logs have been formed into information usable by the neural network, these processed

[8]The reason for this threshold is that having too few logs for a communication pair often yielded false results in preliminary testing.

Listing 5.1: Raw Data Passed Into Neural Network

```
unique_ip,unique_sp,pkt_count,protocol,ave_pkt_sz
5,5,17601,17,898
1,3,9,17,74
1,3,11,6,90
5,5,15024,17,948
5,5,13144,6,1054
1,4,9,6,74
1,4,11,6,74
...
```

logs are then passed to the neural network for processing and a result is returned. These results are then presented to the user with information about the findings and recommendations on how the user should act in accordance with these findings.

Once the threshold is reached and a new log for a destination IP is received, this implementation simply forgets the oldest log, thus allowing for a facilitation of the dynamic nature of IPs on the Internet.

5.2 Fourier Analysis

The method in which a function is approximated by the infinite sum of simpler trigonometric functions is known as Fourier analysis, and the Fourier transform of $f(x)$ and its inverse are represented by Equations 5.2 and 5.3 respectively (Bracewell, 1965).

$$F(\alpha) = \int_{-\infty}^{+\infty} f(x) e^{-2\pi i \alpha x} dx \tag{5.2}$$

$$f(x) = \int_{-\infty}^{+\infty} F(\alpha) e^{+2\pi i x \alpha} d\alpha \tag{5.3}$$

A common use of Fourier analysis is in audio processing (Smith, 2007). To elaborate, one can take a known sound wave function and decompose it into its primary components. Using this one can determine what major contributing frequencies make up the sound wave, and from this one can determine the musical notes played in order to achieve the sound represented by the function. This works fine and well for known equations in an analogue environment.

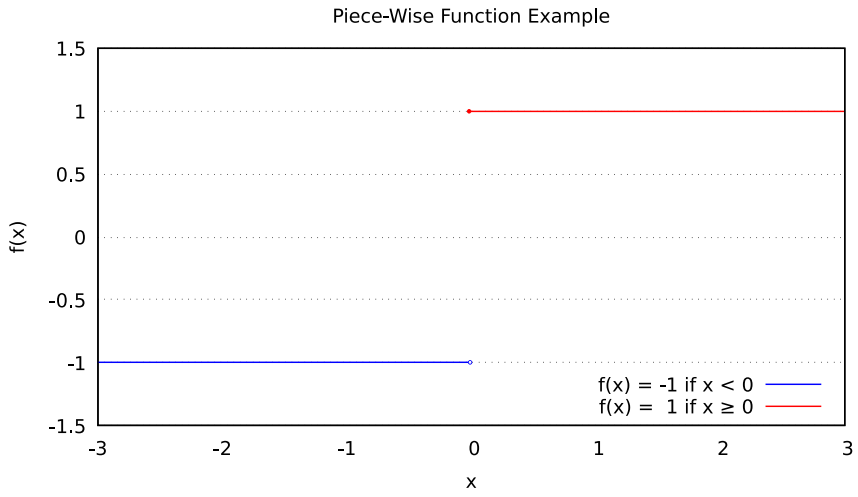

Figure 5.5: Piecewise Function Example (Equation 5.4)

To better understand how a digital computer represents these analogue functions, one can look into how functions can be represented. Functions can be represented by a set of rules, and this is named a piecewise function (Stewart, 2016), an example can be referred to in Figure 5.5 which is made up from Equation 5.4.

$$f(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{5.4}$$

These piecewise functions can be made up of as many rules as one requires and these rules can go as far as to represent a single value at a single dependant value at single independent location. An example of this is shown in Figure 5.6 which is represented by Equation 5.5.

$$f(x) = \begin{cases} -1 & \text{if } x < -1.5 \\ 1 & \text{if } x = -1.5 \\ -1 & \text{if } -1.5 < x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{5.5}$$

Now if one were to consider a simple line on a piece of paper, to reproduce this one could pick up a pen or pencil and continuously press on the paper with the tip and move it around to their heart's content, or one could do something a little more in line with how a computer represents

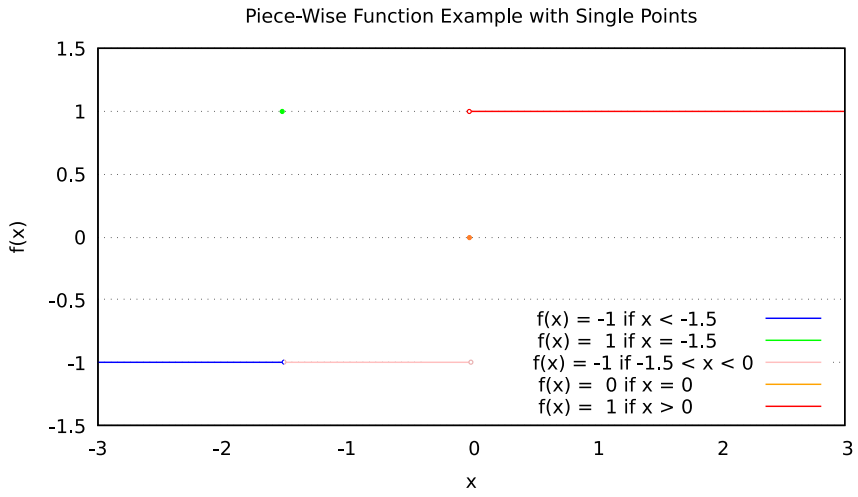

Figure 5.6: Piecewise Function Example with Single Points (Equation 5.5)

Figure 5.7: Example of a Line Made Using Dots

a line. A computer represents any imagery as a set of points and these points can be represented as pixels if one were to display them. If one were to take that same pen or pencil and draw a series of dots close enough together so their sides overlapped by placing each dot's center within another dot, one could draw a line in this manner. A working example of this is shown in Figure 5.7.

This dot representation of a line could be further built upon to show that if one were to make the distance between each dot infinitesimally small, one would effectively include every point in the line that the representing function makes up. Furthermore, one can gain the function of this line in this manner too; this is through the use of the previously explained piecewise function. If one were to represent every point on the line as its own rule in a piecewise function, one would make up a function that represented every point of the line, it would be infinitely large and no one could ever write down every rule, but this piecewise function would work as a substitute to the actual equation that represents the line.

A computer cannot represent a function with a rule set of this magnitude though; the limitations on memory is the first major drawback among others. A computer can however represent

enough points from this function in order to reproduce a very good representation of the function (Roads, 1996). Also, given a set of known outputs to inputs (this being independents with their resultant dependants), one can trivially reproduce the rule set for that dataset for a piecewise function.

The Discrete Fourier Transform (DFT) draws its use from this fact, where one of the binding rules of a Fourier transform is that it requires a continuous function. A DFT can represent the same result for that same function with only N known points evenly distributed by space T on the dependant axis; a representation just like a piecewise function. This resulting accuracy is however dependant on the size of $\frac{N}{T}$. Larger values of this result in higher accuracy, as it means that the point density is higher and therefore the dataset is more likely to contain the finer characteristics of the represented function.

Proving the equivalence of the Fourier transform to the DFT is performed through generalization of the Fourier transform into a sum (Lim, 1990). This generalization can be shown when one considers $f(x) \rightarrow f(x_k)$ by letting $f_k \equiv f(x_k)$, where $x_k \equiv k\Delta$, with $k = 0, ..., N-1$. Following this logic gives the DFT in Equation 5.6.

$$F(x) = \mathscr{F}[\{f_k\}_{n=0}^{N-1}](n) \tag{5.6}$$

This Equation 5.6 can be written more formally as the DFT equation shown in Equation 5.7 and its inverse in Equation 5.8.

$$F_n = \sum_{k=0}^{N-1} f_k e^{\frac{-2\pi i n k}{N}} \tag{5.7}$$

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{\frac{2\pi i k n}{N}} \tag{5.8}$$

Drawing back to earlier text on the use of Fourier analysis to process audio signals, one can now apply a DFT to an unknown audio signal by using the above method. If a system receiving an unknown audio signal were to record a sample every T distance apart for a count of N samples, one could use a DFT to perform Fourier analysis on this signal, as it conforms to the requirements of a DFT (Chen *et al.*, 2002; Mersereau and Oppenheim, 1974). From this point one can draw out the audio signals primary components and work out what the major contributing frequencies are to the audio signal along with other meaning (Marple Jr, 1987).

Table 5.1: Packet Count per Time Slice

Time (ms)	Packet Count
0 - 99	34
100 - 199	2
200 - 299	103
300 - 399	57
400 - 499	93

```
46516 15.572362000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=1/256, ttl=64
47653 16.571770000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=2/512, ttl=64
48900 17.571812000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=3/768, ttl=64
50066 18.571804000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=4/1024, ttl=64
51406 19.571808000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=5/1280, ttl=64
52789 20.571834000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=6/1536, ttl=64
54016 21.571857000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=7/1792, ttl=64
55787 22.571839000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=8/2048, ttl=64
60361 23.571873000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=9/2304, ttl=64
62723 24.571857000  192.168.0.1 -> 192.168.0.100 ICMP 98 Echo (ping) request  id=0x1ab7, seq=10/2560, ttl=64
```

Figure 5.8: TShark Dump of Simple Ping

DFTs aren't limited to only audio signal processing and their primary use is listed under Digital Signal Processing (DSP). Essentially, this means that if a signal's intensity can be sampled at a known sample rate and mapped into the time domain, one can apply a Fourier transform to it to further analyse the signal.

A network packet's 'intensity' can be represented by number of packets or by number of bytes, however, more complicated ways can be devised. As network packets act in real time, one could bucket the chosen intensity recording method into time slices. To better explain, one can choose how large they want a time slice to be and count the number of packets or bytes a network link receives in that time slice and record it. Repeat the process over a length of time and one will gain N samples spaced T apart; or N samples at sample rate T. An example of this is shown in Table 5.1.

A simple applied example would be through analysis of traffic generated by a ICMP-based ping. Using the Ping network utility (Muuss, 1983) yields the result of transmission of a basic ICMP ping packet to the specified destination every second. An example TShark dump of this traffic is shown in Figure 5.8.

Using a sample rate of 100 ms and intensity based on byte count on this traffic, one can now apply a DFT to it to yield the result shown in Figure 5.9. One can note that the global maximum of this graph is found at the 1 hz mark. After this harmonics are found at every 1 hz increment thus serving to emphasise the significance of the 1 hz global maximum. Hertz are determined simply by how many times something happens a second, and it is known that a ping packet is

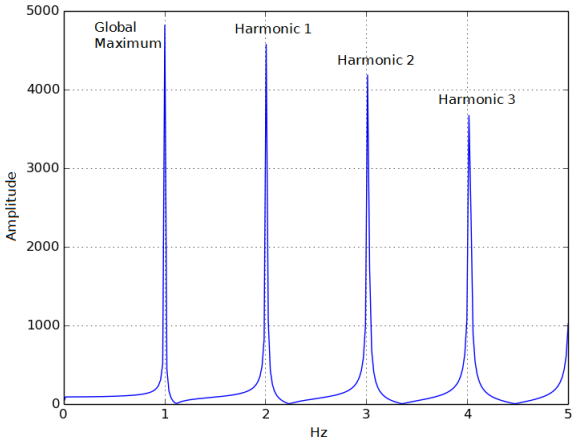

Figure 5.9: DFT of Simple Ping Traffic at Sample Rate 100 ms and Intensity on Byte Count

transmitted once a second. This means that applying a DFT to the data set depicted in Figure 5.8 shows that there is something happening every second, and that something is known to be the ping packets being transmitted.

One can also compare the use of Fourier analysis of unknown signals to network traffic based DFTs as well. Like the prediscussed example of receiving and applying a DFT to an unknown audio signal to get the major components of the signal, one can apply the same methodology to network based traffic too. From here one can find the major frequencies in a set of network traffic, however this alone is worthless.

The reason for the worthlessness of this information can be shown when again considering the unknown audio signal. One can say something about the frequency characteristics of the audio signal because of prior knowledge to analysis of the DFT of the audio signal. One knows what a local or global maximum at a given point means (set frequencies make up musical notes). Before one can say anything meaningful about a DFT of network traffic one needs to know their network and have either a general knowledge of anomalies that they could look for, or a database to refer results to in order to detect these anomalies for them. With the right knowledge base, one could tell a lot from a simple DFT of a day's data.

```
Input: incoming processed NetFlow logs
Output: FFT of data on byte and packet count over varying sized sample sets

if host not CUDA compatible then
    halt;
end
get available VRAM;
calculate and store dataset chunking size using available VRAM;
CUDA kernel compilation;
connect to publisher/s using ZMQ;
while halt signal not received do
    receive NetFlow log;
    fetch byte count and packet count from NetFlow log;
    add each count to their respective bucket stores;
    if bucket store count larger than calculated dataset chunk size then
        break bucket store up into chunks that will fit in VRAM;
    else
        make the bucket store a single chunk;
    end
    forall bucket store chunks do
        load chunk to VRAM;
        run CUDA kernal on chunk;
        store chunk results;
    end
    amalgamate all chunk results on 5 minutes, 1 hour and 24 hours time windows;
    generate graphs;
    update output window;
end
```

Algorithm 2: CUDA-Based FFT Calculator

5.2.1 Implementation

The language that is the work horse of this processor module is CUDA (nVidia, 2007). As mentioned in Section 3.1.2, CUDA is designed to compile for and run on a GPU for use in GPGPU specific applications. DFTs are one of these GPGPU calculations that see benefits from being tasked to a GPU. The reason for this is that as explained, GPUs deal well when applying one instruction to multiple data. A lot of the calculations in a DFT are based on performing the same calculation multiple times on multiple data, and so the benefits of applying a GPU to this task are clear.

There are downsides to the use of CUDA for this task. The main issues of this implementation is that at the time of development CUDA did not have any libraries to interact with network interfaces. This is because it was designed to work alongside programs as a co-processor rather than as the program itself. As this is the case, a parent program was required in order to compile the CUDA program into a kernel and upload that to the GPU in order to perform its task. Following this, the parent program would continue to build the data into a format that could be represented in the aforementioned time domain, and then given to the GPU for processing. Results would be fetched after the data had been processed by the GPU and used for further analysis.

The implementation of a DFT used for this implementation was the Fast Fourier Transform(FFT) which was designed to reduce the time needed to compute a DFT by factorizing the DFT matrix into a product of sparse factors (Van Loan, 1992). The term sparse in this context refers to the nature of the DFT matrix in that it is mostly made up of zeroes. This optimization reduced the computational time required to perform a DFT from O(n^2) to O($n \log n$).

The parent program that runs the GPGPU implementation of the CUDA FFT algorithm was designed to cater for two use cases. These are listed below:

Live: Allows for use on a live system where data is managed and handed to the GPU by the parent at a specified sample rate. Data is received through the ZMQ socket, see Algorithm 2.

Playback: Allows a user to give a previously recorded traffic capture to the parent program which will then process the data according to the specified sample rate and then pass this data set on to the GPU for calculation. This replaces the ZMQ interface with a read from file interface. In Algorithm 2 this replaces the functionaility at "*receive NetFlow log*".

As previously discussed in Section 5.2, a higher sample rate and sample count leads to better results when computing a DFT. For this reason the more frequently NetFlow logs are ejected to the collector that will eventually feed this processor, the more accurate the results will be. This is due to the nature of NetFlow in that it only logs out a NetFlow packet if the connection completes (for whatever reason), or a time out occurs in which the log is ejected immediately. For this reason a low time out is suggested, however one should note that this leads to higher bandwidth requirements by the NetFlow source.

A concern during this implementation was too much data. If one monitored too many NetFlow sources, the issue of overlapping frequencies could affect results. For this reason it was decided that this process would only listen for a single source's NetFlow logs which is easily discernible

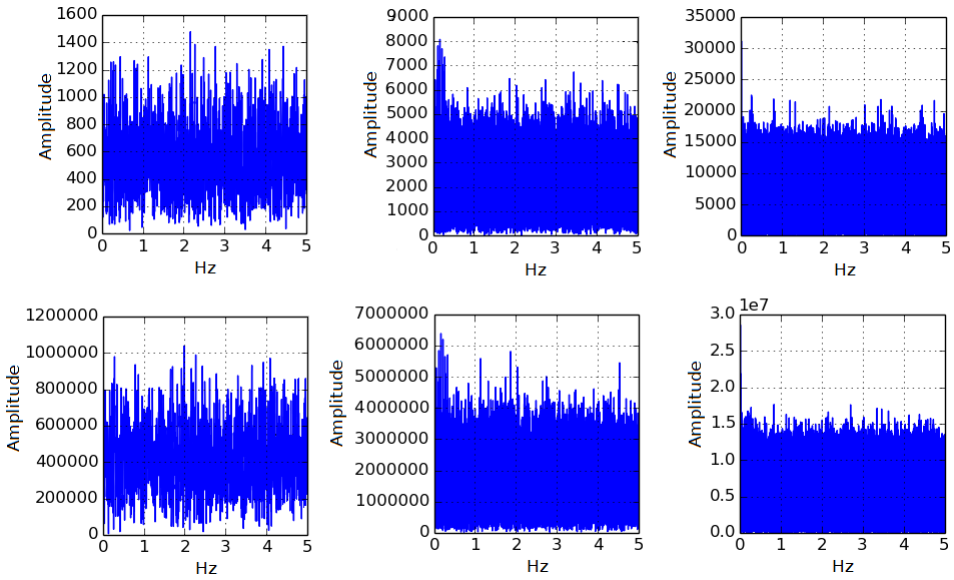

Figure 5.10: Example Output of FFT Implementation

by the NetFlow source ID. Monitoring multiple NetFlow sources using multiple systems does not affect the overall performance of this system as the overheads of scaling out is handled solely by the use of ZMQ as a data distribution platform.

This does not in any way mean that performing DFTs over a larger data set is a not appropriate. Where Fourier analysis on a single link may help to detect and further analyse specifics of a segment of one's network, a DFT of groups of NetFlow sources may help in detection of anomalies on a larger scale.

Figure 5.10 shows the default output of this processor. Network traffic can be accounted for in terms of packets and in terms of bytes, both are presented to the user for analysis as both can tell different stories. This is because multiple packets can fit in 1,000 bytes, however a single packet could also be a 1,000 bytes.

To keep things simple the display was grouped into a top row based on packet count, and a bottom row based on byte count. These rows were then split into 3 columns which represented the time window each graph represented. By default column 1 is a 5 minute time window, column 2 is an hour time window and column 3 is a 24 hour time window.

To save on processing resources in terms of both CPU and GPU, these DFTs are only rendered when new relevant data is received from the publisher. This unfortunately leads an unaware OS to believe the process has halted, and thus it treats the window in its default manner for halted process. This includes greying out of the window on a Linux system running a Unity desktop

Table 5.2: Time to Complete Long Scan for Varying Values Between Probes

Time (seconds)	Time to Complete	Time to Complete (days)
30	8 h 20 min	0.347
60	16 h 40 min	0.694
120	33 h 20 min	1.389
180	50 h	2.083
240	66 h 40 min	2.778
300	83 h 20 min	3.472
600	166 h 40 min	6.944

manager, or on a Windows system, Explorer will notify the user by an overlay that a program has stopped responding. Waiting for the next set of data from the publisher resolves this.

5.3 Port Scan Detection

The goal of this processor was to develop an algorithm which could detect that a host was being targeted by any of the port scanning techniques mentioned in Section 2.4.2. This goal was then extended to attempt detection of port scans performed where probes were a large amount of time apart. For the purpose of this section these large time gap port scans will be referred to as long scans. This form of scan is used by attackers to hide their activities as the scan would not be performed all at once.

One must note that a typical port scan probes the most commonly used 1,000 ports on the Internet (Abramson, 1985; Lyon, 2009). When considering this number and a delay between probes of 2 minutes[9], this totals 2,000 minutes, or 33 hours and 20 minutes. Referring to Table 5.2 one can find the total time to complete long port scans for varying probe delays.

Having a 10 minute time between probes seems unreasonable but one should note that this only takes a week. If an attack only has one system in which to perform the task, or wants to keep as low profile as possible in order not to raise alarms before a vulnerability is even brought to light, a week or more is viable (Dabbagh *et al.*, 2011).

The viability of this is driven further by the life-cycle of systems on the Internet. If a FQDN is allocated to a dynamic IP, this is not a problem to a potential attacker because it means one of two things. Either the administrator of the FQDN is updating the IP belonging to the FQDN in which a probe would find the targeted server, or the FQDN will point to the wrong IP in which

[9] A fairly large wait on an active network.

the server has brought itself offline. Depending on the attacker's goals, this is either the intent in which the attacker has achieved their goal, or there is nothing more the attacker can do.

Systems on the Internet also try to obtain the best high availability rating they can. This involves trying to keep a system online and available to serve users for as much time as possible in a year. Typically this is performed by identifying single points of failure in a system and making sure that point does not fail, and in the event that it does, that the system recovers quick enough to not disturb the user (Piedad and Hawkins, 2001).

For the above reason, an attacker can expect a system to be up for long periods of time. Furthermore, once a system is deployed and is deemed functional it enters its maintenance phase of its life-cycle causing most functionality will be set. Any updates to the system are most likely to be running fixes and not major changes to the operations of the system. This means that an attacker has time in the order of weeks, months and in some cases (like that of the Network Time Protocol (NTP) server amplification attacks (Czyz *et al.*, 2014)) even years to research and develop an attack on a specific functionality of a system.

5.3.1 Implementation and Configuration

One of this implementation's goals was to leverage current technologies to access computing power available on current systems. These help address problems based on CPU power required to follow the many connections a network may have, as well as memory limitations allowing this system to track more data on more connections for longer. The effectiveness of this however, will depend on the CPU and memory resources of the processor's host, and as such configuration for the system will be allowed in terms of IP grouping, time window to record and detail of recording.

The data collected by this processor relies on Python to complete all main computation, and is designed as a multi-threaded environment. The threads of execution are listed below and will be detailed later:

Anomaly Detection: Attempts to classify NetFlow logs into malicious and non-malicious connections based on IP pairs in a connection.

Port Cleaner: Keeps port records in order, arranged from newest to oldest, counts unique ports and ejects old or irrelevant port data.

IP Pair Cleaner: Keeps IP pairs in a connection sane, ensures return data from the target is not logged, expunges old records.

Input: port count threshold, record time out, incoming processed NetFlow logs
Output: notification of the existence and IPs involved in possible port scans on a network

```
connect to publisher/s using ZMQ;
configure port cleaner;
start port cleaner thread;
configure IP pair cleaner;
start IP pair cleaner;
while halt signal not received do
    receive NetFlow log;
    extract source IP, destination IP and destination port;
    extract byte and packet count; generate time stamp;
    if have not seen source IP then
        create new port record for source IP and add destination port to it;
        create new IP pair record using source and destination IP to record connection
         direction;
    else
        if destination port does not exists in port record for source IP then
            add port to port record for source IP;
        end
        if IP pair does not exist for source and destination IP then
            create new IP pair record using source and destination IP to record connection
             direction;
        end
    end
    using the IP pair and port record list for the source IP, determine whether the NetFlow
     log was generated as a response to the destination IP;
    if a response then
        output "response detected";
    else
        if packet count greater than 10 then
            Comment: this is more than the packets used by any of the scanning tech-
                     niques this processor module is attempting to detect
            output "packet count exceeds maximum detection threshold";
        else
            if unique port count for source IP greater than threshold then
                perform further analysis;
                output "likelihood of source IP's involvement in a port scan";
            end
        end
    end
end
```

Algorithm 3: Port Scan Detection

Referring to Algorithm 3, at process start a connection is attempted to the publisher, if this is successful the two cleaner threads are initiated and the host system is analysed for later use in thread allocation. Once these have initialized the main loop of execution begins, the anomaly detector. This process at an abstracted view simply receives published NetFlow logs, appends the needed data to existing records (or creates them anew) and then performs general anomaly detection on them. Factors used in detection are listed below:

- The number of ports a source IP has attempted to connect to.
- The number of ports a destination IP has received attempted connections from.
- Which ports are these attempted connections trying to connect to.
- Are the port connections part of the common 1,000 ports scanned.
- Is the log a response to a scan.
- Is a scan made being performed from multiple hosts.

After taking into consideration all these factors, a weighting is given to the received log based on the previously received logs that are relevant to the connection being analysed. This is then used in determining whether to report a source IP's possibility of being involved in a port scan, or not.

Port and *IP* cleaners are similar in the way they start execution, but after this point they vary in what they aim to achieve. Both start by analysing the number of tracked records and the detail at which these records are tracked. This varies during runtime of this processor. Using this analysed data along with the collected start time information about the underlying host system in which this processor runs on, a number of threads are started and a work size is determined for each thread.

From this point each cleaner gets to work at achieving its own task. The port cleaner at its core uses the provided system start configuration to determine whether records should be kept and how they should be kept. This configuration data is given at start time and includes the depth at which ports should be remembered per IP pair (where an IP pair is made of a source and destination IP), the number of ports to remember, and what amount of time a port should be remembered for.

Similarly the *IP pair* cleaner uses its own configuration to determine how many IP pairs should be stored and how long these IP pairs should be stored for. It also performs further analysis

into what IP pairs were generated due to responses created by an attempted connection, and further determines whether these are necessary for storage based on memory limitations of the underlying host. In the basic case this is performed by searching for an immediate response to the source IP and port in the IP pair.

Keeping the stored information collected from published logs helps in more effectively and accurately detecting possible port scans. It also allows for better memory management and the use of multiple threads that are adjusted to the available resources of the underlying system, allowing for the application to tailor itself to best use the underlying system's resources.

Given a configuration time which aims to log data using a large time window, and underlying hardware that has enough memory and CPU resources in order to successfully achieve the requirements of this configuration, one can effectively use this processor for detection of long scans. Simple fast-as-possible port scans are also easily detected by this processor, and configuration can be set to better utilize system resources for this task as well.

5.4 Sudden Port Bandwidth Use Change Detection

The sudden increases and decreases of activity on a port of a system or a network can be an indication of anomalous behaviour. For this reason a module was created that keeps track of the running average of incoming and outgoing traffic on port-IP pairs, and notifies the user when the behaviour of a pair deviates from the running average. To better explain the use of such a module, consider these three cases:

1. A port suddenly starts transceiving network traffic.
2. A port suddenly stops transceiving network traffic.
3. A port that has a known transceiving characteristic suddenly deviates from the norm.

In the first case, a port that on average sees no traffic inbound or outbound from it, and is not used by any process on a system or network in a normal operating environment, should never deviate from a running average of no traffic. If this occurs due to no premeditated plan, be it upgrade of software or hardware or a new service installed on the system, this is reason for alarm and at least investigation of the cause of this anomalous traffic (Walter, 2012). This is much easier to detect at a per system level when compared to network wide monitoring. The reason

for this is that if a system has a known IP with known processes running on it, the expected functionality of the system is well known, and thus deviation from this expected functionality is easy to detect. At a network level things become a bit more complicated. Simply having an IP reassigned to a different host, or a system leaving the network, would cause sudden changes in network activity on a particular port and IP.

The second case is typical of a service going offline. If a service is set to run on a specific port-IP pair and it is reported to have a sudden decrease in traffic transceived from that port-IP pair, this serves to reason that the service is no longer accessible for some reason. The reason for the service no longer being accessible falls out of the scope of this module, as this module is intended to simply notify a user to a change.

The last case given here refers to either a sudden increase or decrease in traffic to a port. An example of the former can be shown through consideration of a DoS or DDoS attack. If a system providing a service is known to receive on average 50 Mbps of bidirectional traffic at a given time of day, and this suddenly increases to the link saturation point, this is a cause for an alarm, and would be detected by such a proposed module. It is also true that data extraction or insertion can also exhibit this behaviour when considering data upload or download (Liu *et al.*, 2009).

If there was a sudden decrease of traffic to and from the service to 5 Mbps, this is a good indication of service degradation of the medium in which traffic is being transceived on the given network link (Jaiswal *et al.*, 2004).

This proposed module does little more than notify a user of a change in behaviour of a port-IP pair on their network, however these changes are usually a good indication of something larger at play that may otherwise go undetected.

5.4.1 Implementation

The implementation of this module is based on some simple statistics involving the mean of a dataset and its standard deviations. A standard deviation is a value that when, added and subtracted from the mean of a set of data, creates a range around the mean in which all data should fall within, within a normally distributed dataset. The standard deviation is of course calculated from the dataset in question.

If one takes a single standard deviation and creates a range around the mean with it, this is known as 1 standard deviation of the mean, and 68.27% of all data in the dataset fall within this

```
Input: configuration file, record count, incoming processed NetFlow logs
Output: notification as to sudden changes in bandwidth use on ports and IPs within a
        network

read in configuration file and extract the subnetwork blocks and ports to monitor;
connect to publisher/s using ZMQ;
while halt signal not received do
    receive NetFlow log;
    extract source port, destination port, source IP and destination IP;
    extract byte count;
    if source or destination IP exists in the monitored subnetwork and port is monitored
     then
        extract direction using source and destination IP;
        if record of port-IP pair does not exist then
            create a new record store for port-IP pair;
        end
        using the current record store for the port-IP pair calculate the mean and standard
         deviation for that port-IP pair;
        use new NetFlow log and compare to mean and standard deviation using byte
         count, port, IP and direction;
        if new NetFlow log falls outside of 3 standard deviations then
            output warning to user;
        end
        if number of records in port-IP record store is equal to record count then
            expunge oldest record from port-IP record store;
        end
        add new record to port-IP pair record store using byte count, port, IP and direction;
    end
end
```

Algorithm 4: Sudden Port Bandwidth Use Change Detection

range. Using two standard deviations to calculate the range around the mean will yield the range in which 95.45% of all data can be found. And lastly for this module, 3 standard deviations of the mean yield a range in which 99.73% of all data points in the dataset lie (Grafarend, 2006). An image better depicting this text can be referred to in Figure 5.11 and is referred to in statistics as a Bell Curve.

Using this information one now has a method in which to classify data as being expected, or normal, by checking if it falls within a number of standard deviations when new data is presented. Anything that falls outside of this range is known as an outlier. An outlier is simply a data point that falls outside of the normal distribution range of the dataset determined by

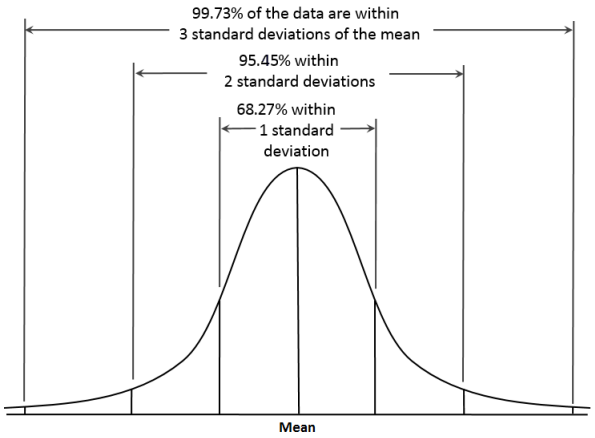

Figure 5.11: Diagram Depicting Standard Deviations: Normal Distribution

the standard deviation. These outliers are anomalous data points that vary from the norm; something this module is aiming to detect.

Applying this concept to a processor module, one can use this statistical theory to create a mean for transceived traffic (the average bandwidth consumed by a port-IP pair). One can then window this dataset in a time domain based on a set of the most recent data points to make this a running mean (the running average). Using this windowed dataset one can generate the standard deviation, and using 3 standard deviations of the mean for the dataset one can generate the range in which 99.73% of the data points should lie. The pseudocode for this processor module can be viewed in Algorithm 4.

The use of such a method can however become problematic for network traffic that does not follow a standard distribution, that is traffic that follows random patterns. For this reason additional functionality has been applied to allow for exclusion of a port-IP pair from monitoring if it is know to be of such nature. The control limits for the working data set is determined by the maximal and minimal bandwidth rates available to the host which is monitored.

This information is then used when a new data point arrives, to see whether the data point falls within 3 standard deviations of the mean or falls outside of it. In the former case the data point (bandwidth used by the port-IP pair) is seen to be normal and thus expected. In the latter case the data point is an outlier and therefore anomalous; a cause for alarm. Implementing this module in this way will allow the module to be able to detect sudden changes from the normal characteristics of a port-IP pair and notify the user of such an event.

Listing 5.2: Sudden Port Traffic Increase/Decrease Configuration File

```
192.168.0.0/16     # Network blocks to monitor
10.42.1.0/24
*                  # Separator
80                 # Ports to monitor
22
23
8080
443
445
21
25
20
```

5.4.2 Configuration

The downside of this implementation falls to a restriction in memory. One can not monitor every port on every IP address in the entire range of both[10], as at the time of writing there is simply just not enough memory in any known system. For this reason the approach applied to this module was to simply specify what subnetworks to monitor, as well as what ports to monitor for all subnetworks. This implementation also allows for monitoring of a single host, as defining the configuration of an IP in a /32 block would result in just that IP being monitored. An example of a configuration file is provided below:

This file is made up of two parts. The first part is which IP subnetworks should be monitored and the second is a list of all ports that should be monitored on the defined subnetworks. These two parts are separated by a '*' character. The dataset window sizing parameter is set at the start of runtime, and cannot be changed without a reboot of the entire module.

One should note that reception of network traffic on every network port and every IP on a network is highly unlikely. Referring forward to Table 7.4, this table contains the IP blocks allocated to the monitored real-world network used later in this research. The main point to draw out of this table is the fact that there are 196 assignable IPs on this network of which not all are assigned.

[10]This would result in $2.814749767 \times 10^{14}$ unique pairs in the IPv4 space alone. This is a highly unlikely scenario.

This means that the maximum number of port-IP pairs that can be stored for this network is 12,845,056. The information stored in a port-IP pair in this processor module is as follows:

- Time stamp
- IP
- Port
- Last 100 byte count records stored for the port-IP pair (default configuration).

Diving into a bit of code one will notice that the size of a Python *datetime* object is 48 bytes (refer to Listing **??**), the IP is simply a 4 byte integer, the port number is a 2 byte integer, and finally each of the byte count records are 8 byte integers. This means that a fully populated port-IP record is a total of 854 bytes in size. Multiplying this through by the total port-IP pairs that can be stored for this 196 node network (if all IPs are assigned), one would require 10.216 GB. This is an achievable requirement at the time of writing. This being said, as the monitored network blocks are not fully allocated, the memory requirement for testing in Chapter 7 will be less than this calculated maximum.

5.5 Reputation Analysis System

Discovering the source of a problem through analysis of data can take time, and this often leads to delays before mitigation of the problem occurs. Mitigation of an attack can be performed by stopping the source from continuing to perform its attack at a target, however discovering what the source is of the attack can take time. This is due to the fact that if packet sources are spoofed in an attack it can mislead mitigation through use of the source IP[11].

In Section 5.6 the NetFlow source ID is used to determine where in the world a NetFlow log is being generated from, through use of geolocation. If a system were put in place to use the same NetFlow source ID to also log data about what attacks are common to different NetFlow source IDs, one can start to profile networks these NetFlow sources reside on (Pennefather and Irwin, 2014). Using this information one can place an attack source at a location of a network without the need for the source IP address; this increases in usefulness the larger a network gets.

Having this information can lead to a good first guess at where an attack is originating from and what it is. For example, if there is a DDoS attack in which attackers use spoofed source IPs

[11] Please note that this work is previously publish as part of Information Security South Africa 2016 (Herbert and Irwin, 2016a).

Input: processor module output
Output: none

open ZMQ pull interface;
notify processor modules of pull interface;
while *processor module subscription count greater than 0* **do**
    receive push processor result;
    lookup metadata tag as to what the result is;
    store result for later use in relevant table;
end

Algorithm 5: Reputation System Data Capture

(Section 2.9), blocking packets from these sources will do nothing more than block off legitimate users of your service (JAVAPIPE, 2017; Argyraki and Cheriton, 2005). However, if one knows that DDoS attacks that follow a given packet size with spoofed IPs always have NetFlow logs generated from a set of NetFlow source IDs, one can begin to mitigate an attack by starting to block off packets from networks whose traffic passes through these relevant NetFlow sources[12].

Although not the complete solution for system defence, being able to at least work out a most likely source of attack is a first step that can be quickly concluded from past events while other systems perform their analysis.

5.5.1 Implementation

As this processor module does not perform analysis itself, it is not intended to be placed with processor modules of this system. Because of this, this processor will instead be referred to as the reputation system. This reputation system will be fed information from the results generated by processor modules and store this information for later use if the need arises. This functionality is outlined in Algorithm 5. Its place in this system from a high-level overview can be seen in Figure 5.12.

There is not much more to the reputation system than an open network socket that takes in data of a set format and stores it in a database. This data acts as the input to the database and

[12]The method one intends to perform this traffic blocking is out of the scope of this research.

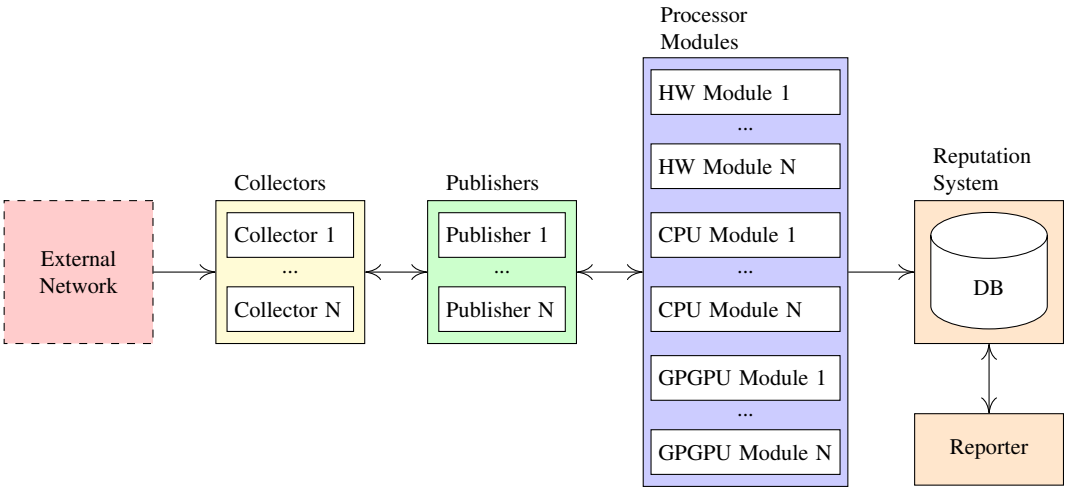

Figure 5.12: Updated High-Level System Overview with Reputation Subsystem

Table 5.3: Reputation System's Database Table Format

Field Name	Data Type	Description
Unique ID	Integer	Simply a unique identifier for an entry
NetFlow Source ID	Integer	The NetFlow source ID which the NetFlow log was generated from
Attack Type	Integer	The type of anomaly or attack detected by a processor module
Attack Score	Integer	The confidence the processor module had towards the detection of the Attack Type
Metadata	Blob	Any extra information the processor module wants to store about the anomaly or attack
Time Stamp	Datetime	The data and time of when this log was inserted into the database

these inputs shall now be discussed. It is also noteworthy that the reputation system will receive packets from multiple processors, and so considerations to code efficiency should be taken into account.

This processor module is implemented using C and SQL and the combination of these two languages allows for highly efficient and effective manner of data retrieval and storage. However, this data has to be meaningful and simple in order to allow for a quick retrieval of results that are accurate. To achieve this, the fields used to store information about a detected anomaly or attack from a processor presented in Table 5.3, is accompanied by the database data type of each field.

These fields give the most concise storage of information produced by processor modules by

simply storing the results produced. One can use attack types to home in on a set of attacks. From here one can set a time window to look at, and use confidence levels to further whittle down the results. After this one can then decide on which parts of a network to focus mitigation on, based on the returned NetFlow source IDs, which will point to the locations of a network that the anomaly or attack is originating from.

Lastly it must be noted that the reputation system is not a requirement of the overall system and does not need to be run at all if a user does not deem it necessary.

5.5.2 Configuration

Configuration of this reputation system within Bolvedere is simply performed through specifying a database in which to use, and then starting the pull interface on the ZMQ handle within this implementation. After this interface is successfully created, the Bolvedere processor modules are notified that the reputation system is ready to receive detected malicious network flows. At this point the reputation system pulls processed records' results from the processors and couples the information with the geographic information gathered from referencing the NetFlow source ID received in the original NetFlow log.

5.6 Source IP Address Anomaly Detection

If a network's NetFlow sources are correctly configured, each one should have a unique NetFlow source ID associated with it (Cisco, 2003a). This means that one can tell where a log is being generated from on a network by simply keeping track of which NetFlow source ID is assigned to each NetFlow source, and by using this information to find where the log originated from.

This can be used in a simple test to further help detect anomalies on a network regarding spoofed packet source IPs (Templeton and Levitt, 2003). This algorithm simply watches NetFlow logs generated from the internal interface to the external interface of a NetFlow source. When a log is received, a check is performed to determine whether a packet sourced from the NetFlow source's internal interface belongs there. This can be done by simply looking up what IP blocks are assigned behind the internal interface of the NetFlow source and comparing this to the NetFlow source ID generating the log.

```
Input: configuration file, incoming processed NetFlow logs
Output: notification of possible spoofed packets sourced from a monitored network

read in configuration file and extract which subnetwork is sourced from which NetFlow
  source's internal network;
connect to publisher/s using ZMQ;
while halt signal not received do
    receive NetFlow log;
    extract NetFlow source ID, source IP and interface ID;
    if flow is sourced from internal network then
        lookup NetFlow source ID's internal subnetwork block;
        if IP source does not belong to NetFlow source ID's internal subnetwork block
          then
            output warning to user;
        end
    end
end
```

Algorithm 6: Source IP Anomaly Detection

If the source IP is found to not fall into a network block assigned to the internal network that the NetFlow source is logging, this would mean that the packet with the source IP in question should not have been transmitted from that internal network. This will allow one to better manage the validity of networks, keep a physical system structure as to which system belongs on which network, and help in detection of malicious activities such as attacks using spoofed IPs.

As an example, consider a network (referred to as the internal network from this point) with all IPs assigned in the 192.168.42.0/24 subnetwork block, which is connected to an external network through a firewall system that also acts as a NetFlow source. With such a configuration, one would expect all packets generated from the internal network and passing out to the external network through the firewall system to have a source IP within the 192.168.42.0/24 subnetwork block. If a source IP outside of the assigned subnetwork is observed, one should be alarmed.

5.6.1 Implementation and Configuration

As this algorithm achieves a simple task, the implementation of this processor was kept simple as one can see in Algorithm 6. In order to run this processor one needs to construct a configuration file that describes which IP blocks belong to which NetFlow source ID. The format

Listing 5.3: Source IP Anomaly Detection Template File

```
1 # Unique NetFlow ID, Subnet block
2 100 193.151.192.0/19
3 100 146.0.0.0/8
4 200 158.72.0.0/16
5 200 146.0.0.0/8
6 300 212.2.0.0/19
```

of this configuration is a list of two elements with one entry per line. An entry consists of a NetFlow source ID and an IP block. This is then used to build a rule table of what source IPs should be expected from a NetFlow source based on its NetFlow source ID. An example of this configuration file is shown in Listing 5.3.

Each NetFlow source ID is configured disjointly to any other, and so any subnetwork that is common to multiple NetFlow sources is required to be individually configured for that NetFlow source ID. An example of this can be seen at Lines 3 and 5 in Listing 5.3.

Other than these rules, nothing else is required for storage during runtime of this system. This is because every log received is treated disjointly as no previous or forthcoming log received can change the outcome of this form of stateless test. This means that the memory requirement of this processor module is only what is needed to store its configuration.

To continue detailing the runtime of this processor module, after the configuration script is processed, the system then attempts a connection to relevant publishers. All filtered logs received from these publishers has the configuration rules applied to them, and if a source IP is detected to not belong to a NetFlow source ID's assigned network block, the anomaly is reported.

5.7 Malware NetFlow Fingerprinting

One of the questions asked by this research is how much information is enough to be sure about an event on a network. This module aims to help answer this question through analysis of NetFlow data records generated by different malicious activities on a network. The data collected from these known attacks are then stored as fingerprints for use in detection of later attacks (Yen and Reiter, 2008).

It was also understood that a non-malicious network flow can give the same NetFlow log results, as NetFlow log generated from a malicious flow. It was decided that dealing with false positives

```
Input: known fingerprint database, incoming processed NetFlow logs
Output: notification of matched exploitation fingerprint

load fingerprint database;
connect to publisher/s using ZMQ;
while halt signal not received do
    receive NetFlow log;
    extract NetFlow source IP, destination IP, source Port, destination Port, interface ID,
     byte count, packet count and protocol;
    Comment: As most fingerprints occur over multiple flows, the running state window
             contains a history of the previous 10 (by default) flows for a given source
             and destination IP.
    if running state window for source and destination IP does not exist then
        create new running state window for the source and destination IP;
    end
    if running state window for source and destination IP's count is equal to 10 then
        remove oldest state item;
    end
    add new state item to running state window using extracted information;
    forall known fingerprints do
        compare running state window to current fingerprint;
        if match found within set confidence level then
            output notification to user;
        end
    end
end
```

Algorithm 7: Fingerprint Matching

was better than dealing with false negatives in this system, as the latter case would mean that a successful attack would go unrecognised. The results of these true and false positives would both be presented to the user with all related information for the user to discern by one's self as this module is intended to notify the user and not take mitigating action[13].

5.7.1 Implementation

The major potential bottleneck identified in this module was on the rule set processor. In order to mitigate this bottleneck the use of a SQL database was decided on as this provides the quick

[13]Please note that this work is previously publish as part of Information Security South Africa 2016 (Herbert and Irwin, 2016a).

access to the rule sets used in discerning whether a network flow is malicious or not. There are many variants of SQL databases which range from a file on disk, as SQLite[14] implements, to entire databases loaded into Random Access Memory (RAM) to achieve maximum throughput, as MemSQL[15] implements. Given that these implementations all share a common language, one can swap out the back-end of this module according to the needs and/or limitations of the host system.

The rest of the software which applies the rule set to incoming NetFlow logs of this module was written in C and a Python-based prototype also exists, the pseudocode for these implementations can be found in Algorithm 7. An incoming processed NetFlow data record received by this module would then have key features extracted according to what each fingerprint is looking for. The most common features are:

- Flow direction
- Source port
- Destination port
- Flow byte count
- Flow packet count
- Protocol

If a set of NetFlow data record matches a fingerprint, a notification of the type of attack detected is raised to the user. As explained before, this can lead to false positives, and this can be alleviated at configuration time through tweaking of configuration information, as the rules generated during the fingerprint creation phase also have confidence levels associated with them. These confidence levels are calculated based on the variance between each field watched during a known attack. This information is then used to define a range rather than a specific value for a fingerprint field. The fields which vary the most in these attacks are packet count and byte count, and so these will typically be represented by the aforementioned range, which can later be tweaked. Fields such as destination port are usually constant (due to a service always running on a specific port) and so will see a set figure required to match in the fingerprint.

Finally, it is to note that fingerprints for failed attempts are also stored. A failed malicious attempt on a system would generate a NetFlow data record that can also be compared against

[14] https://www.sqlite.org/
[15] http://www.memsql.com/

in the database. This requires little overhead if the database is indexed, and allows one to know what kind of attacks are being used against one's systems and/or networks in order to better focus on what is important in their defences.

5.8 Summary

This chapter draws out the scope of the problem that Bolvedere can solve through the use of its modular design, which varies from simple rule matching to Fourier analysis. If one can take anything away from this chapter, one should note the varying hardware architectures and software languages used throughout this chapter that are completely compatible with Bolvedere's collector and publisher.

To summarize this chapter, **Section 5.1** starts off by showing the use of neural networks in DDoS detection using R as its primary language. Some heavier mathematical computation is then required by the Fourier Analysis module in **Section 5.2**, and makes use of GPGPU technologies via CUDA. Port based anomaly detection modules were then designed and developed in **Sections 5.3** and **5.4**, both were Python-based and applied simple statistic methods in order to achieve their respective purposes.

A reputation database was then proposed in in **Section 5.5** that would better indicate the source of a malicious attack on a network, and showcases the use of SQL in a Bolvedere module. **Section 5.6** shows that retaining the geographic information of where a NetFlow source is implemented, and linking that information up with its unique ID, can be used to better locate the source of spoofed IP packets exiting an internal network; this module was implemented in C. This chapter finally closes with **Section 5.7** which explains the workings of a basic finger printing module that makes use of pre-recorded vulnerability exploitations to detect further occurrences of these attacks. This module again leveraged SQL and C.

Chapters 6 and **7** follow on from this chapter putting the design and implementation of Bolvedere and its processor modules through proper testing. The testing performed in **Chapter 6** ensures that Bolvedere works properly as a whole in a controlled environment with known inputs and outputs, whereas **Chapter 7** puts Bolvedere to use on a real-world dataset in order to evaluate its real-world application.

6

Testing

THIS chapter presents results to the reader concerning the validity of output from the Bolvedere Base system as well as processor modules compatible with Bolvedere. Testing also proves proper functionality of each of these components of Bolvedere. This chapter begins with **Section 6.1** which simply tests whether the NetFlow records arriving at Bolvedere's collector and being transmitted from Bolvedere's publisher are correct and as intended. Following this **Section 6.2** through to **6.8** all test the proper functionality and results of processor modules that perform analysis for Bolvedere on NetFlow data records.

Section 6.2 tests the first processor module, which attempts to perform the well researched task of DDoS detection through use of NetFlow records. Following this, **Section 6.3** attends to frequency analysis techniques used on traditional full-packet analysis, and applies them to NetFlow records. In **Section 6.4** one can find the results to a processor module that keeps track of subnet's reputations based on type of activity based on Flow Source IDentification (FSID). **Sections 6.5** and **6.6** test the ability of the Bolvedere processor modules to detect port scans and sudden port characteristic changes of hosts on a monitored network. **Section 6.7** shows of Bolvedere processor module's capability to detect spoofed network traffic. Finally **Section 6.8** looks at fingerprinting common system vulnerability exploitations and reapplying these fingerprints for further detections of repeated exploitations.

These section's will be presented in the following format:

- Description of test.
- The relevant testing environment.
- Results obtained.
- Analysis of obtained results.
- Any figures will be set in a floating style.

Hardware used for testing, unless otherwise specified, was an Intel DQ77MK Motherboard with Intel Core i5-3570K Processor[1] at 3.4GHz, 8GB of 1333Mhz RAM and an Intel E10G42BT X520-T2 10Gigabit Ethernet Card[2] for main software bulk mode tests; this level of host was thought to be well rounded for general testing of this system. The switch used to connect hosts to each other was a Dell PowerConnect 6248 with a Dell P/N P623D 10Gbase-T expansion module. Hosts which ran the bot software [3], as well as acted as gateways for other systems on the created physical environments used in testing, were Foxconn N15235 all-in-one systems. Other hardware or systems used during testing performed in this chapter will be specified in the relevant section's environment. All systems used in testing were running Ubuntu Linux 14.04 LTS [4] and the version of Softflowd used by the gateways, where applicable, was 0.9.9.

6.1 The Base Bolvedere System Work

The base of the Bolvedere system referred to in this section when referring to Figure 4.1 consists of the collector and publisher. The ability for these two subsystems to fulfil their roles, as well as communicate with each other consistently in an expected manner, is paramount to the success of this system. If either of these two components were to fail in any way, this would cause failure in subsystems further down the logic tree. This will of course lead to mis-identification and classification of network flows by Bolvedere processor modules. This testing will focus on the ability of these bases subsystems to receive NetFlow packets, and correctly output processed NetFlow data records to processor modules. Testing of this base system will address each subsystem separately and then the entirety of both subsystems.

[1] http://ark.intel.com/products/65520
[2] http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/ethernet-x520-t2.html
[3] Custom written by the author.
[4] http://www.ubuntu.com/

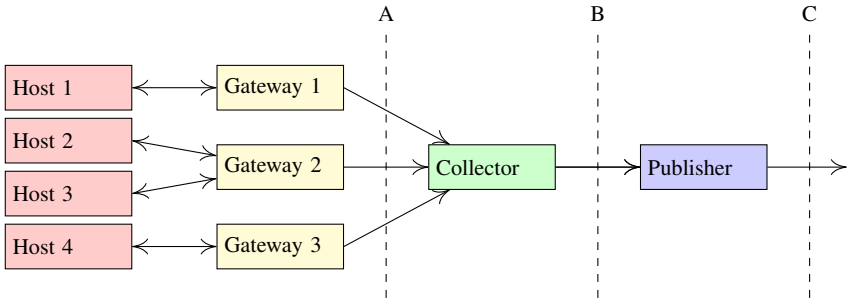

Figure 6.1: Physical Collector Publisher Testing Environment Overview

6.1.1 Environment

The testing of these subsystem was performed using 9 physical hosts that generated live network flows using IPv4. The configuration of these hosts and direction of network traffic can be seen in Figure 6.1. Note that this figure has points marked off as A, B and C. These points denote where monitoring occurred during testing, and will be referred back to later in this section. The list below helps better describe each component of this environment:

Host #: These are physical hosts set up to access the Internet through their connected Gateway. Network access is performed through automated bots that simulate Internet browsing.

Gateway #: The Gateways' role in this environment is two fold. In this environment a Gateway gives Hosts access to the Internet for simulated browsing but they also run Softflowd (Miller, 2016) in order to log network flows and export these records to the Collector.

Collector: This is the Bolvedere subsystem that collects NetFlow records, discerns and then re-orders them into the format requested by the Publisher in order for the Publisher to optimally process them.

Publisher: This is the Bolvedere subsystem that is in charge of all CPU-based processing before distribution to processor modules. This includes application of filters and interfacing to ZMQ.

The 4 host systems in this environment are tasked with generating network flows through their gateways. These gateways generate NetFlow records according to the NetFlow version 9 protocol and forward these records to the collector. The collector discerns these records, then re-orders them into the format required by the publisher before finally passing them onto the publisher via a socket. The publisher can now optimally apply filters to these re-ordered NetFlow records before distributing them to the processor modules.

Listing 6.1: Collector Configuration for Functional Testing

```
1 IPV4_SRC_ADDR
2 IPV4_DST_ADDR
3 L4_SRC_PORT
4 L4_DST_PORT
5 IN_PKTS
6 IN_BYTES
7 FLOW_SOURCE_ID
```

6.1.2 Collector Testing

Testing of the collector was performed by configuration of the collector to a known re-ordering and monitoring of input packets to the collector at point A, and output packets at point B (refer to Figure 6.1). The configuration script in Listing 6.1 denotes which fields from a NetFlow data record, as well as the order in which the publisher wants these fields. Missing fields are noted by setting every bit in the re-ordered NetFlow data record's field to 1.

The names in Listing 6.1 are defined by the user in the template file that is used during build time as discussed in Section 4.3. This template file is used by the publisher to build its code base from, and will be discussed in more detail in Section 6.1.3. The names in this configuration file are also used to identify the types and lengths of fields, based on what is also stated in the template file for each name. One can refer to Appendix F for a full listing of the template file used in this section.

Using the configuration produced by Listing 6.1 the collector is expected to behave in the following manner. In the case that the relevant template record to discern a data record set is not present, the data record set will simply be ignored. If the template record required to discern the date record set is present, then where the required fields for re-ordering are present, those fields will be placed in the correct order defined by the given configuration file, and where the fields are not present in the data record, all the bits in the bytes allocated for that data will be set to 1.

Testing will occur over a 1,000 exported NetFlow packets each containing between 1 and 30 records of varying type (either template or data). The count as to how many template records and data records will be collected at runtime along with the bytes they contain, and these will be used to determine the correct results which the collector should produce. Lastly one must note the format of the NetFlow data records exported by Softflowd to the Bolvedere collector (only the IPv4 template is considered here due to the network being IPv4-based). The fields and their order are shown in listing 6.2. One can note that all fields required for re-ordering exist in the exported data records by Softflowd. Testing for missing data fields will take place in Section 6.1.4 once basic functionality is shown to work here.

Listing 6.2: IPv4 Softflowd NetFlow Template Record Format

```
1  IP Source Address
2  IP Destination Address
3  Last Switched
4  First Switched
5  Total Bytes Transferred
6  Total Packets Transferred
7  Input SNMP
8  Output SNMP
9  Layer 4 Source Port
10 Layer 4 Destination Port
11 Protocol
12 TCP Flags
13 IP Protocol Version
```

6.1.3 Publisher Testing

The packets transmitted from the collector are monitored at point B and received by the publisher for filtering, and again transmitted via ZMQ and monitored at point C (refer to Figure 6.1). The configuration script used to build this Bolvedere publisher's internal filters is the same as the one used in the collector's testing. This can be referred to again in Listing 6.1.

As discussed in Section 4.3, the template file provided at build time contains the code to be built into the publisher in order to filter and pass on NetFlow logs to processor modules. Referring again to Appendix F one can note that the code is written in C, this was also explained in Section 4.3. The actions performed to received logs from the collector, and what is determined to be pushed to the ZMQ buffer for transmission, is determined here.

In the case of this simple functionality test the filter simply accepts all re-ordered data records and pushes them into the ZMQ buffer for transmission. It is also worth noting that what is pushed into the buffer is also printed to terminal by these simple filters. At runtime of this testing this printed output can be redirected to file for later analysis, and when coupled with the monitored network traffic produced by the ZMQ handle, more accurate results can be obtained.

Testing will again occur over 1,000 exported NetFlow packets in which each can contain between 1 and 30 data records. Again, the count as to how many data records will be collected at runtime along with the bytes they contain, and these will be used to determine the correct results which the publisher should produce.

One must take note that the publisher will never see a template record, as this is used solely by the collector to discern data records. The publisher already knows what is arriving at its ingress interface due to the configuration script at build time.

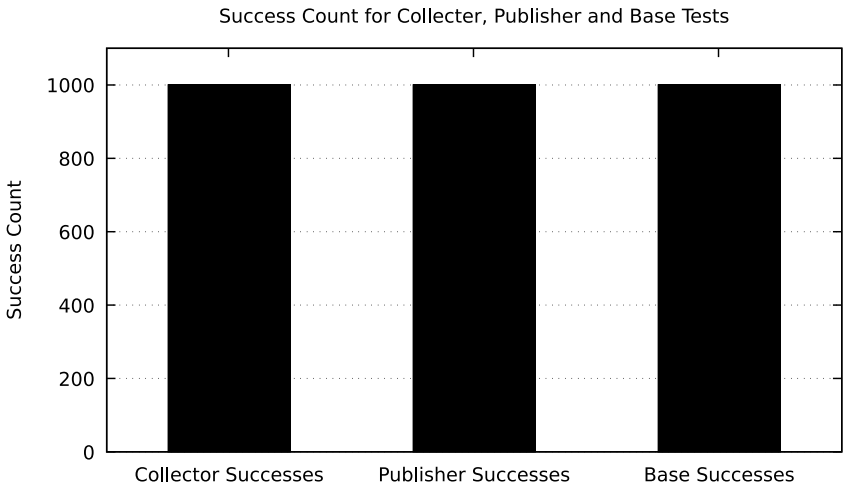

Figure 6.2: Count of Successfully Processed NetFlow Records

6.1.4 Base Subsystem Testing

This test monitors NetFlow records at points A and C when referring to Figure 6.1. Although this test is not required, as if collector specific testing and publisher specific tests are successful, then this should be too, the testing of the entirety of the base subsystem for completeness is still justified. An extension to testing performed in Sections 6.1.2 and 6.1.3 will see the requirement of the publisher of a NetFlow field not supplied by Softflowd.

For this testing to be successful NetFlow data records received by the collector at point A should be transmitted via ZMQ at point C, with the missing field's bits all set to 1 (referring to Figure 6.1). If this test and the tests in Sections 6.1.2 and 6.1.3 are successful, that is not corrupting or dropping any data, Bolvedere processor modules will be unhampered by this base subsystem and thus error in processor module results will be due solely to the processor module.

This testing will again occur over 1,000 export NetFlow packets in which each can contain between 1 and 30 template or data records. These again will be counted as to how many data records were collected at runtime along with the bytes present in each packet, in order to determine the correct output of the base subsystem for use in denoting the success of the base system.

6.1.5 Collector Subsystem Results

Referring to Figure 6.2, of the 1,000 NetFlow records that were transmitted by the gateways to the collector, the collector successfully received all of them. Furthermore, data records were not

Listing 6.3: Input NetFlow Data Record to Collector

```
1  // Header Information and Other NetFlow Records
2  ...
3  D4 48 64 C9    // Source IP          : 212.72.100.201
4  68 10 41 CB    // Destination IP     : 104.16.65.203
5  00 00 13 8E    // Last Switched      : 5006
6  00 00 05 7E    // First Switched     : 1406
7  00 00 30 C3    // Total Bytes        : 12483
8  00 00 00 70    // Total Packets      : 112
9  00 00          // Inbound SNMP       : 0
10 00 01          // Outbound SNMP      : 1
11 6D 85          // Source Port        : 28037
12 00 50          // Destination Port   : 80
13 06             // Protocol           : TCP
14 00             // TCP Flags          : None
15 04             // IP Protocol        : IPv4
16 ...
17 // More NetFlow Records
```

processed until the correct template record was received. From this point data records received had all relevant data extracted and re-ordered, and packets being transmitted and monitored at point B were correct. An example of bytes received by and transmitted from the collector can be viewed in Listings 6.3 and 6.4 respectively.

One can observe from these listings that all fields required by the publisher were available in the received NetFlow data record. Furthermore, the transmitted re-ordered data records were correct in contained data and order. One can note that the flow source ID is not part of the data record. This field was instead extracted directly from the NetFlow packet header's source ID field. Due to the success of these results, the collector was deemed fully functional and consistent.

6.1.6 Publisher Subsystem Results

One must note before analysing these results that 1,000 re-ordered data records were in question. This meant that more than 1,000 NetFlow records were received by the collector during as template records were also transmitted to the collector in order to discern the received data records.

Listing 6.4: Output Re-Ordered NetFlow Data Record from Collector

```
1  // Header Information and Other Re-Ordered Data Records
2  ...
3  D4 48 64 C9     // Source IP          : 212.72.100.201
4  68 10 41 CB     // Destination IP     : 104.16.65.203
5  6D 85           // Source Port        : 28037
6  00 50           // Destination Port   : 80
7  00 00 00 70     // Total Packets      : 112
8  00 00 30 C3     // Total Bytes        : 12483
9  00 00 00 64     // Flow Source ID     : 100
10 ...
11 // More Re-Ordered Data Records
```

Listing 6.5: Input Re-Ordered NetFlow Data Record to Publisher

```
1  // Header Information and Other Re-Ordered Data Records
2  ...
3  D4 48 64 C8     // Source IP          : 212.72.100.200
4  68 1C 13 0B     // Destination IP     : 104.28.19.11
5  34 26           // Source Port        : 13350
6  00 50           // Destination Port   : 80
7  00 00 00 2F     // Total Packets      : 47
8  00 00 22 49     // Total Bytes        : 8777
9  00 00 00 64     // Flow Source ID     : 100
10 ...
11 // More Re-Ordered Data Records
```

Listing 6.6: Output Filtered NetFlow Data Record from Publisher

```
1  // More Printouts
2  ...
3  [Data] IPv4 source address       : D44864C8
4  [Data] IPv4 destination address  : 681C130B
5  [Data] L4 source port            : 3426
6  [Data] L4 destination port       : 0050
7  [Data] packets communicated      : 0000002F
8  [Data] bytes communicated        : 00002249
9  [Data] Flow Source ID            : 00000064
10 ...
11 // More Printouts
```

6.1.7 Base Subsystem Results

The final round of testing for Bolvedere's base subsystem was intended to be targeted at the special case involving the event that a field does not exist in an exported NetFlow data record. This test does however act two-fold. The first was explained previously however the second is to again ensure that data passes through the collector publisher pair as expected, as the entirety of this system's implementation rests on these two subsystems working properly.

A note must be made when referring to Figure 6.1 that the 1,000 successful NetFlow records are data records that made it through the collector and out the publisher. Template records in this testing were used only by the collector and not seen at all by the publisher.

Listing 6.7: Modified Output of Softflowd Sent to Collector

```
// Header Information and Other NetFlow Records
...
D4 48 64 C9    // Source IP          : 212.72.100.201
CC 4F C5 C8    // Destination IP     : 204.79.197.200
00 00 94 36    // Last Switched      : 37942
00 00 86 26    // First Switched     : 34342
00 01 98 DC    // Total Bytes        : 104668
00 00          // Inbound SNMP       : 0
00 01          // Outbound SNMP      : 1
31 4C          // Source Port        : 12620
00 50          // Destination Port   : 80
06             // Protocol           : TCP
00             // TCP Flags          : None
04             // IP Protocol        : IPv4
...
// More NetFlow Records
```

Listing 6.8: Output of Modified Softflowd Input from Publisher

```
// More Printouts
...
[Data] IPv4 source address      : D44864C8
[Data] IPv4 destination address : 681C130B
[Data] L4 source port           : 3426
[Data] L4 destination port      : 0050
[Data] packets communicated     : FFFFFFFF
[Data] bytes communicated       : 00002249
[Data] Flow Source ID           : 00000064
...
// More Printouts
```

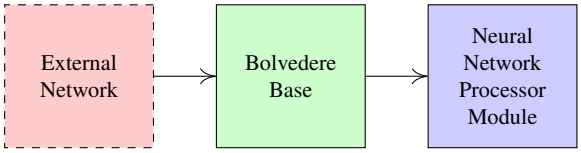

Figure 6.3: Neural Network Processor Module Environment

In this testing Softflowd was modified and recompiled from source to omit the packet count field when generating template and data records to a NetFlow sink. The bytes transmitted from this modified Softflowd can be referred to in Listing 6.7.

The output shown by Listing 6.8 is the output generated by the base system from the input of Listing 6.7. The results are as expected in this selected case and every other case during testing, and can be seen in this selected case by all bytes referring to the packet count (packets communicated) being set to 0xFF; this is the value of every bit in a byte set to 1.

With these results consistently found throughout all 1,000 data records this test was determined to be a success and thus the proper functionality of base subsystem is confirmed.

6.2 Neural Network Based DDoS Detection

This section evaluates the effectiveness of this neural network processor module, which attempts to detect targets of DDoS attacks (Section 5.1). There are two areas to consider when training a neural network to achieve a task. This is the time taken to train the neural network versus the effectiveness of the trained neural network, and the actual ability of the neural network to distinguish between legitimate network flows and those as part of a DDoS attack. Testing in Section 5.1 described both of these areas.

6.2.1 Environment

The environment used for this testing was the neural network processor module subscribed to the publisher of a Bolvedere base system, that published filtered data records containing the fields defined in 5.1. This can be seen in Figure 6.3. These data records were then scrutinised by the trained neural network in order to detect whether a network flow is part of a DDoS attack or not. To explain each system shown in Figure 6.3, one can refer to below:

Table 6.1: Time to Train Neural Network and Success Rate According to Size

Test	IPs Seen	Dest. Ports Seen	Packets	Ave. Bytes	Protocol	Result
1	1	1	100	100	17	0.992
2	1	1	100	1,000	17	1
3	1	1	10,000	1,000	17	0
4	1	1	3,000	1,000	17	0.002
5	1	1	2,750	100	17	0.371
6	1	1	2,750	1,000	17	0.996
7	1	1	2,750	100	6	0.999
8	1	1	2,750	1,000	6	1.001
9	1	5	2,750	100	17	0.442
10	5	1	2,750	100	17	0.371
11	5	5	2,750	100	17	0.442
12	5	5	5	120	17	1

External Network: This is the external network where NetFlow records are sourced from into the Bolvedere Base.

Bolvedere Base: This refers to the two subsystems tested previously, the collector and base, in Section 6.1.

Neural Network Processor Module: This is the subsystem this section is testing that will run the neural network that attempts to discern whether a network flow is part of a DDoS attack or not.

The training set used to train the neural network in these tests was synthetically generated from a multi-host DDoS attack. The style attack had characteristics similar to those found by Haris *et al.* (2010). This type of DDoS attack is formally referred to as a SYN Storm attack, and its characteristics are listed below:

- TCP-based communication with SYN flag raised.
- Short communications (no longer than a burst of 3,000 packets) from multiple IP addresses (typically more than 10).
- Packet payloads padded out to sizes over 500 bytes.
- Targeted at multiple commonly open TCP ports targeted.

This dataset was then converted into NetFlow records using Softflowd (Miller, 2016) and sorted by hand as to which data records were involved in the DDoS attack, and which were not. This

Table 6.2: Neural Network Classifiers

Classifier	Description
Lower packet count	Counts below 2,500 cause the resultant to increase even when the average packet size is low.
Higher average packet size	Found to increase the resultant from around 285 bytes and up.
Most common protocol seen used	TCP was noted to act as a deciding factor when edge cases were presented.
Number of unique ports	Has a minor effect on the result, but more unique ports cause an increase in the result.
Number of unique IPs	This had a negligible effect.

data is then given to the neural network for training, and with the known result of each network flow, it was trained. The training set contained 1,000 network flows known to be part of the DDoS attack, and 1,000 legitimate network flows known to be legitimate. The neural network was then trained according to the NetFlow fields defined in Section 5.1.1.

Table 6.1 presents the results of hand crafted inputs to the neural network after training has occurred. These inputs were designed to show the resultant characteristics of this neural network due to its training, and how its inputs interact with each other to produce a result.

These results presented in Table 6.1 show that there is a strong relation between number of packets seen and the average size of these packets, as to whether a DDoS attempt is detected or not. The common protocol[5] seen has a "final say" effect when the results produced are uncertain (this can be seen between tests 5 and 7). Lastly the number of IPs seen seem to have no effect on the result at all (tests 9 to 11 depict this behaviour). This information will be valuable when performing analysis on real-world datasets later in Chapter 7 and is summarized in Table 6.2 in order of most important to the neural network, to least important:

6.2.2 Effectiveness Test

It must be noted that this test was performed after training of the neural network. This test simply tests the ability of this neural network implementation to detect network flows involved in a simple SYN Storm style DDoS attack detailed in 6.2.1. The neural network consisted of 3 hidden layers in the configuration of 8 nodes, 24 nodes and 12 nodes. Input node count was automatically chosen to be 4 after training, as determined best by the R neural network training

[5]Protocol 6 is TCP and 17 is UDP as per RFC 791 (Postel, 1981a).

implementation. A single output node was created to give the confidence level that the data record representing the network flow was part of the DDoS attack, or not.

Results produced by this implementation are in the form of a confidence level between 0 and 1 that represents how sure the neural network is that the network flow is involved in a DDoS attack or not. Anything with a confidence level above 0.9 was considered as being part of a DDoS attack, and any result with a confidence level below 0.1 was considered not.

The data introduced to the neural network for learning and later detection is modelled around the simple SYN storm DDoS attack style. This form of attack in terms of NetFlow data records sees connections with low packet counts and packet sizes from multiple hosts. To add accuracy to the detection of these attacks, the data is preprocessed into packs of the last 5 data records received for a source IP. This data includes the number of unique source ports used, number of unique destination ports used, the source IP, average packet size and average number of packets.

6.2.3 Time to Train versus Effectiveness Test

This set of test iterations is used to determine the effectiveness of a neural network's complexity versus the time taken to train that neural network. This will allow for optimization according to one's needs at a later point, as well as the ability of the resources at hand at the time of training. An iteration of this test will occur in the following manner:

1. First a percentage of the number of nodes as that of the values used in Section 6.2.2 will be decided upon.
2. The training of the neural network will then be performed and timed using the same training set as in Section 6.2.2 a total of 10 times.
3. Lastly the use of data used to test the effectiveness of the generated neural networks will be the same in each case as, to equally evaluate the performance of each neural network. Effectiveness will be determined by the number of trained neural networks out of the 10 that can successfully detect at least 80% of the DDoS attack flows that were detected by the neural network trained in Section 6.2.2.

Configurations that are used in this testing can be referenced in Table 6.3.

Table 6.3: Size of Each of the DDoS Detection Neural Network's 3 Hidden Layers

Percentage Size of Default	1st Layer	2nd Layer	3rd Layer
12.5%	1	3	2
25%	2	6	3
50%	4	12	6
100%	8	24	12
150%	12	36	18
200%	16	48	24

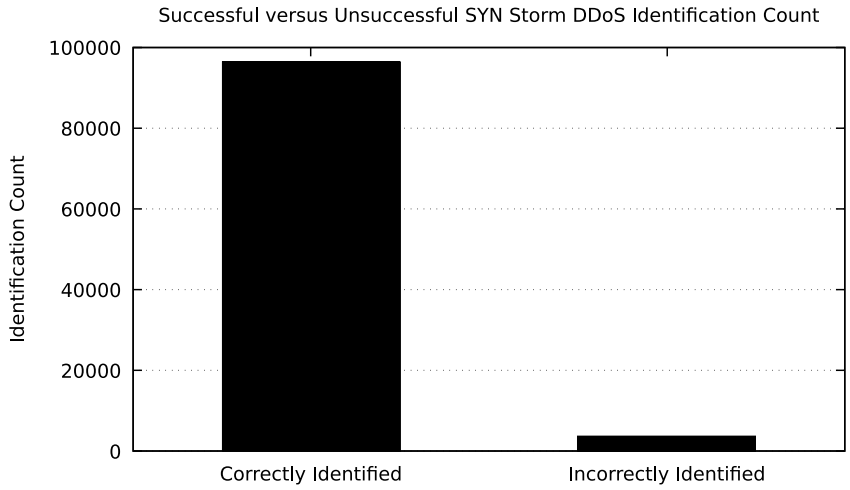

Figure 6.4: Neural Network Detection of Simple SYN Storm DDoS Attack Results

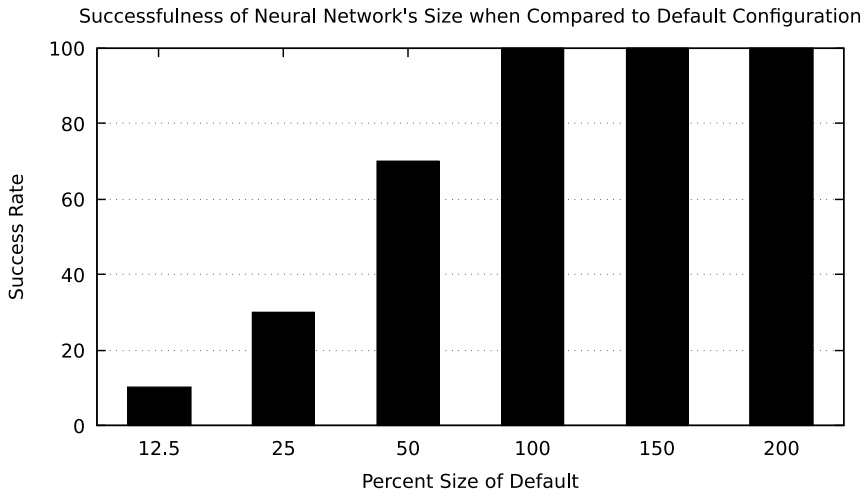

Figure 6.5: Neural Network Success Rate Compared to Default Configuration Size

6.2.4 Effectiveness Results

The results of 100,000 iterations of testing can be referred to in Figure 6.4. Of the 100,000 packed NetFlow data records that were given to the neural network, 96,479 were correctly identified as to be part of, or not part of, a SYN style DDoS attack, where 3,521 were incorrectly identified. This means that the neural network trained to detect the aforementioned anomaly in this testing was 96.479% accurate.

These results seem impressive but, due to the sheer number of methods to perform a DDoS attack, these results that are aimed at a specific type of attack are mild. This method of denial of service is also aimed only at saturation of the network link in which a server would receive requests, whereas there are many other mechanisms within a server that can be exploited to achieve the same effect.

This testing however does serve its purpose in the context of this research. The ability for a neural network to learn from data produced by Bolvedere and produce correct and meaningful results was the main point of this testing. With these results, it is proven that such an approach to more complex neural networks such as a long short term memory neural network (Sak *et al.*, 2014) or bidirectional recurrent neural network (Schuster and Paliwal, 1997) can be implemented and use Bolvedere as a data source.

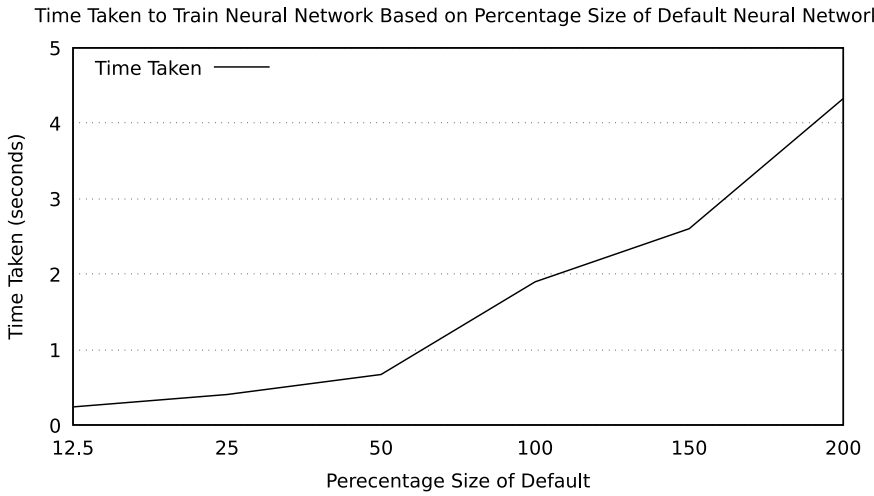

Figure 6.6: Neural Network Training Times Compared to Default Size

Table 6.4: Time to Train Neural Network and Success Rate According to Size

Percentage Size of Default	Time to Train (seconds)	Successful Neural Network
12.5%	0.243	10%
25%	0.407	30%
50%	0.672	70%
100%	1.900	100%
150%	2.603	100%
200%	4.327	100%

6.2.5 Time to Train versus Effectiveness Results

Each generated neural network's configuration, and thus size, in comparison to the default configuration used in Section 6.2.2 can be referred to in Table 6.3. Time taken to train each neural versus its relative effectiveness at DDoS detection when measured against the results in Section 6.2.4, are tabulated in Table 6.4 and graphed in Figure 6.5. Finally, the time taken to train each sized neural network used in DDoS detection can be referred to in Figure 6.6.

These results are averaged over 10 iterations of training with the same training set used for the default neural network in Section 6.2.2. After each neural network was trained, they were each subjected to the same 100,000 packed NetFlow data record testing as was used in the default neural network, and effectiveness was compared to that of the default neural network.

As is expected, the smaller the neural network's size, the faster it is to train, as there are less neurons within the neural network to adjust during backwards propagation. The effectiveness of the neural network as size varies does not hold any outliers either. Simply when there is not enough neurons in a neural network to effectively compute the task, accuracy suffers. In the same vein, a larger neural network will always be able to at least perform the functionality of one smaller than it of similar design (referring to number of input neurons, number of output neurons and number of hidden layers).

The time taken to train each of these neural networks as they grow in complexity is fairly linear. Referring to Figure 6.6 one can see from 50% size through to 200% follows a linear trend. This is due to the nature of the backwards propagation used in these neural networks. Each neuron is visited individually and adjusted according to the correctness of the outputs. In the simplest case, which is used in these neural networks, adding a neuron to the neural network simply means that that neuron has to also be looked at; in this case that adjustment is disjoint from other adjustments. This serves the bases for the linear growth in time taken to train a neural network versus the complexity of the neural network itself.

The problem given to these neural networks, as stated in Section 6.2.4, is fairly simple in terms of what can be expected in real-world application. In contrast to an all-in-one solution to every DDoS attack currently in existence, and that will possibly exist, taking just shy of 2 seconds to solve the task of a SYN style DDoS is negligible. However, the time taken varies on the size of the neural network, the learning algorithm used, the neural network algorithm used and the size of the training set. Some training algorithms used to train neural networks can grow exponentially in time required as more neurons are added. Furthermore, adding the detection of just one or more type of DDoS attack will require the training set for that type of DDoS. This

doesn't even highlight the need to gather the data for the training set and work out how the use of each training set together affects the ability and accuracy of the neural network's to detect each form of attack.

6.3 Fourier Analysis

The Fourier analysis processor modules in this implementation transform time dependent events into frequency dependent ones in order to see if there are any periodic events happening on a network. Packets received are placed in time frames, which were called buckets, as to either how many bytes or packets were received at a given time bucket; each were dealt with disjointly. Furthermore, it must be noted before further reading of this section that Fourier analysis is deeply time based. NetFlow's method of generating data records severely distorts the time domain in which the data records arrive. For this reason Fourier analysis on both NetFlow data records, and the actual network traffic the network flows were generated from, is performed to analyse the effectiveness of NetFlow based Fourier transforms.

6.3.1 Environment

The supporting hardware environment used during the execution of all GPU-based testing in this section is as follows:

CPU: i5-3570@3.4Ghz

Memory: 32GB DDR3 1600Mhz

Hard Disk: 1TB Western Digital Blue

Network Interface: Intel 82574L Gigabit Network Connection

Implementation of this processor can be set to either real-time Bolvedere processor module or standalone bulk processor modes. In the former this processor connects to a Bolvedere publisher and manages calculation times and bucketing of data in real-time. The latter sees this system taking in an input file containing recorded network flows and performing the required calculations on this data.

As this processor is intended to detect periodic data within noise, it was decided that the standalone mode should be used in generating these results (this is fine as the calculation remains

unchanged). This is due to the noise being random in nature where testing requires consistency in order to achieve accuracy in results. This allows for generation of a set dataset with known noise to the tester, and thus expected results can be formulated.

The test set created for these tests consists of the program Ping (Braden, 1989) generating periodic traffic between two hosts at a frequency of 1 hz on a live network where multiple other hosts are communicating with each other. These other communications are random and thus generate noise in our data. This data recorded as both a packet capture file stored as a pcap and as NetFlow data records.

6.3.2 Packet Based versus NetFlow Data Record Based FFT Effectiveness

This test was aimed at detailing the effectiveness of Fourier analysis on NetFlow data records when compared to a traditional packet capture technique. Traditionally, packet capture would occur and data received would be bucketed by how many bytes were received in a given time slice. These would then become discrete time dependant points of how many bytes were received at a given time. After this conversion a FFT can be performed on the data to give a resultant frequency graph showing periodic behaviour within the packet capture.

For NetFlow data records these tests followed a similar approach. As data records are received by the NetFlow collector, the amount of bytes that were transferred in the flow is bucketed on the time the data record was received. This data is then, like the packet capture method, also represented as discrete time dependant points of how many bytes were received at a given time. This allows one to perform the same FFT that one would perform on the packet capture data, on these NetFlow data records.

The logical flow for this test was as follows:

1. Start Ping program on live network with known IPs.
2. Start packet capture.
3. End Packet capture and Ping program.
4. Filter data into 3 sets namely: Noise only, Ping only, and Noise and Ping.
5. Use Softflowd to generate NetFlow records on the packet capture.
6. Using the standalone bulk processing mode of the FFT processor module feed in both the packet capture data and NetFlow data record based data.

7. Gather resultant graphs.

Once both results are collected, comparisons will be drawn as to the effectiveness of NetFlow based Fourier Transforms to that of traditional packet capture based Fourier Transforms.

6.3.3 Packet Count versus Byte Count Analysis Effectiveness

Use of different dependant data that represents like data is important when performing a FFT, as periodic data is subject to loss due to comparatively high levels of noise. This is due to the nature in which an FFT operates. An FFT defines the dependant result as a value of how much of a given frequency is detected. It does this through summing the amplitudes of waves common to a frequency (this includes harmonics). These frequencies are then compared to other detected frequencies and the ones that present prominent maxima in the results represent the dominant frequencies in the input data. If the amplitudes of the periodic data are so small that when a FFT is performed on the entire data set it causes the summed total of that frequency to be that of the level of the noise, one will not be able to gain any meaningful information from the resultant FFT.

In the generation of data for the above byte based tests, as explained in Section 6.3.2, the liberty was taken to also generate packet count based data too. This could be done, as in both the packet capture and NetFlow data records generated, this data could be collected. In the packet capture this was done by simply counting the number of packets that were seen during a time slice, and in the NetFlow records the data was collected from the field containing the packet count of a network flow. The same FFT as performed in Section 6.3.2 was performed on this data, and comparisons were then drawn between these results.

6.3.4 Packet Based versus NetFlow Based FFT Effectiveness Results

The first FFTs performed were on that of the traditional packet capture of just the ping packets received by the capturing node. Figures 6.7 and 6.8 show the results of the FFTs on this data at two different bucketing levels, this being at a resolution of 10^{-2} and 10^{-1} respectively. These results are as expected for ping, as the Ping program transmits a ping packet every 1 second and this data shows a strong presence of the 1 hz frequency, as well as every harmonic frequency 1 hz is part of.

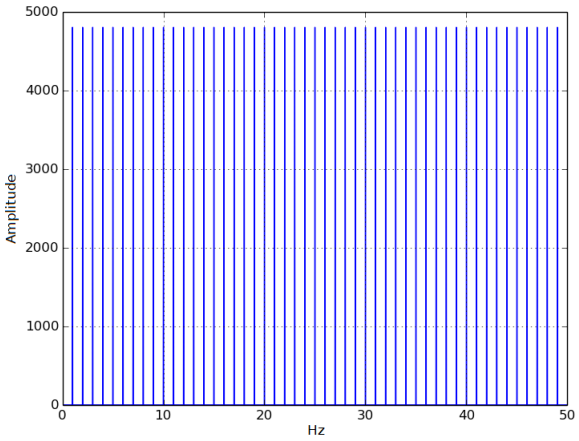

Figure 6.7: FFT of Ping Packet Capture Bucketed by Bytes Received at 100 Hz

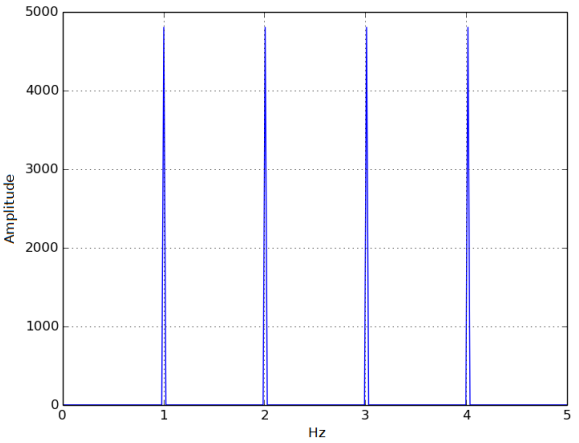

Figure 6.8: FFT of Ping Packet Capture Bucketed by Bytes Received at 10 Hz

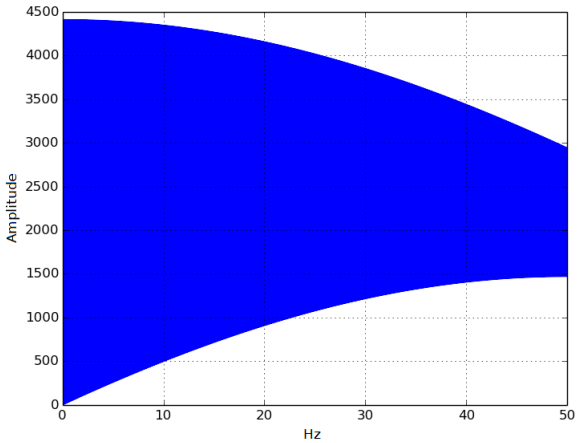

Figure 6.9: FFT of NetFlow Ping Data Records Bucketed by Bytes Received at 100 Hz

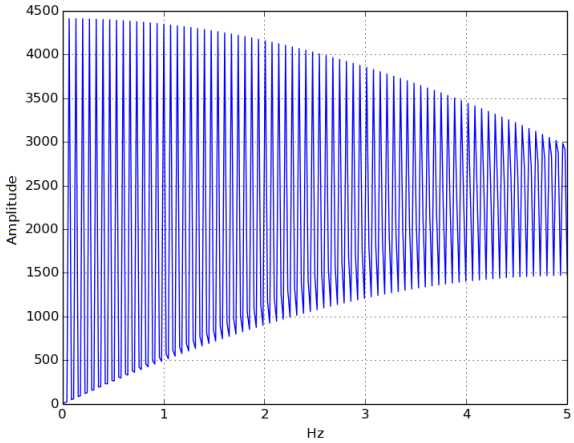

Figure 6.10: FFT of NetFlow Ping Data Records Bucketed by Bytes Received at 10 Hz

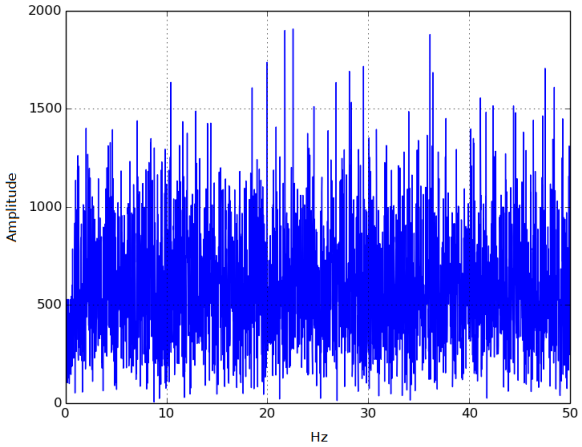

Figure 6.11: FFT Captured Noise Bucketed by Bytes Received at 100 Hz

Figures 6.9 and 6.10 represent the second FFTs performed on that of the NetFlow data records of just the ping packets. The results produced by these graphs show no signs of periodic information, especially as it is known that the maximum searched for here is at the 1 hz mark, which it is not in these results. This shortcoming is due to the nature of FFTs mentioned in the opening paragraph of Section 6.3. Granulating the sample data into less time samples reduces the effectiveness of the FFT, which is heavily reliant on the time domain. The negative effect of this is clear when comparing Figures 6.7 and 6.8 to Figures 6.9 and 6.10. Further result from the NetFlow data yield no interpretable results, however for completeness the packet capture data will be used to show the effectiveness of Fourier analysis on network traffic.

The FFT of just the noise in the packet capture is given at a resolution of 10^{-2} in Figure 6.11, and no zoom was given as the data has no special qualities about it; it is just noise on a live

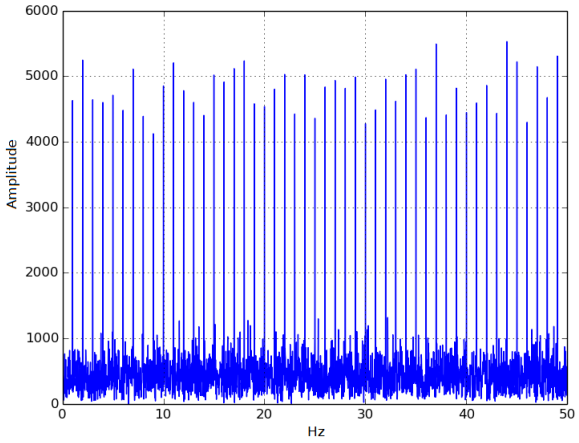

Figure 6.12: FFT Ping and Noise Bucketed by Bytes Received at 100 Hz

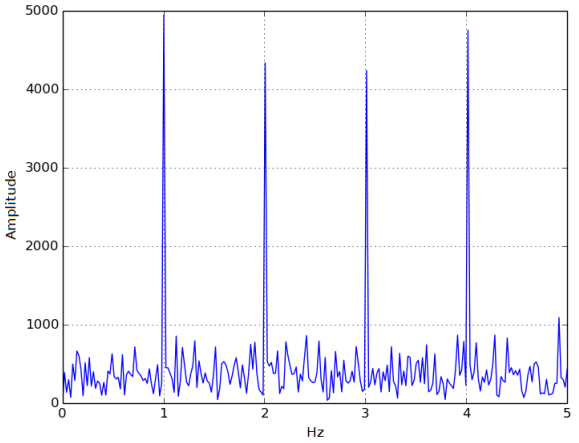

Figure 6.13: FFT Ping and Noise Bucketed by Bytes Received at 10 Hz

network.

Lastly, Figures 6.12 and 6.13 are FFTs performed on the entire packet capture, containing both the noise and ping. This is where the strength of Fourier analysis is shown, as one can clearly pick out that there are dominant frequencies within the data. This of course are the ping packets being sent by the Ping application, and this fact is backed up by the dominant frequencies of these graphs, being 1 hz and all harmonics of it, as was shown by Figures 6.7 and 6.8. This is all information that is lost upon the granulation of the time domain as NetFlow does.

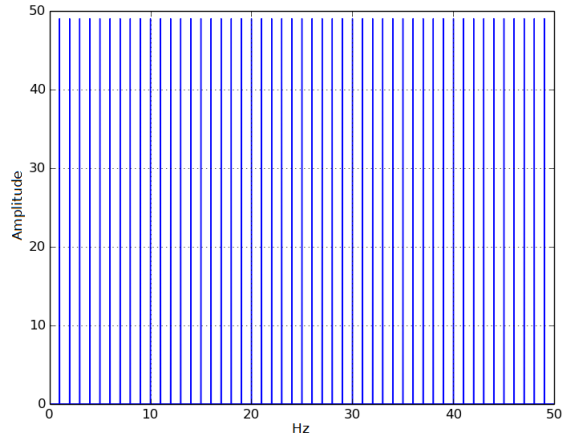

Figure 6.14: FFT of Ping Packet Capture Bucketed by Packet Count Received at 100 Hz

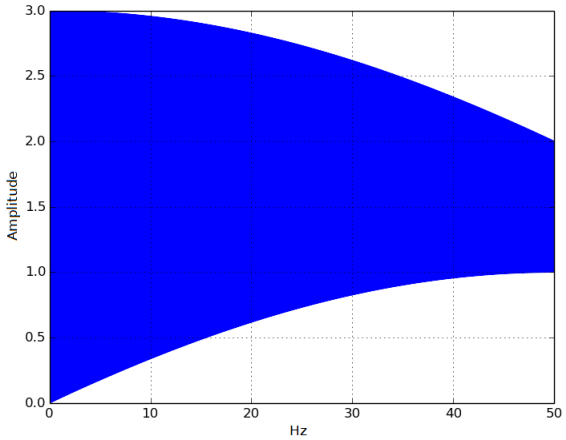

Figure 6.15: FFT of NetFlow Ping Data Records Bucketed by Packet Count at 100 Hz

6.3.5 Packet Count versus Byte Count Analysis Effectiveness Results

The results depicted by Figure 6.14 show the FFT of the packet capture in Section 6.3.4 bucket on packet count rather than byte count. Again, one can observe the existence of a dominant frequency within the data set. This is again 1 hz and all harmonics of 1 hz, as expected by the transmission of a ping packet every second by Ping.

Again, the results acquired by performing an FFT using NetFlow data records as input data to the function reveals little meaning, this can be seen in Figures 6.15 and 6.16. Performing the FFT with packet count as the the dependant variable shows a similar trend to that of data based on byte count, and further data collected through packet count based data yields no meaningful data.

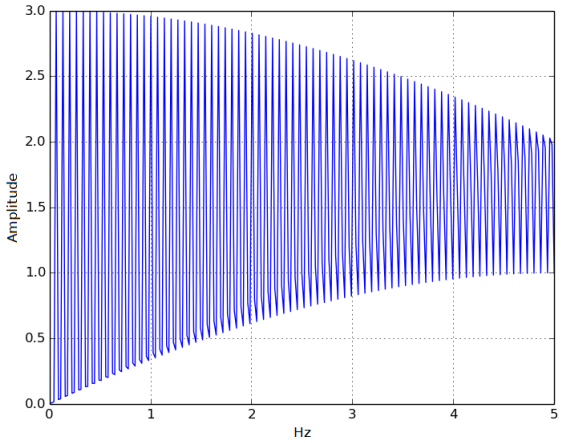

Figure 6.16: FFT of NetFlow Ping Data Records Bucketed by Packet Count at 10 Hz

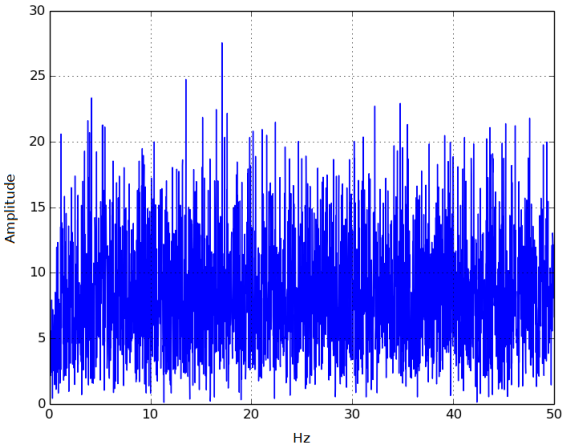

Figure 6.17: FFT Captured Noise Bucketed by Packet Count Received at 100 Hz

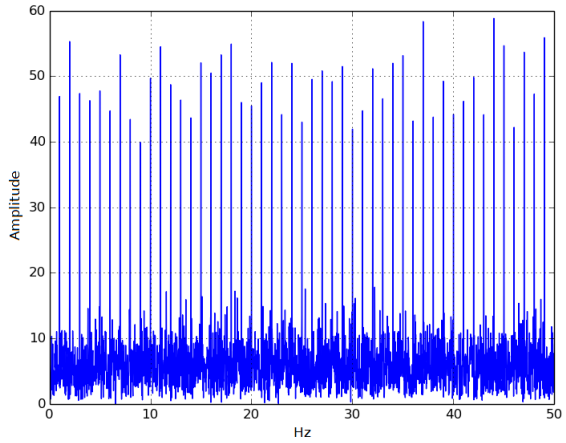

Figure 6.18: FFT Ping and Noise Bucketed by Packet Count Received at 100 Hz

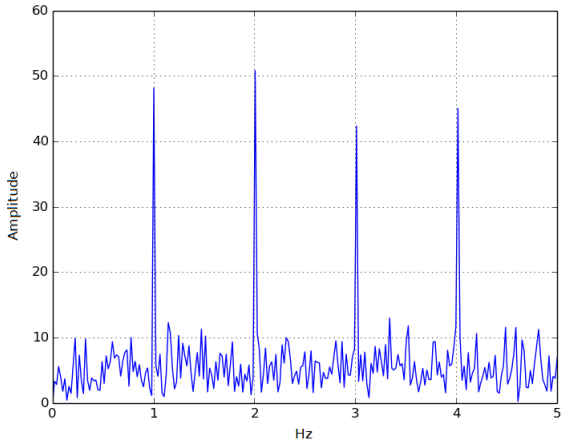

Figure 6.19: FFT Ping and Noise Bucketed by Packet Count Received at 10 Hz

The FFT of the packet count bucketed data generated by the noise in the packet capture is shown in Figure 6.17. Again, this shows no meaningful information as it is just background noise generated on the live network this packet capture was collected on

Finally, the results of packet count based input data can be compared directly to that of its byte based counterpart in Figures 6.12 and 6.13. The results of these Figures 6.12 and 6.13 show the exact same dominant frequencies as in the results of the byte based input data in Figures 6.12 and 6.13. This initially suggests that use of packet count or byte count can be used interchangeably when performing frequency analysis. However, this is not true.

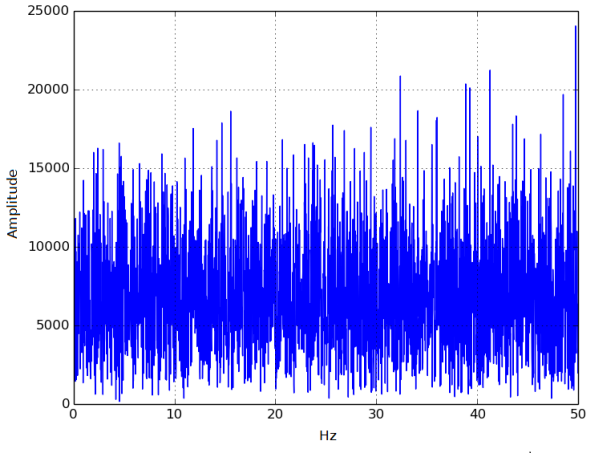

Figure 6.20: FFT Ping and Large Noise Bucketed by Bytes Received at 100 Hz

To elaborate this point, consider Figure 6.20. This graph, although fairly meaningless, uses the exact same data as in Figures 6.12 and 6.13 except that the byte count of each packet that is identified as noise has been increased to be 20 times larger than before. This increase makes the noise results large enough to completely dwarf the periodic ping packets, thus resulting in a result devoid of the dominant periodic frequency that should be in them.

In this case it is possible to use the packet based input to get the frequency graph required to get this periodic information out of the packet capture. This is possible, as only the size has been changed in the noise packets in the packet capture, not the number of packets. In the case that the periodic data in this packet capture increased in byte count, even if more noise packets were introduced into the data, one would see the tables turn in the results.

A byte based FFT in this case would show the dominant frequencies clearly, as the periodic data would have a far greater byte count than that of the noise. On the other hand, using packet count would show the periodic data becoming less distinguishable, as the number of packets related to the packet count would be drowned out by the increasing count of noise packets.

For this reason the final implementation of this processor module buckets both packet count and byte count separately, and renders them in parallel in order to mitigate the short falls in observing only one of the two fields.

6.3.6 CUDA Performance on Varying GPUs

Table 6.5 shows the performance speeds of different nVidia CUDA compliant products when computing the same FFT work load. A chunk in this table represents 2,500,000 million discrete points, and the workload can be calculated by referring to the chunk count in the given table.

Table 6.5: CUDA GPGPU Time Taken to Compute FFT in Seconds

nVidia Product Name	4 Chunk Load	64 Chunk Load	512 Chunk Load
GTX 1080ti	0.444	4.521	39.834
GTX 750ti	0.576	6.104	58.737
GTX 480	0.561	5.864	57.001
GTX 465	0.591	6.158	59.438

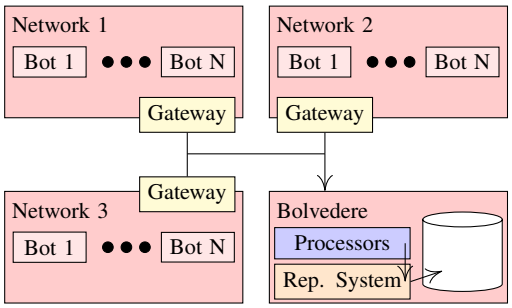

Figure 6.21: Testing Environment Overview

6.4 Reputation Analysis System

This subsystem sought to collect findings produced by processor modules running as part of the Bolvedere system and couple these findings with real-world geographic locations based on the NetFlow node ID in which the network flows were generated from. As these findings are intended to be stored for use at a later point, this testing will be broken into two parts. The first is generation of malicious network activity in which the Bolvedere modules generate findings for the reputation system to store, and the second being the accuracy of these results that are stored and how they can be applied.

6.4.1 Environment

These tests were all performed in a virtual environment which allowed the creation of disjoint networks that could then be connected through common gateways. This was to represent an Internet like structure of a network with connections between Internet service providers and/or telecommunication companies. The configuration of this can be referred to in Figure 6.21. To better explain objects and logic flow in this figure one can refer to the descriptions below:

Bot: This refers to one of two automated systems. The one form of system generates legitimate network traffic through HTTP (Fielding *et al.*, 2009), ICMP (Postel, 1981b), TCP (Postel,

1977) and UDP (Postel, 1980) requests. The second form generates malicious network activity through the use of Metasploit (Maynor, 2011).

Gateway: These are virtual systems that allow access to other virtual networks within the virtual environment. These gateways also generate NetFlow logs using Softflowd (Miller, 2016) and thus act as NetFlow source nodes. Each of these Softflowd processes running on the separate gateways had been recompiled to use a unique source ID.

Bolvedere: This is the NetFlow sink of the virtual environment. All gateways send their NetFlow logs to the Bolvedere system which then processes these logs to detect malicious activity before handing these findings to the reputation system.

Network #: These are reference names of each virtual network in which the bots reside. These names exist purely for ease of referral in the results text.

6.4.2 Runtime Malicious Activity Collection

The purpose of the initial test is to gather malicious network flows from different networks and then determine which network the activity originated from using the NetFlow source ID. To achieve this, each of the three virtual networks was assigned certain malicious tasks in which it would perform, and how often it should perform each. This would allow for checking accuracy at a later point as to the reputation score of each network, according to the attacks performed (i.e., how likely a network was to perform a certain form of malicious network flow). A distribution of the malicious network tasks performed by each network and NetFlow source ID number can be referred to in Table 6.6 (time spent performing each task is out of a total of 100).

The results displayed by the reputations system were printed to terminal at the runtime of this system, and can be referred to in Listing 6.9. The format of the results produced for these tests displays three key pieces of information, however there is more information stored in the reputation system's database which was discussed in Section 5.5.1. The fields shown in this tests output are described below:

Source ID: NetFlow source node ID number in which the malicious flow record was sourced from.

Location: Network name that the NetFlow source node ID is recorded as.

Attack Type: The attack type that was detected in the malicious flow by a Bolvedere processor module. Results were collected over 1,503 malicious network flows and 11,312

Listing 6.9: Terminal Output of Reputation System

```
...
[Source ID: 100, Location: Network 1, Attack Type: ms08_067_shell]
[Source ID: 100, Location: Network 1, Attack Type: ms08_067_vnc]
[Source ID: 100, Location: Network 1, Attack Type: ms08_067_shell]
[Source ID: 300, Location: Network 3, Attack Type: samba_symlink]
[Source ID: 200, Location: Network 2, Attack Type: unreal_ircd_3281]
[Source ID: 300, Location: Network 3, Attack Type: samba_symlink]
[Source ID: 200, Location: Network 2, Attack Type: ntp_mon_list]
...
```

Table 6.6: Virtual Network IDs and Malicious Activity by Percentage

Name	ID	ms08_067_vnc	ms08_067_shell	samba_symlink	unreal_ircd_3281	ntp_mon_list
Network 1	100	40	40	10	0	0
Network 2	200	10	10	0	30	50
Network 3	300	0	0	100	0	0

non-malicious network flows. These results, again referring to the snippet in Listing 6.9[6], show the attacks that were assigned to the malicious bots are identified by the Bolvedere processor modules. Furthermore, the reputation system put in place to bind these findings to a set location also shows correct output for the test inputs provided to the system as a whole; this being the Bolvedere system and the reputation system.

6.4.3 Reputation Storage Results

The records of the 1,503 malicious network flows that were recorded by the reputation system in Section 6.4.2 will now be verified through processing of these records in this section. As the reputation of each network for all known attack findings given to this reputation system are generated at runtime, this section will simply look at these reputation scores and weigh them up against the known input used in testing. The higher the score the more likely the network is to source the respective malicious attack.

The first positive that the results in Table 6.7 reflect is that the networks assigned not to perform a form of malicious attack in Table 6.6 do not score any points for those attacks. The next point to notice is that the *unreal_ircd_3281* and *ntp_mon_list* attacks that were only assigned to the Network 2 networks, score 100 for each of these malicious attacks in the Network 2 reputation. This is good, as the Network 2 network is the only network that produces these attacks, and so

[6]The size of the snippet is due to the sheer size of the output.

Table 6.7: Virtual Network Malicious Reputation Scores out of 100

Name	ID	ms08_067_vnc	ms08_067_shell	samba_symlink	unreal_ircd_3281	ntp_mon_list
Network 1	100	82.7	79.9	15.8	0	0
Network 2	200	17.3	20.1	0	100	100
Network 3	300	0	0	84.2	0	0

if these attacks occur, Network 2 is the only network that these attacks have been sourced from, and so is the most likely candidate; hence the 100 point rating towards this network.

Finally, the attacks that were assigned to multiple networks show a distribution of reputation points between the assigned network in accordance to the ratio assigned to each network. The error in points is due to the randomness introduced in the malicious bots when choosing which attack to perform. These results show that the reputation system does correctly store and assign a reputation based on type of attack to a network location. Furthermore, these results can be used in order to better point out which network a malicious attack is most likely to be sourced from, and place that network flow at a known physical location irrespective of the source IP address.

6.5 Port Scan Detection

The port scan detection processor module is tested in this section. The testing performed in this section is on data known to contain port scans and data known to not have port scans. This processor module is simply aimed at being able to perform the equivalent of that of existing full packet analysis systems, this being detection of known common port scan techniques targeted at a host, or set of hosts, on a subnet. It must be noted that this processor module only executes in real-time, and so all results collected are as the NetFlow records are sunk into and distributed out of Bolvedere's base subsystem.

6.5.1 Environment

This testing was performed on a physical network containing live hosts behind a gateway host. The gateway host ran an instance of Softflowd for the entirety of the the port scan traffic generation, and thus created the NetFlow records containing the logs of that of the port scans. The port scans generated are known to the tester, and thus success or failure in the results generated by this processor module are known.

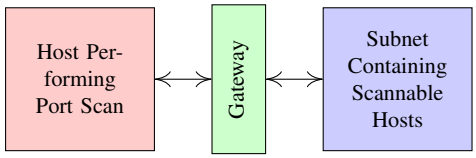

Figure 6.22: Port Scan Detection Environment

Referring to Figure 6.22 one can take note of the basic hardware set up of these tests. In these tests the port scanning technique used is a direct method on the subnet containing the scannable hosts. The port scanning host generates standard data through browsing of the Internet through a web browser and other standard network tasks such as acquiring an IP over Dynamic Host Configuration Protocol (DHCP), as well as performing port scans against single hosts and the entire subnet space allocated to the scannable hosts. The NetFlow records are generated live by the gateway running Softflowd, and thus results are generated as a direct result from interaction by the port scanning host on the network.

Port scans performed by the port scanning host in this testing is performed using Nmap (Lyon, 2015).

6.5.2 Port Scan Detection

For this testing 5 different modes were used to perform port scans when using Nmap, these are listed below:

sS: SYN Stealth Scan performs its scan by sending TCP SYN packets at the target host/hosts, the response from the host is then used to classify the openness of the targeted port.

sT: TCP connect() Scan performs its scan by using the underlying operating system's socket library's connect() method to generate SYN packets at the target host/hosts, the response is then used to classify the openness of the targeted port.

sA: ACK Scan performs its scan by sending TCP ACK packets at the target host/hosts, the response or lack of response to this packet is used to classify the openness of the target port.

sW: Window Scan performs its scan in the same manner as the ACK Scan, however it also checks for a feature in some operating systems in which open ports can respond with a RST packet. The detection of this feature is done by checking the window field, if the value is positive it is an open port, and if it is zero it is closed.

Table 6.8: Detection Rate of Each Port Scan Mode

Scan Mode	Target	Detection Rate (%)
sS	Gateway	91
	Subnet	88
	Slow Scan	86
sT	Gateway	97
	Subnet	92
sA	Gateway	91
	Subnet	93
sW	Gateway	92
	Subnet	89
sM	Gateway	87
	Subnet	85
False Positive	Gateway	7
	Subnet	11

sM: Maimon Scan performs its scan through exploitation of a feature in BSD operating systems, in that a closed port responds to a FIN/ACK packet with a RST packet, however it ignores the FIN/ACK packet if the port is open. This means that if no response is received to a FIN/ACK packet, and the host is running BSD, the port is open.

Each of these modes are used to attack two separate targets, these being the gateway itself and the entirety of the /24 subnet that is behind the gateway. There is an exception to this regarding an 11th scan performed using the sS mode where probes are spaced 2 minutes apart. This is to test the viability of dealing with port scans which occur over extended periods of time to better hide their signatures.

The detection rate for each of the port scan modes, along with the detection rate of false positives, will be recorded and taken into account when discussing the results of this testing. A conclusion as to whether this processor module works in these simple cases will be drawn from this. These tests are all performed in real-time.

6.5.3 Port Scan Detection Results

Table 6.8 provides a summary of successful detection rates, and also the false positive rate of this processor module. These results show a definite need for fine tuning of the processor module. However, they are promising on the outset. The false positive rate alone in these results

Table 6.9: Detection Rate of Each Port Scan Mode

Time (minutes)	Detection Rate (%)
2	86
5	82
10	74

does mean that a third-party needs to double check results produced by this module, as in the worst case currently 1 in every 9 legitimate connections are regarded as a port scan.

The inaccuracy of this module can be accounted to the granularity of the NetFlow protocol in that finer details are lost. This loss of finer detail leads to some NetFlow data records looking close enough to others that they can be mistaken for each other when performing a task such as the port scan detection in this testing.

Lastly, one should note the slow scan results produced in this testing. A strength of this module is to detect port scans with probes spaced minutes apart. The slow scan results in Table 6.8 was performed with probes 2 minutes apart. Table 6.9 can be referred to to view the detection rate for higher times between probes. The tests in Table 6.9 were performed only once, as they are for demonstration purposes only and take up to a week to perform.

With this taken into consideration, after fine tuning takes place for this module it will be considered a healthy addition to the processor modules provided with Bolvedere. Reduction of false positives to sub 5%, while increasing accuracy of every other case to over 95%, will see this processor module work as intended with minimal overhead for result validation.

6.6 Sudden Port Bandwidth Use Change

This section will show the test results for the Sudden Port Traffic Increase/Decrease Detection module. To reiterate the functionality of this module defined in Section 5.4, this module uses link bandwidth use, and applies 3 standard deviations of the mean of a windowed dataset, in order to detect sudden increases and decreases of bandwidth use on a port-IP pair in terms of bytes. If a new data point introduced to the module falls within 3 standard deviations of the mean, the data point is seen as normal and there is no cause for alarm. If a new data point is identified as an outlier to the existing dataset, the user is then notified of the occurrence.

Testing of this module was performed by simply streaming into it the source IP, destination IP, source port, destination port and protocol from the Bolvedere base system. Each data point fed

to this module is precalculated by the data point source as to fall within 3 standard deviations of the mean recorded by the module, or outside of it. This allows a test iteration to determine the accuracy of the results produced by this module by knowing whether a data point is an outlier or not, before it is introduced to this module for processing.

6.6.1 Environment

The data given to this module for testing was done through controlled generation of NetFlow records in order to test for correctness of the statistical principles applied in this module. These NetFlow records are then passed into Bolvedere's collector, which are then processed and passed onto the publisher before being broadcast out to this module. The results produced by this module can then be compared to precalculated results, as the dataset is completely known, and the modules correctness can be inferred from this point.

The dataset is made up of multiple hosts connecting to both monitored and unmonitored IPs and ports on the given emulated network. These tests were then run a second time excluding the unmonitored IPs and ports in order to see the effect of these records on the module. Five iterations of this are performed to ensure that the module produces consistent results. This has to be the case, as the same values applied to the same formula will always yield the same values in the statistical theory applied in this module.

A lead time of 25 records per port-IP pair was given to the module, as it needs to first generate a dataset in which to calculate the mean and, more importantly, the standard deviations for the given dataset. A dataset with one item will produce a mean of that single item with a standard deviation of 0. This means that the next point which is not exactly the original point would be flagged as an anomaly, as it will be an outlier; this result is fairly useless.

The given dataset window size for testing this module was 100 data points. This was seen to be ample in order to say something meaningful about recent traffic to a given pair when applying this test dataset.

6.6.2 Sudden Port Bandwidth Use Change Results

The initial batch of results generated over the five iterations of this testing proved to be 100% accurate in terms of the precalculated results for the used dataset, with every iteration yielding the same result at every point of each test iteration; this was deemed a success. Because of the

Table 6.10: Error Rate for Initial 25 Data Points

Data Point Range	Error (%)
1 - 5	84
6 - 10	76
11 - 15	36
16 - 20	8
21 - 25	0

lack of interesting points in these results, a change was made in the planned dataset, to see how many records the module requires to have seen before it starts to produce accurate results.

This was done by removing the 25 record lead time per port-IP pair in order to collect a dataset large enough to say something meaningful about subsequent data points. As expected, the module flagged 78% of initial 10 records across all pairs as outliers, and thus anomalous. However by the time all port-IP pairs had 21 data points the module was again accurate to that of the preprocessed results for the dataset. The counts for error per record received per pair can be viewed in Table 6.10.

With these results this module was seen to work correctly and as expected after enough data points are collected for each port-IP pair. This value may vary from network to network, but even in the off chance that the number of required records exceeds the 100 data point window given for this dataset, this window size can be adjusted to suit such cases.

6.7 Source IP Address Anomaly Detection

The module that deals with detection of Source IP Address Anomalies is tested in this section. As a recap of Section 5.6, this module pairs a NetFlow source node ID to a subnet which allows geolocating of a network flow regardless of the source IP. This module uses this information to then detect whether a network flow sourced from a network is legitimate or not, based on whether the source IP falls within the subnet expected from that network. Testing whether this module works or not is as simple as injecting spoofed source IP packets that are bound from an internal network to an external network, as shown in Figure 6.23. If the source IP in the packet is not that which belongs to the subnet of the internal network, the user should be notified.

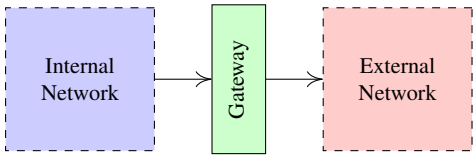

Figure 6.23: Internal to External Network Flow

6.7.1 Environment

The network environment used was the same as that in Section 6.4 and can be referred to in Figure 6.21. The reasons for this is that in the environment set up in Section 6.4 there exists multiple networks each running NetFlow source nodes with unique IDs. Furthermore, each internal network in the environment has its own subnet allocated to it; this is the ideal environment to test this module.

The changes made will be that the bots running in these networks will continue operation as per usual, however these systems will periodically transmit a packet with spoofed source IP destined to the external network. This will generate a new outbound network flow which this module will then be expected to detect as an anomaly.

6.7.2 Controlled Generation of Source IP Spoofing

As explained in Section 6.7.1, this testing will be performed through injection of spoofed network traffic into legitimate network traffic created by the aforementioned bots. This spoofed traffic's IP addresses will be selected at random, outside of the subnet assigned to the network in which the spoofed traffic is coming from.

Expected results for this testing will be detection that a network flow is indeed spoofed, this will be done through comparison of ingress and egress interfaces, as well as the source IP address seen. The second objective of this testing is to locate which network a spoofed network flow originated from. This should be easily achievable through use of the source NetFlow node's unique ID assigned to it.

6.7.3 Controlled Generation of Source IP Spoofing Results

The graph shown in Figure 6.24 summarizes all that is to be said about this round of testing in terms of the ability for this processor module to detect spoofed traffic. The 100% detection

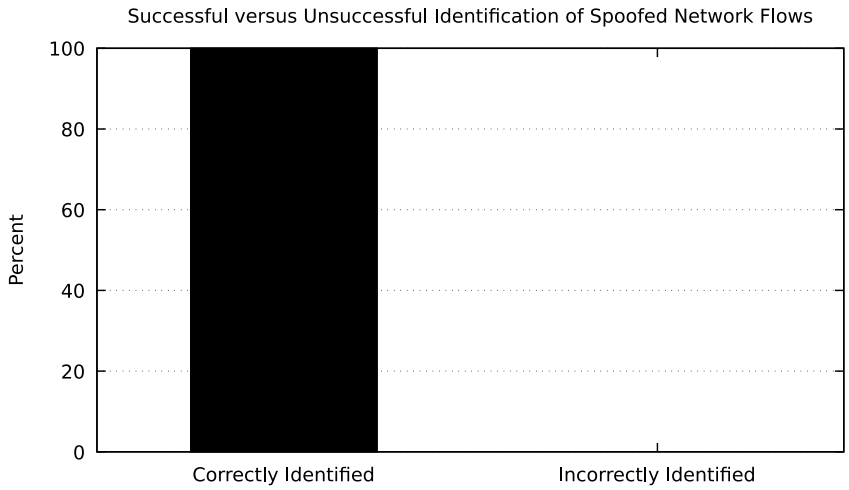

Figure 6.24: Success of Spoofed Network Flow Detection

Listing 6.10: Terminal Output of Spoof Detection Processor Module

```
1 ...
2 [Okay]     FSID 200; Source IP: 212.72.100.201
3            Destination IP 180.149.132.47
4
5 [Warning] Anomaly detected on 200; Source IP: 54.186.76.208
6            Destination IP 104.16.67.203
7 ...
```

ratio is due to the simple fact that a packet's source IP either belongs to a network, or it does not. For this reason one can say with absolute certainty whether a packet is meant to be seen at an ingress interface of a gateway or not. Referring to the terminal output of the processor module used in verbose mode for debugging in Listing 6.10, one can see two selected logs one after each other chronologically. The subnet assigned to FSID 200 was 212.72.100.0/24, thus legitimizing the first log generated with source IP 212.72.100.201. However, the very next log generated was a network flow with the source IP 54.186.76.208, which did not belong to FSID 200's subnet. This was immediately detected and a warning generated for it. Furthermore, this result shows that one can say that that spoofed network flow was generated from the subnet attached to the ingress interface of the gateway that owns FSID 200.

6.8 Malware NetFlow Fingerprinting

This module sought to collect NetFlow logs generated by malicious network flows on a network, and then analyse them for generating rule sets for use in Bolvedere to detect further attempts

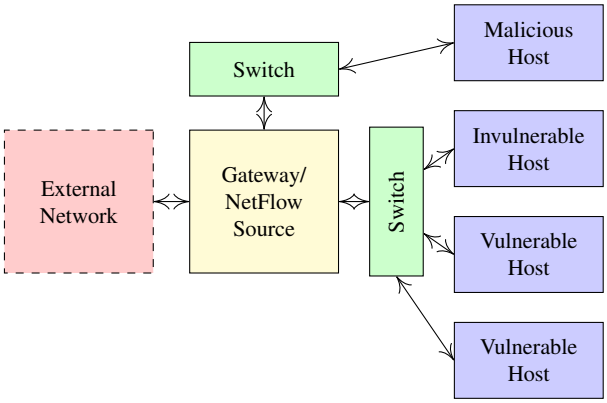

Figure 6.25: Virtual Network Overview

of these attacks. For this reason this result section will be broken down into two major parts. The first part will deal with the collection and analysis of NetFlow logs generated by controlled malicious attacks in order to build rule sets to detect these attacks, and the second part of these results will be targeted at the running of the automated Bolvedere system module that implements these rule sets to detect these recorded malicious attacks.

6.8.1 Environment

These tests were all performed within a virtual environment. The basic configuration of the virtual network this virtual environment used can be seen in Figure 6.25, and one can see that hosts are broken up into 4 groups. These are listed below with a short description:

Malicious Hosts: These hosts launch malicious attacks on vulnerable and invulnerable hosts, typically through the use of Metasploit or custom written code as to the documented exploitation attack vector.

Vulnerable Hosts: These hosts are built to be vulnerable to a monitored attack.

Invulnerable Hosts: These hosts are built to be made resistant to a monitored attack.

Gateway/NetFlow Source: This host acts as a gateway to an external network (advertises that it is connected to the Internet) and also runs the NetFlow log generator to store the network flows that pass through it. This host uses Softflowd to generate all NetFlow logs (Miller, 2016).

One can observe that, in order for the malicious host to communicate with a vulnerable or invulnerable system, it has to first pass through the gateway system running NetFlow. This

means that the network flows generated between these hosts can be fully logged and analysed into rule sets at a later point.

6.8.2 NetFlow Logs and Rules Generated

The purpose of this section is to collect the NetFlow logs generated by Softflowd when observing the network flows caused by malicious attacks. The defining features of these generated logs are then extracted and used to form rule sets that can discern further attempts of the attacks that generated the network flow. The logs generated have been tabulated and can be referred to in Tables 6.11 to 6.18. These results were gathered over 6 iterations, of which 3 iterations were designed to be successful and 3 were designed to fail. The failures did not show a large variance in results and so have been omitted but will still be discussed later in this section. Also, due to the manner in which these results were collected there, is some variance. This is handled through displaying the result as a range rather than a set value where necessary; this lines up with the confidence level discussed in Section 5.7.

Terminology used in these results is explained below:

Attacker: The host that is launching the exploitation.

Target: The host which the attacker is attempting to exploit.

Victim: A third-party that is affected due to an Attackers exploit.

A, B and C: These refer to randomly assigned ports by the operating system when a connection is created without being told to use a specific port.

N: This refers to all consecutive numbers after the last until process is terminated.

X and Y: These refer to counters of varying size relating to packet and byte counts.

Exploit: Used to define the stage of the exploit in which the exploitation is being attempted.

Payload: Used to define the stage of the exploit in which the payload is being transferred and executed on the system.

Runtime: Used to define the stage of the exploit in which the payload is running.

Table 6.11: NetFlow Logs: Successful ms08_067_shell Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	445	43 - 47	9,900 - 10,100
2	Exploit	target → attacker	445	A	42 - 44	7,600 - 7,700
3	Payload	attacker → target	B	Set in Exploit	8	695
N	Runtime	Bi-Directional	B/C	Set in Exploit	X	Y

Table 6.12: NetFlow Logs: Successful ms08_067_vnc Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	445	43 - 47	9,900 - 10,100
2	Exploit	target → attacker	445	A	42 - 44	7,600 - 7,700
3	Payload	attacker → target	B	Set in Exploit	278	416,549
N	Runtime	Bi-Directional	B/C	Set in Exploit	X	Y

Table 6.13: NetFlow Logs: Successful java_rmi_server Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	1099	6 - 7	358
2	Exploit	target → attacker	1099	A	7	567
3	Payload	attacker → target	A	Set in Exploit	7	7,400 - 7,500
N	Runtime	Bi-Directional	A/B	Set in Exploit	X	Y

Table 6.14: NetFlow Logs: Successful distcc_exec Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	3632	7	656
2	Exploit	target → attacker	3632	A	4	276
3	Payload	attacker → target	B	Set in Exploit	4	216
N	Runtime	Bi-Directional	B	Set in Exploit	X	Y

Table 6.15: NetFlow Logs: Successful samba_symlink_traversal Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	445	10	975
2	Exploit	target → attacker	445	A	8	790 - 800
N	Runtime	Bi-Directional	B	Set in Exploit	X	Y

Table 6.16: NetFlow Logs: Successful samba_usermap_script Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	139	7	733
2	Exploit	target → attacker	139	A	4	356
3	Payload	attacker → target	B	Set in Exploit	3	164
4	Payload	target → attacker	Set in Exploit	B	2	135
N	Runtime	Bi-Directional	B	Set in Exploit	X	Y

Table 6.17: NetFlow Logs: Successful unreal_ircd_3281_backdoor Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	Set in Exploit	3	164
2	Exploit	target → victim	Set in Exploit	A	2	135
N	Runtime	Bi-Directional	A	Set in Exploit	X	Y

Table 6.18: NetFlow Logs: Successful ntp_mon_list Exploit

Flow Index	Exploit Stage	Direction	Source Port	Destination Port	Packet Count	Byte Count
1	Exploit	attacker → target	A	123	1	60, 90 or 234
2	Exploit	target → victim	123	A	up to 10	up to 4,460

6.8.3 Results

Unsurprisingly, the first point to note is that it is the attacker that always starts the communications in these exploits. The method is usually performed through fingerprinting a target to identify which services are running on a system (these logs aren't displayed). Once a vulnerable service has been identified, the exploit is then launched and, if successful, the payload is then uploaded to the vulnerable host and an attacker gains access to the target in their chosen method. As these vulnerabilities are found in services running on a host and these services run on specific ports, it is to note that these specific ports are what an exploit targets.

A significant point that arose when an exploit was repeated was that the initial NetFlow log's packet count, byte count and service port were consistent (the service port consistency is important as some services utilize multiple ports). This means that one can say that for a new NetFlow log between two hosts, if a set port is connected to that receives a set packet count with set total byte count, one should check that targeted host for an occurrence of an attack that is represented by this signature. Although one should also note that as a NetFlow log only contains the metadata of a network flow, a perfectly legitimate network flow could also cause this NetFlow log to be generated.

Some finer details to notice are that MS08-067 (exploitation of Microsoft RPC service) exploitations tend to have their packet count and byte count vary more than exploits utilizing other vulnerabilities in these results. Another point is that the NTP monitor list attacks were generated using 3 separate monitor list request packets, these were of size 60, 90 and 234 as found in the wild (Rudman and Irwin, 2015, 2016). As this attack is an UDP-based reflection attack, the attacker did not receive any feedback as to success or failure of their attack, and instead only a response was generated by an NTP server to the victim which the attacker intended to DDoS. For this reason the attacker also requires the use of a third-party discovery tool, such

as Ping, to see whether the victim was still reachable or not (these ping logs were not shown as they are not part of the exploit tested).

The failures of these exploits for the most part resulted in a TCP reset at some point in the exploit attempt. The resulting NetFlow logs depict this with an initial flow from the attacker with a response flow of 1 packet that is 46 bytes in length. The only two exceptions to this were the MS08-067 based attacks, which showed a response flow from the target before a follow up flow was generated in order to access the payload which was responded to with a flow of the aforementioned TCP reset. The second was the NTP-based attacks which, because they were UDP-based, showed no response to the exploit of any form.

6.8.4 Rule Sets Generated

Tables 6.11 to 6.18 in the results section, Section 6.8.3, are in the format of the rule sets that will be given to the Bolvedere module that will attempt to discern these exploits[7]. It is notable that the rules outlined by these tables require far fewer checks in an attempt to discern a network flow than deep packet analysis does; this is due to the fact that every packet in a network flow doesn't get analysed, but rather it is the existence of a network flow that matters most. Coupling this with the sheer reduction in the amount of throughput the overall system has to handle, as a NetFlow log only contains the metadata of a network flow, this allows for multiple NetFlow source nodes to sink their generated logs into fewer hosts running Bolvedere than the equivalent amount of hosts required for a deep packet analysis solution.

6.8.5 Automated Module in Action

In order to check proper functionality and usability of this Bolvedere module, one has to provide control data for the results to be compared against. For this reason legitimate network traffic is required to the services running on the vulnerable host. In this testing, the legitimate connections and use of the services on the vulnerable host was performed by bots. These bots were programmed to perform simple tasks that required use of these services at random times, ranging between 500 milliseconds and 10 seconds. One must note that the vulnerable target host was running every exploitable service in which the rule sets were generated for allowing for ease of testing[8]. Furthermore, Microsoft Windows services were made available on this system through use of a Windows XP virtual machine, running on the vulnerable host configured

[7] The tables were developed this way to save space.

[8] This system is provided by RAPID7 and is available for download at https://information.rapid7.com/metasploitable-download.html

Listing 6.11: Terminal Output of Bolvedere Module

```
[10.42.0.45:45677 -> 10.42.0.33:445,
 Size:9991,
 Count:44]: Potential ms08_067_shell

[10.42.0.13:34782 -> 10.42.0.33:445,
 Size:15232,
 Count:113]: Seems Legit

[10.42.0.68:40029 -> 10.42.0.33:123,
 Size:60,
 Count:1]: Potential ntp_mon_list

[10.42.0.13:37087 -> 10.42.0.33:139,
 Size:23011,
 Count:146]: Seems Legit

[10.42.0.103:28928 -> 10.42.0.33:445,
 Size:18002,
 Count:174]: Seems Legit
```

to bridge its network interface with that of the vulnerable host system. At runtime, the bots were first enabled to start communicating with the vulnerable host, and then the attacks were manually launched and results observed through the terminal output of the Bolvedere module.

Testing occurred over 124 separate connections, consisting of multiple network flows depending on the task at hand. The success or failure of a result was considered on a per connection bases, and were discerned as to whether a connection was malicious or not by the NetFlow logs generated by the entire connection. Terminal output of this module can be referred to in Listing 6.11 which includes a false positive regarding the detection of a *ntp_mon_list* attack. This was in fact a legitimate request for the monitor list from the NTP server. Referring to Figure 6.26 one can see the results produced by these 124 separate connections.

Of these 124 connections 117 were successfully identified as either a malicious or legitimate connection, where the only 7 failures were false negatives produced when trying to determine whether a monitor list request from the NTP service was legitimate, or part of a DDoS attack. This means that the rule set produced for this Bolvedere module is 94.355% accurate when attempting to discern the exploits recorded in Tables 6.11 to 6.18.

These results suggest that detection of malicious activities, when legitimate network flows closely resemble that of malicious flows, becomes difficult. In the case of *ntp_mon_list*, this is because a legitimate request is used to exploit an amplification attack on a victim, which is near impossible to detect against other legitimate requests. For this reason it is suggested that further revisions take into account previous connections made by an IP address, however potential

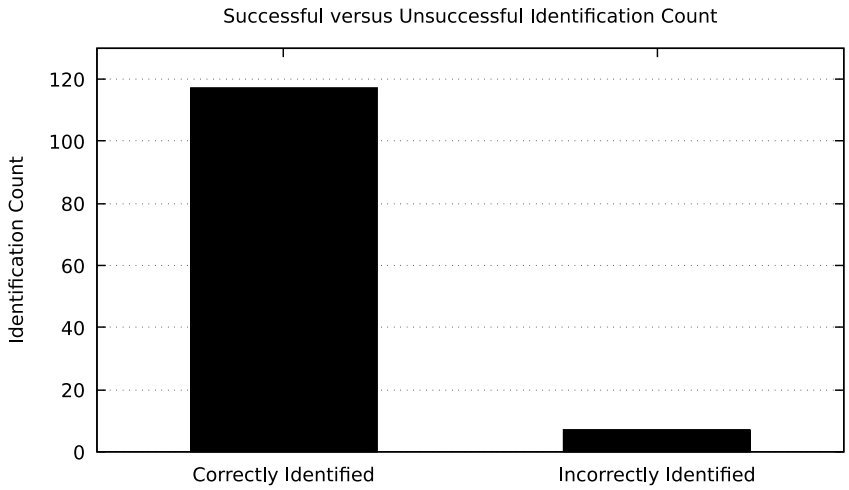

Figure 6.26: Comparison of Success versus Failure of Network Flow Identifications

memory requirements of the host system should be considered before this step is taken.

Considering the high level of accuracy produced by this module when considering the given rule set and, non-*ntp_mon_list* exploit and legitimate connections, these results hold promise into extension to detection of other malicious connections. In all, the fact that no malicious connections were missed, even though there were false positives, means that this Bolvedere module has served its purpose successfully.

6.9 Summary

This chapter sees the testing of the Bolvedere base system (comprised of both collector and publisher) and processor modules designed and implemented in **Chapter 5**. Each section outlines what was tested, why it was tested, and the environment in which testing occurred. The testing started in **Section 6.1** by looking at the correctness of the collector, publisher and then base subsystems of Bolvedere. Once this was determined to be working in a correct and expected manner, which was important as every module relies on the correctness of the base system, testing of each processor module then occurred.

Module testing starts in **Section 6.2** with the R-based neural network module, which sought to detect DDoS attacks. This was followed by testing of the Fourier analysis and then the reputation analysis modules in **Sections 6.3** and **6.4**. Testing of port anomaly detection modules was then performed in **Sections 6.5** and **6.6**. Source IP anomaly detection was tested in **Section**

6.7 before finally drawing the chapter to a close with **Section 6.8** which tested the vulnerability exploit detection module.

In all, these modules all worked in their expected manner given their configuration, and in the case of the neural network, its training set. Each module showed strengths in its targeted scope, except for the Fourier analysis module, which demonstrated its shortcomings when dealing with a dataset in which the time domain (which Fourier analysis bases its calculations on) is aggregated by the NetFlow protocol. This was an expected outcome of such a processing.

In all, Bolvedere in its entirety was proven to work in the manner expected under a controlled environment, which concluded success in itself, and showed promise for real-world application. An implementation of Bolvedere in a real-world application follows this chapter in **Chapter 7**.

7

Real-World Application

THIS chapter demonstrates the capability of Bolvedere to interact with real network data. The reason for this is that, up until this point in this document, datasets have been crafted in a synthetic manner to reflect specific recorded characteristics, or have been selectively filtered out of existing datasets in a controlled manner. Even though one can combine these datasets and include real-world traffic into a test dataset, the results concluded from them in terms of the real-world success of Bolvedere will never be as conclusive as an actual test on real network data.

Section 7.1 starts by showing basic metrics around the recorded dataset, as well as the basic structure of the network. This chapter leads on with **Sections 7.2** and **7.3** discussing the rate at which one can expect to process NetFlow records using Bolvedere, followed with a break down of how long each processor module takes to process data. It was noted that the rise of the Mirai botnet (Kolias *et al.*, 2017) occurred during the 17 months the dataset was recorded, and details of Mirai are brought forth in **Section 7.4**.

The neural network DDoS detection module is the first of Bolvedere's processor modules that is tested, and analysis and discussion of its findings are presented in **Section 7.5**. Results concerning port scan detection and detection of sudden changes in bandwidth use of ports and their

Table 7.1: Network Totals Overview

Index	Year	Month	Packet Count	Traffic (GB)	Flow Records
1	2015	Jul	6,981,414,596	1,363	114,191,149
2		Aug	5,356,590,331	887	115,368,660
3		Sep	8,914,154,692	2,099	166,142,920
4		Oct	9,206,301,273	2,239	171,567,671
5		Nov	10,454,546,950	2,047	191,432,860
6		Dec	2,226,338,868	1,121	51,819,800
7	2016	Jan	7,835,083,592	1,920	133,233,053
8		Feb	9,325,696,716	2,226	193,933,750
9		Mar	10,150,001,770	1,975	250,342,199
10		Apr	8,513,790,538	1,775	224,834,832
11		May	13,718,167,671	2,811	327,872,388
12		Jun	11,196,382,368	2,179	262,838,935
13		Jul	12,994,489,128	2,379	201,854,916
14		Aug	7,535,088,754	1,352	170,690,519
15		Sep	6,653,231,772	1,612	132,865,240
16		Oct	13,035,429,739	3,703	250,458,526
17		Nov	5,103,117,299	1,387	102,083,158
Total			149,199,826,057	33,082	3,061,530,576

associated modules is then analysed in **Sections 7.6** and **7.7** respectively. Detection of source IP anomalies is discussed in **Section 7.8**. The final module used in monitoring the recorded network is the vulnerability fingerprint detection module, and its results are detailed in **Section 7.9**. This chapter then draws to a close with a brief discussion of the failure in application of the Fourier analysis module in **Section 7.10**.

7.1 Dataset

The real-world dataset under analysis was collected entirely by nfcapd, which was configured to use NetFlow v9 as its logging protocol. Data was collected over a 17 month time period from of the beginning of July 2015 to the end of November 2016; there were no significant interruptions during this time frame. The dataset totals 33 TB which averages to a throughput rate of 750.787 KBps. A total of 149,199,826,057 packets were seen during this time frame, amounting to 3,386 packets per second. The average packet size for the dataset is 227 bytes. The number of network flows recorded in this dataset was 3,061,530,576 unique NetFlow version 9 network flow records. This information is tabulated in Tables 7.1 and 7.2 and has also been graphed in

Table 7.2: Network Averages Overview

Index	Year	Month	Packets/Sec	Kilobytes/Sec	Flows/Sec
1	2015	Jul	2,693	526.141	44
2		Aug	2,066	342.322	44
3		Sep	3,439	809.927	64
4		Oct	3,551	863.850	66
5		Nov	4,033	789.753	73
6		Dec	858	432.555	19
7	2016	Jan	3,022	741.029	51
8		Feb	3,597	859.099	74
9		Mar	3,915	761.992	96
10		Apr	3,284	684.852	86
11		May	5,292	1,084.755	126
12		Jun	4,319	840.836	101
13		Jul	5,013	918.135	77
14		Aug	2,907	521.815	65
15		Sep	2,566	622.124	51
16		Oct	5,029	1,428.958	96
17		Nov	1,968	535.228	39
Average			3,386	750.787	69

Figures G.1, G.2 and G.3, which can be found in Appendix G as to better visualize the numbers presented. The fields recorded in each record are listed in Table 7.3, the order present within the template record.

The dataset was processed in one month chunks, and measures were taken to ensure that flows that extended months leading into the one being processed were included in the dataset in order to keep the stateful nature of NetFlow true. Python-based processor modules used in this testing ran using the Pypy Python interpreter (Biham and Seberry, 2006). The total time taken to collect all processor modules' results was 25 days.

7.1.1 The Network

Blinding was used on the all data collected from the monitored network. This was done due to privacy concerns for the users of the given network. The monitored network existed with a single /24 subnetwork block, and so a simple blinding has been performed for reporting in this document. The IPv4 block used for this remapping was TEST-NET-1 (192.0.2.0/24) as defined by RFC 5735 (Cotton and Vegoda, 2010) which is defined specifically for documentation

Table 7.3: Softflowd's Recorded IPv4 Template Fields In Order

Field Name	Field description
IP_SRC_ADDR	IPv4 source address
IP_DST_ADDR	IPv4 destination address
LAST_SWITCHED	System uptime since the last packet of this flow was switched
FIRST_SWITCHED	System uptime since the first packet of this flow was switched
BYTES	Byte count
PKTS	Packet count
INPUT_SNMP	Input interface index
OUTPUT_SNMP	Output interface index
L4_SRC_PORT	TCP/UDP source port
L4_DST_PORT	TCP/UDP destination port
PROTOCOL	IP protocol used
TCP_FLAGS	Count of all flags seen for this flow
IP_PROTOCOL_VERSION	Version of the IP protocol used

and example code. Additionally, TEST-NET-2 (198.51.100.0/24) is used for IPs not belonging directly to the monitored network, but may still require protection of privacy in some form.

The monitored network's core router connects directly to the South African National Research Network (SANReN) network[1] through a firewall on which nfcapd was executed. Six schools were connected to SANReN through the core router, each having their own gateways which implemented NAT to/from their internal networks. Each school's network was named according to a random country name generator[2] for ease of reference in later text. The names of each subnet can be found in Table 7.4 accompanied by their allocated subnetworks according to their original subnetwork size and each of their default gateways. A diagram depicting the configuration of this monitored network can referred to in Figure 7.1. Permission was obtained by the school's consortium before collection of this dataset took place.

External networks monitored in this dataset, these are networks found in the "*External Network*" block in Figure 7.1, were not blinded throughout the results presented in this chapter. The estimated user base of all six school networks combined is around 2,000 users. NetFlow monitoring performed by nfcapd was done on the interface connected to the *external network* (SANReN).

[1] http://www.sanren.ac.za/

[2] http://www.fantasynamegenerators.com/country_names.php

Table 7.4: Network Name and Subnet List

Pseudo-Name	Subnetwork /x	Primary Gateway	IP Size
Firewall	192.0.2.0/30	-	4
Core	192.0.2.64/26	-	64
Seblor	192.0.2.128/27	192.0.2.130	32
Aplana	192.0.2.160/27	192.0.2.162	32
Ocron	192.0.2.192/28	192.0.2.194	16
Ocrington	192.0.2.208/28	192.0.2.210	16
Juwhiestan	192.0.2.224/28	192.0.2.226	16
Brevania	192.0.2.240/28	192.0.2.242	16

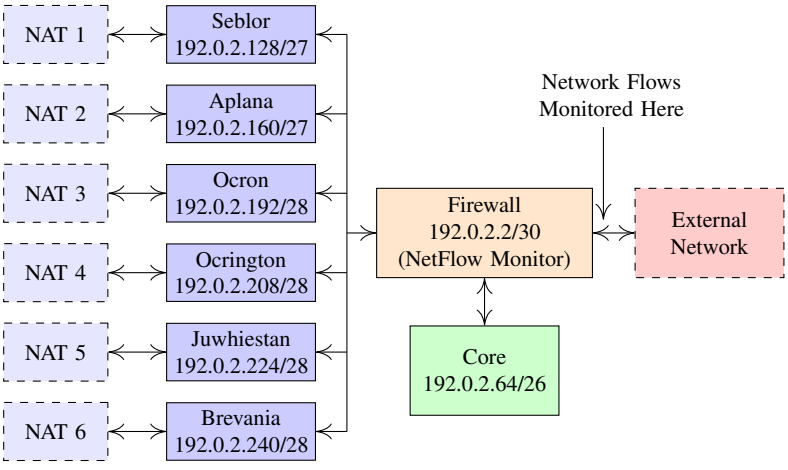

Figure 7.1: Network Overview

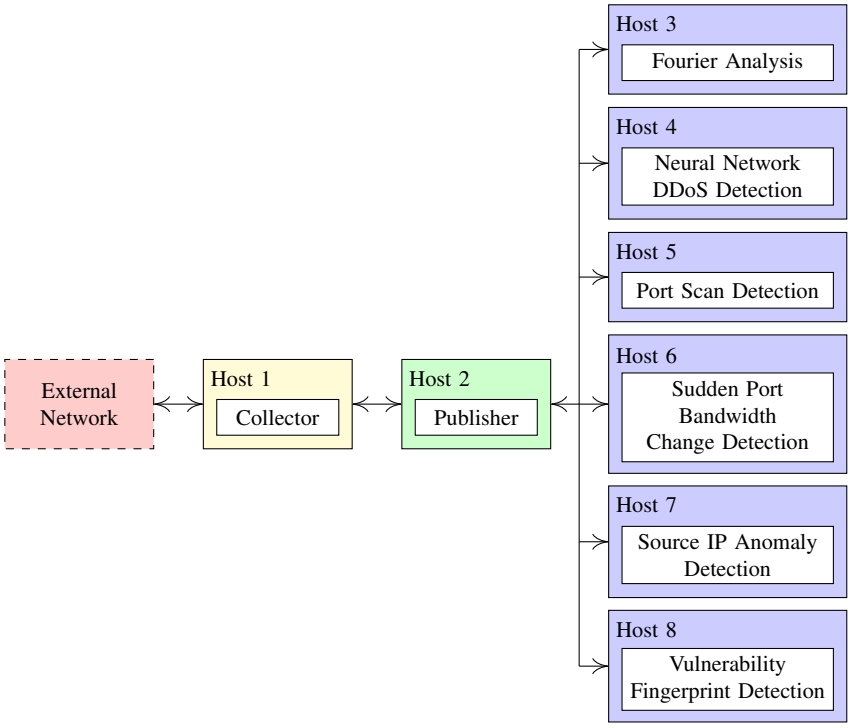

Figure 7.2: Host Configuration for 17 Month Data Processing

7.2 Completeness Testing of Bolvedere

In this section the ability for Bolvedere to function correctly in a distributed manner, as well as its ability to process all data on the given link, is reasoned. These two aspects of Bolvedere are important, as the former allows for leverage of modern multi-threaded processor and distributed computational architectures, where as the latter is required in order for Bolvedere to be completely accurate in its capacity[3]. If Bolvedere was unable to process every record that it received it would lose evidence of events which could be required to correctly classify a record.

7.2.1 Distribution

The testing environment used followed the structure shown in Figure 7.2. One can see from this configuration that each module was distributed onto a separate host which all subscribed to a common publisher (one can refer back to Figure 4.1 for an overview of the Bolvedere architecture). Throughout testing, all modules running on the 6 separate hosts (Hosts 3 to 8)

[3]if data is missed it could lead to a loss of accuracy if a processor module requires previous data in order to calculate a later result (stateful in nature)

received all records published by the publisher and processed them according to the module on the host. This was validated through comparison of the NetFlow data logs received by each processor module to the number of NetFlow data logs in the entire dataset.

One can also see that the NetFlow records were replayed from the *External Network* into the collector, which then passed on its results (this being the re-ordered data logs discussed in Section 4.3) on to the publisher. All three of these components mentioned also ran on their own separate hosts, further justifying the ability for Bolvedere to run in a distributed manner.

Although these tests were performed through a NetFlow replay program (nfreplay[4]), this program replayed the NetFlow packets as one would receive them on a live interface. This further justifies Bolvedere's ability to run on a live network, which was proven in Chapter 6's initial functional testing. The reason for using this replay mechanism is due to two reasons. The first is that this system did not exist 17 months prior to its completion and so could not be run live for 17 months to get these results. The second reason is simply that it is faster to stream the data through the system in this mode than waiting 17 months for the data set to be generated. This would simply take too long for the results that are required by this research, as this implementation needs to remain current and waiting over a year for a result would severely hinder this fact. It must be noted at this point that this means that Bolvedere can be used to analyse historical data.

7.2.2 Correctness

The maximum throughput Bolvedere's base system can handle can be tested by ensuring that, for all records sent by host running nfreplay to the collector, the required records by the publisher are distributed out of the publisher to the processor modules in the order required by the publisher. A monitoring system was set up with 2 network interfaces listening to all traffic sent by nfreplay and sent by the publisher to the modules, Figure 7.3 depicts this configuration. The monitoring system was also configured to know the order in which the collector passed records onto the publisher, as well as which fields of the records were published by the publisher. This meant that the monitoring system could check that for a given input, the output generated by the monitored system was correct.

This was done by pregenerating the final correct output of the publisher for every NetFlow record generated by nfreplay beforehand. From here, when a NetFlow data record was transmitted by nfreplay, the known result could be looked up and then compared to the output by the monitored publisher.

[4] http://nfdump.sourceforge.net/

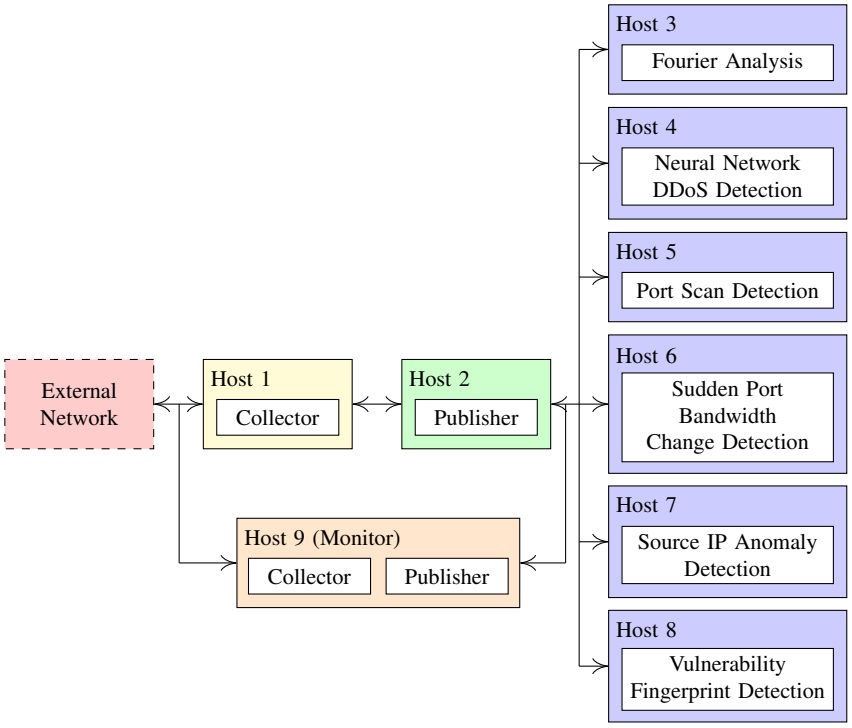

Figure 7.3: Host Configuration for 17 Month Data Processing with Monitor

The result of this testing was that of all NetFlow packets that were duplicated and introduced into both the Bolvedere base system and the monitoring system, none were dropped. Furthermore, the expected result for each processed NetFlow data record produced by the publisher was identical to the preprocessed expected output. From this one can conclude that the Bolvedere base system works as intended.

7.3 Maximal Throughput

Nfreplay has an option field for delay between transmission of NetFlow record packets of which each record packet contains 30 records (Claise, 2004). The point of this section is to test the minimum delay between record packets each module can handle before record loss occurs. Each module had record counting enable, thus allowing them to count the number of records they each successfully processed. This number could then be used against a dataset of known record count to see whether record loss had occurred. Each module was tested separately. The FFT processor module was omitted from this testing due to the fact that it performs batch processing every 5 minutes of runtime. The batch processing of this module stores up to the previous 24 hours worth of NetFlow logs, and during the entirety of this 17 month run did not

Table 7.5: Minimum Delay Between Processing a NetFlow Records by Module

Module	Minimum Delay (ms)
Neural Network DDoS Detection	3.897
Port Scan Detector	44.441
Sudden Port Bandwidth Change Detection	6.535
Source IP Anomaly Detection	3.771
Vulnerability Fingerprint Detection	5.076

see a batch take longer than 5 minutes to complete.

Testing of a module would start with nfreplay set to 100 ms between each data record packet. If the results of the test showed that all logs transmitted were received and processed by the processor module. The delay between the data record packets would then be reduced by 10 ms. This process would be repeated until record loss occurred. At this point the halfway delay time would be used between the last successful test iteration, and the one that just failed. This would be repeated until a successful iteration could be stated with a delay to third decimal place (this process can be directly compared to a binary search (Thomas *et al.*, 2009)).

Table 7.5 shows the minimum delay that each module can be given data record packets at before record loss occurs. It is worth noting that in none of these test iterations was the base system a bottleneck in the processing of records through the entire system.

The total number of NetFlow records generated over the entirety of the dataset was 3,061,530,576 which averages 68.406 records per second. This totals an estimated 0.00287 Mbps attributed to NetFlow Records[5]. Given that this entire 17 month dataset was sped up to be processed within a 25 day run time (due to the need to cater for the Port Scan Detection module's delay between record packets), this meant that the number of NetFlow records Bolvedere would have had to deal with per second was 1,476 totalling an estimated throughput of 0.0619 Mbps due to NetFlow records.

With respect to this network, the total bytes transferred amounted to 33,083 GB. This means that on average there were 11,603 bytes of network traffic counted per record packet. Using this result, one can estimate that Bolvedere, in the configuration of this testing, can deal with

[5]This is due to a recorded IPv4 data record being 42 bytes in length but a template record varying in size depending on the template being transmitted.

NetFlow records generated on a network that produces an average of 134 Mbps of throughput. Removing the Port Scan Detection module, Bolvedere could effectively be run on a network with an average of 951 Mbps of throughput. This equates to 10,040 NetFlow records per second.

A simpler way to look at these numbers is to realise that 17 months of data was processed in just 25 days; this means that 518 days of data was processed in 25 days. Effectively this configuration of Bolvedere for this network can receive 20.72 times more records on average per day before record loss would occur.

There is no definite way to calculate the number of records that Bolvedere can process for a given network without actually running that network's NetFlow records into Bolvedere, however these results serve as a good estimate. This is because each network will have different requirements for monitoring, and thus run different modules in different configurations that affect how long they take to process a record. On top of this, the fields that are recorded in the NetFlow records can vary. The results of these tests do however suggest that for this configuration, meaningful results can be discerned at a throughput approaching 1 Gbps. With more capable host hardware, this system can process NetFlow records generated by a network which produces more than 1 Gbps of traffic. To quantify this amount, this is 125 MB/s of sequential data (Seagate, 2011).

7.4 Mirai's Effect

Focussing on Tables 7.1 and 7.2 a distinct spike of network flows, packets and bytes transferred in May and October of 2016 is discernible[6]. A point of interesting here is that October 2016 sees the largest byte count of any other month in this dataset, and this will now be addressed.

Mirai (Section 2.5.6) was first discovered in August 2016 and performed two of its largest attacks at the end of September (Krebs, 2016) (targeting KrebsOnSecurity[7]), and middle of October 2016 (Bonderud, 2016) (targetting OVH[8] and DynDNS[9]). This would better account for the spike seen in October 2016 in the aforementioned tables. As stated in Section 2.5.6, the Mirai attack vector is that of telnet (port 23/TCP and port 2323/TCP) where it attempts to login into the targeted device using insecure login credentials.

[6]Please note that this work is previously published as part of Southern Africa Telecommunication Networks and Applications Conference 2018 (Herbert and Irwin, 2018).

[7]https://krebsonsecurity.com/

[8]https://www.ovh.com/

[9]https://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/

Listing 7.1: TShark Logs for Telnet Login

```
 1   1 0.000000 196.198.0.1->196.198.0.2 TCP      74 55020->23 [SYN]...
 2   2 0.000024 196.198.0.2->196.198.0.1 TCP      74 23->55020 [SYN,ACK]...
 3   3 0.000047 196.198.0.1->196.198.0.2 TCP      66 55020->23 [ACK]...
 4   4 0.000154 196.198.0.2->196.198.0.1 TELNET   93 Telnet Data...
 5   5 0.000166 196.198.0.2->196.198.0.1 TCP      66 23->55020 [ACK]...
 6   6 0.004727 196.198.0.2->196.198.0.1 TELNET   78 Telnet Data...
 7   7 0.004744 196.198.0.1->196.198.0.2 TCP      66 55020->23 [ACK]...
 8   8 0.004794 196.198.0.2->196.198.0.1 TELNET  105 Telnet Data...
 9   9 0.004804 196.198.0.1->196.198.0.2 TCP      66 55020->23 [ACK]...
10  10 0.004893 196.198.0.2->196.198.0.1 TELNET  145 Telnet Data...
11  11 0.005215 196.198.0.2->196.198.0.1 TELNET   69 Telnet Data...
12  12 0.005308 196.198.0.2->196.198.0.1 TELNET   69 Telnet Data...
13  13 0.005494 196.198.0.2->196.198.0.1 TELNET   69 Telnet Data...
14  14 0.005599 196.198.0.2->196.198.0.1 TELNET   69 Telnet Data...
15  15 0.005616 196.198.0.2->196.198.0.1 TELNET   86 Telnet Data...
16  16 0.043801 196.198.0.1->196.198.0.2 TCP      66 55020->23 [ACK]...
17  17 0.043820 196.198.0.2->196.198.0.1 TELNET   78 Telnet Data...
18  ...
```

Listing 7.1 shows a connection and successful login to a telnet server in a controlled environment. The columns of interest in this listing are columns 3 and 5 which depict the IPs involved in the connection and the size of each packet transmitted respectively. The login process for telnet, and sustained connection if a login is successful, consists of multiple small packet transmissions from the IP connecting to the device running the telnet server. This behaviour mimics the behaviour which the DDoS detection processor module's neural network uses to identify the SYN storms it was trained to detect (Section 5.1). This might account for a spike of detected DDoS attacks in October 2016 (Section 7.5).

Successful infection of a device by Mirai would result in it being added to a botnet which could in turn be used for performing a DDoS attack at a target. Devices in DDoS attacks typically spoof their source IP in order to prevent their targets from discovering the location of their device, or adding firewall rules to deny traffic from an attacking device (Jin *et al.*, 2003). If the school networks get infected by Mirai, this may lead to geographic anomalies being detected by the geographic anomaly module (Section 7.8).

7.5 DDoS Detection

This section sees use of the same neural network trained in Section 6.2.1. To summarize how this neural network classifies potential DDoS attack NetFlows after it was trained, one can refer to Table 7.6 which was previously stated in Section 6.2.1:

Table 7.6: Neural Network Classifiers

Classifier	Description
Lower packet count	Counts below 2,500 cause the resultant to increase even when the average packet size is low.
Higher average packet size	Found to increase the resultant from around 285 bytes and up.
Most common protocol seen used	TCP was noted to act as a deciding factor when edge cases were presented.
Number of unique ports	Has a minor effect on the result, but more unique ports cause an increase in the result.
Number of unique IPs	This had a negligible effect.

Classification of a set of NetFlows in this manner can lead to many false positives in real-world networks. The main contributing factor for this concern is that the packet count has the greatest weight on whether a set of NetFlows to a destination IP is part of a DDoS attack, or not. Some common protocols that use low packet counts in their entire communications are that of DNS (due to their single request, single response nature) and smaller web pages. Given that the network that was monitored is primarily used by school going children, it is expected that there will be a lot of web browser traffic, and thus DNS traffic and web pages. Furthermore, the fact that there are six networks positioned behind NATs will result in many source IPs destined to the few IPs held by the NAT gateway hosts.

To compound this problem, due to the nature in which NAT is implemented, external hosts responding to internal hosts of the NAT would result in the NAT's IP receiving many responses from many IPs to many ports (due to responses to multiple source ports used by a single IP to perform NAT). This would cause an increase in the resultant value from the neural network.

Tables 7.7 and 7.9 show the results collected by this neural network. The first thing to notice is that in every month there is apparent evidence of DDoS attacks. This is to be expected, as even if DDoS attacks were not present in this dataset, there will definitely be false positives generated simply by the students using the network.

Figures 7.4 and 7.5 are the DDoS detection counts per hour for the month of October 2016 depicted as a continuous line graph in Figure 7.4, and each day overlaid for the first 28 days of October 2016 in Figure 7.4. These two graphs provide strong evidence pointing towards most of the results produced being false positives.

The first point to note is in Figure 7.4 where the data shown is cyclic by nature with a period of 24 hours. This is better shown by overlaying each 24 hour period on top of each other in

Table 7.7: DDoS Attacks Detected by Real Use Case

Year	Month	Detection Count	Detections/Hour
2015	Jul	39,102,171	52,556
	Aug	33,992,405	45,688
	Sep	29,238,349	40,608
	Oct	42,979,576	57,768
	Nov	41,831,870	58,099
	Dec	24,396,484	32,790
2016	Jan	35,319,736	47,472
	Feb	51,485,583	73,973
	Mar	55,934,629	75,180
	Apr	41,400,378	57,500
	May	58,989,501	79,286
	Jun	60,366,972	83,843
	Jul	79,289,562	106,571
	Aug	82,752,887	111,226
	Sep	66,232,880	91,990
	Oct	82,405,544	110,760
	Nov	61,271,830	85,099

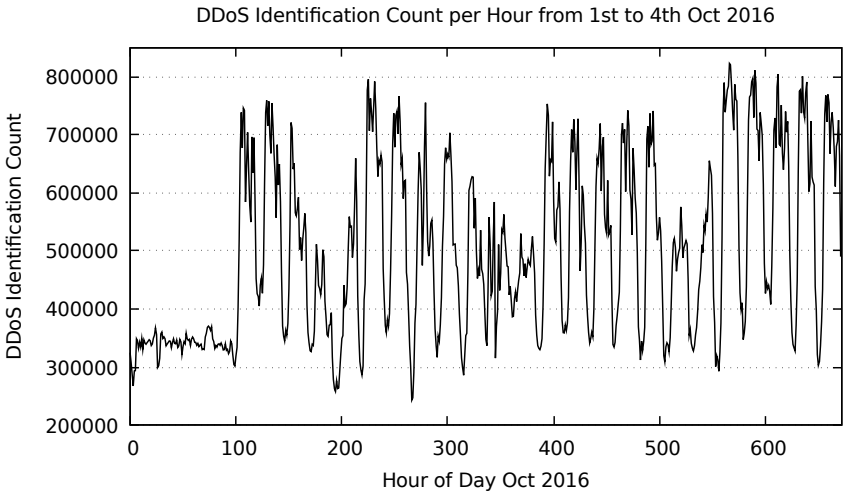

Figure 7.4: DDoS Attacks Detected by Hour in October 2016

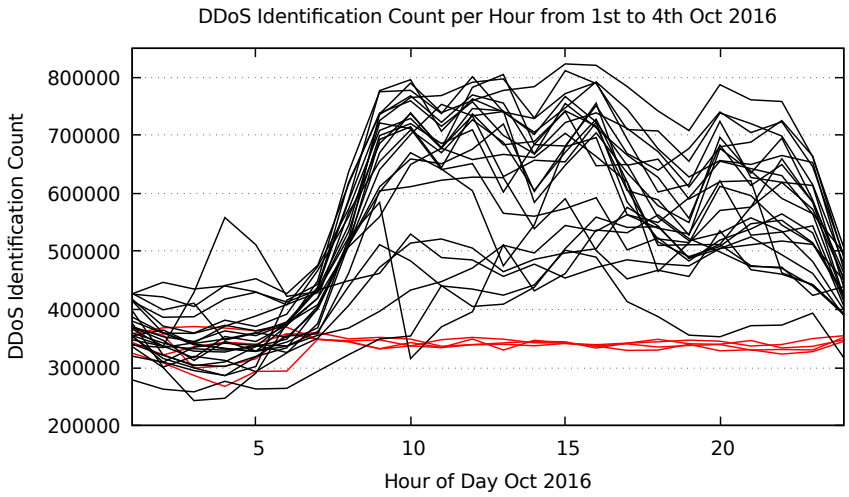

Figure 7.5: DDoS Attacks Detected by Overlaid Days in October 2016

Table 7.8: South African Public School Terms

Term	Start Date	End Date	Weeks	School Days
1	13 January	18 March	10	48
2	5 April	24 June	12	55
3	18 July	30 September	30	53
4	10 October	7 December	9	43

Figure 7.5. This is typical of a network with a high human user base, and the detection rates of the neural network following this trend strongly suggest that the inputs that are being classified as DDoS attempts by the neural network are due to legitimate human traffic (Willinger *et al.*, 1998; Jo *et al.*, 2012).

Aligning the depicted data with the public school holidays in South Africa found in Table 7.8[10][11], shows further evidence that the DDoS detections are due to human interaction with the network. The forth school term starts on the 10th of October and one will note in Figure 7.4 that traffic picks up just after the 100th hour; this is the 5th of October 2016 which is a Wednesday. This suggests that the increase of detections is related to the students returning to residences for the start of a new term.

What one should note at this point is that the detections for the first 4 days of October (marked up in red in Figure 7.5) are most likely not due to human interaction on the network. This means that the potential legitimate DDoS detection count per hour for this month is closer to 35,000

[10]This is a safe deduction as SANReN is based in South Africa.
[11]http://www.schoolterms.co.za/2016.html

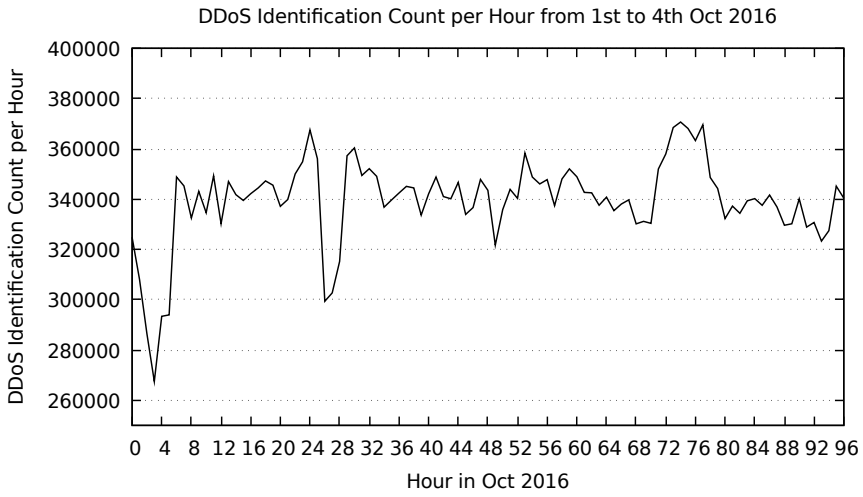

Figure 7.6: DDoS Attacks Detected in Days 1 to 4 of October 2016 by Hour

detections per hour than the recorded spikes of above 80,000. This number is still fairly large, and the fact that it is continuous for the entirety of the month with no response, supports the fact that a large portion of these detections are also false positives.

In this 4 day time period 53.7% of inbound traffic is sourced from port 443/TCP and a further 27.1% of inbound traffic comes from port 80/TCP. These ports belong to HTTPS and HTTP respectively and make up 80.8% of all inbound traffic. This means that more than 80% of traffic entering the NATs' internal networks are due to general web browsing, and thus most likely human interaction. This activity could be due to remaining staff that reside on campus, or due to the holiday being only 1 week long and students that chose to stay at school for the duration of this holiday. This is a possibility that is supported by the same 24 hour cyclic nature[12] found in these four days monitoring when referring to Figure 7.6. The problem with these conclusions is that the largest contributor of network traffic is *Seblor* which follows a 3 term school year. This format does not align its holidays with the 1st to the 4th of October 2016. Digging deeper into the original NetFlow logs sees no conclusive evidence of what may have caused this lull in network traffic.

Table 7.9 serves to give an overview of the top 10 targets of apparent DDoS attacks detected by this processor module. It is no surprise that 8 of the top 10 belong to NAT gateways to the schools' internal networks, and this expected behaviour verifies the correct functionality of this module. As previously stated, the process of the NAT performing an internal IP map to one of its ports, and performing the proxy request using the NAT's own IP, means that all relevant responses from the external network will be destined to the NAT's IP.

[12] Although to a much lesser degree.

Table 7.9: Detected DDoS Attack's Top 10 Targets over 17 Months

IP	Owner	Detection Count
192.0.2.130	Seblor	122,903,123
192.0.2.78	Core	85,333,109
192.0.2.80	Core	39,033,904
192.0.2.190	Aplana	36,022,383
192.175.48.42	DNS-OARC	31,550,528
192.175.48.6	DNS-OARC	31,537,835
192.0.2.210	Ocrington	18,006,122
192.0.2.162	Aplana	14,997,032
192.0.2.167	Aplana	12,850,742
192.0.2.170	Aplana	12,648,895

What this effectively means is that no IP on the NAT's internal network should ever be seen in the external network, but instead the NAT's IP will always be used as a source and destination. This has the side effect that any group of network flows that are viewed to be attempting a DDoS at a target on the internal network, will instead always be mapped back to the NAT that is acting as the proxy. This is the reason why the findings in Table 7.9 consist of mainly the NAT addresses to the schools.

The existence of the *Core* network taking up positions 2 and 3 of Table 7.9 does not come as a surprise either. The reason for this is simply that the *Core* itself acts as a proxy to the school's core systems. However, the existence of the two IPs 192.175.48.42 and 192.175.48.6 in positions 5 and 6 are of concern. IPs originating from the 192.175.48.0/24 block are reserved by the Internet Assigned Numbers Authority (IANA) for their "blackhole servers", and not allocated for public use on the Internet[13]. This block is specifically reserved to respond to 'bad' DNS requests or 'bad' reverse DNS requests[14] (Cheshire and Krochmal, 2013).

When a 'bad' DNS request (a DNS lookup to a non-existent domain, or a reverse DNS lookup to an unallocated IP) is performed, typically a legitimate response is generated by a DNS server from the 192.175.48.0/24 IP block to the requesting host (Cheshire and Krochmal, 2013). This 'bad' request is served with a legitimate response so that the requesting host refrains from performing follow up requests on the 'bad' request due to the host caching the legitimate response[15]. Resolving the domain names of these two IPs in question results in the names blackhole-1.iana.org and blackhole-2.iana.org for 192.175.48.6 and 192.175.48.42 respectively.

[13] https://www.iana.org/

[14] A DNS request to a non-existent domain or an IP that is not assigned or is reserved by IANA.

[15] This reduces load on the DNS servers found globally.

Table 7.10: Top 5 DDoS Detection Months Targeted at IANA "Blackhole Servers"

Month	192.175.48.6	192.175.48.42	Total
Mar	6,544,693	6,545,980	13,090,673
Apr	7,393,232	7,393,514	14,786,746
May	7,354,445	7,358,027	14,712,472
Jun	6,670,793	6,674,960	13,345,753
Jul	2,055,338	2,055,881	4,111,219
Total	30,018,501	30,028,362	60,046,863

A point of interest in this dataset is between the months March 2016 to July 2016 which contributed to 95.179% of the total traffic inbound from the 192.175.48.0/24 subnet block. The count for each of these 5 months according to the two 192.175.48.0/24 IPs that made the top 10 in Table 7.9, can be found in Table 7.10. This result is due to misconfiguration on the internal networks, as the IPs held accountable for targeting these 192.175.48.0/24 IPs all originate from the NAT gateways' IPs[16]. This is most likely due to the internal networks having an incorrectly configured, or no reverse DNS lookup mechanism, and so these requests are passed on to a DNS server in the *external network* (Cheshire and Krochmal, 2013).

This behaviour is typical of an internally facing DNS server configured to pass all requests to IANA reserved space through to one of the IANA blackhole servers, as to reduce erroneous DNS traffic on the internal network. This would result in many of the NATs sourcing DNS requests to IANA's blackhole servers. Furthermore, referring back to Test 12 of Table 6.1, this shows the result of NetFlow logs that one will commonly see generated by multiple DNS queries by multiple hosts (5 unique IPs, 5 unique destination ports, 1 packet, 120 byte average and performed over UDP[17]). This means that a group of these DNS requests would trigger the processor module's neural network, and thus the two blackhole servers found at 192.175.48.6 and 192.175.48.42 would be classified as being the target of a DDoS attack. This was rectified in early to mid June 2016.

7.5.1 Mitigating Neural Network Error

Some methods that could reduce error in the results produced by a neural network tasked at network anomaly detection are listed below:

[16] One should note that processor module is not concerned about direction.
[17] Window sizing set to the last 5 IPs observed.

Training Set: The training set is vitally important in order to achieve success in this Bolvedere module. The training set used in Section 6.2 and again here to train this neural network consisted of 100,000 items and took on average 2.003 seconds to train. This falls in line with the time taken to train results also found in Section 6.2. This means that this training set can be made larger, and still be trained in an acceptable time frame. The result of doing so will improve the accuracy of this Bolvedere module, and thus achieve more accurate results with less error.

Scope: Training a neural network to detect a type of DDoS attack is a viable tactic for, use of this processor module. As one can run multiple processor modules in parallel, one can train separate neural network modules to detect specific kinds of DDoS attack, and then run all these modules in parallel. This will result in more accurate detection of DDoS attacks, as a single monolithic neural network that is trained for multiple cases has a higher chance of error than neural networks trained for specific tasks. These neural networks also require less nodes in order to achieve their task, and thus are easier to process on resource limited systems.

Know Your Network: Consideration needs to be made for the network in which the neural network will be run on, and how inputs collected from the NetFlow logs generated by that network will be processed by the neural network.

7.6 Port Scan Detection

This port scan detection module was at the outset designed solely to detect vertical scans[18]. Due to the nature in which a NAT is implemented, the information gained from this processor module was found to be ineffectual in the context of the monitored network.

To explain this, consider the following example: multiple users on a NAT's internal network accessing a commonly frequented website such as Google[19], Facebook[20] or YouTube[21]. This would result in the NAT translating multiple internal requests to different ports, and making the requests to these websites on behalf of the internal hosts using its own IP. When the website's server responds to each request, each response will be destined to he same IP (this being the NAT host's IP) but on different ports, according to the NAT's translation table. To the port scan

[18] A port scan directed at multiple ports on a single IP
[19] https://www.google.com
[20] https://www.facebook.com
[21] https://www.youtube.com

Table 7.11: Unique Port Scanning Hosts Detected by Month

Year	Month	Detection Count	Unique IP Count
2015	Jul	134,777,189	1,566
	Aug	118,117,205	1,439
	Sep	153,474,410	1,533
	Oct	159,391,811	1,698
	Nov	177,920,402	1,792
	Dec	59,356,590	1,135
2016	Jan	135,659,784	2,101
	Feb	166,636,218	2,456
	Mar	171,847,852	1,921
	Apr	152,709,210	1,598
	May	186,118,233	1,985
	Jun	174,828,588	1,929
	Jul	290,731,187	2,705
	Aug	246,517,236	2,183
	Sep	208,190,022	1,983
	Oct	354,116,914	2,076
	Nov	155,017,694	1,512

detection module, this will look like a single host targeting multiple ports on a single host, and thus raise the possible port scan flag on the sourced IP. An equivalent situation would also occur with DNS requests to a common DNS server, as a commonly used DNS server would receive requests and respond to them in the same manner explained above.

Moving forward with this knowledge, and given that there are an estimated 2,000 users across all internal networks, the configuration for this processor module was set to 60 minute windows (after which the record for a particular port scan would be removed) with a 500 unique port count threshold. This configuration was used in order to allow for 25% of users on the network to connect to website without that site being detected as a source of a port scan. This was thought to limit false positives and focus down on typical port scans as performed by applications such as nmap (see Section 2.4.2) with the inclusion of some probe delay (Durumeric *et al.*, 2013; Barnett and Irwin, 2008). The effectiveness of this configuration will now be discussed.

A count of port scans and unique scanning IPs seen per month is shown Table 7.11. The seemingly large detection counts are partly due to the manner in which this processor module logs its findings. When a NetFlow log is received that causes a source IP to be pushed over the 500 unique port threshold in a single hour, and thus classified as a source of port scanning, it will be logged as such. However, if another log arrives within that same hour window (making the unique port count 501), that is treated as a disjoint event and also logged. This characteristic

Table 7.12: Top 10 External IPs Marked as Port Scanners

IP	Owner	Service	Source Port	Average Events/Month
155.232.135.5	ORG-UP1-AFRINIC	DNS	53/UDP	112
8.8.8.8	Google Inc.	DNS	53/UDP	90
162.246.18.19	Interserver, Inc	DNS	53/UDP	36
192.33.14.30	VeriSign	DNS	53/UDP	32
192.175.48.42	DNS-OARC	DNS	53/UDP	25
192.175.48.6	DNS-OARC	DNS	53/UDP	25
45.32.180.78	Choopa, LLC	DNS	53/UDP	24
23.229.5.19	Meridanhq Inc.	-	46345/TCP	23
8.8.4.4	Google Inc.	DNS	53/UDP	20
23.229.5.21	Meridanhq Inc.	-	46345/TCP	18

causes bloat in the logging mechanism and leads to inflated detection counts. That said, this characteristic of this processor module does not affect the ability of this processor module to detect port scanning hosts. To help ascertain information from this logging mechanism a unique IP count is also provided by the processor module and provided in Table 7.11 as well.

Investigating these results further one can refer to Table 7.12 to get the top 10 external IPs detected as having performed port scans by unique times detected. As expected, one will find that 8 of the top 10 IPs belong to DNS servers. For the sake of interest, further analysis will be performed on the 2 IPs that are not DNS servers.

The two IPs in question (23.229.5.19 and 23.229.5.21) belong to the subnetwork 23.229.5.16/29 allocated to Meridanhq Inc[22]. These two IPs were found to consistently source SYN packets from port 46345/TCP directed at sequentially interleaved ports on a single destination IP. This behaviour can be viewed in Listing 7.2 which shows a snippet of the NetFlow logs involved in a detected port scan. The NetFlow logs viewed in this listing are typical of a vertical portscan performed using the SYN scan method. This is shown in these logs by 23.229.5.21 having the 'S' flag raised, which signifies SYN, followed by 192.0.2.190's response with flags 'A' and 'R' raised, which stand for ACK and RST respectively.

When looking further into the 23.229.5.16/29 subnet block 2 more IPs belonging to it were discovered by this processor module to have also performed port scans. These IPs are 23.229.5.20 and 23.229.5.22, and, when viewing the original NetFlow logs, were found to have performed port scans in the same manner as shown in Listing 7.2. It was also found that the primary target of these port scans was 192.0.2.190, however 192.0.2.191 was also observed to be a target of

[22]Whois - updated: 2017-05-30.

Listing 7.2: Snippet of Original NetFlow Logs Detected as Part of Port Scan (Target 192.0.2.190)

```
Proto  Src IP Addr:Port      Dst IP Addr:Port  Flags Pkts Bytes ...
...
  TCP 23.229.5.21:46345 -> 192.0.2.190:48188 ....S.    1    46 ...
  TCP 192.0.2.190:48188 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:25804 ....S.    1    46 ...
  TCP 192.0.2.190:25804 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:48189 ....S.    1    46 ...
  TCP 192.0.2.190:48189 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:25805 ....S.    1    46 ...
  TCP 192.0.2.190:25805 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:48190 ....S.    1    46 ...
  TCP 192.0.2.190:48190 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:25806 ....S.    1    46 ...
  TCP 192.0.2.190:25806 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:48191 ....S.    1    46 ...
  TCP 192.0.2.190:48191 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:25807 ....S.    1    46 ...
  TCP 192.0.2.190:25807 -> 23.229.5.21:46345 .A.R..    1    40 ...
  TCP 23.229.5.21:46345 -> 192.0.2.190:48192 ....S.    1    46 ...
...
```

Listing 7.3: Snippet of Original NetFlow Logs Detected as Part of Port Scan (Target 192.0.2.190)

```
Proto  Src IP Addr:Port       Dst IP Addr:Port Flags Pkts Bytes ...
...
  TCP 23.229.5.19:46345 -> 192.0.242.191:13296 ....S.    1    46 ...
  TCP 23.229.5.19:46345 -> 192.0.242.191:13297 ....S.    1    46 ...
  TCP 23.229.5.19:46345 -> 192.0.242.191:13301 ....S.    1    46 ...
  TCP 23.229.5.19:46345 -> 192.0.242.191:13305 ....S.    1    46 ...
  TCP 23.229.5.19:46345 -> 192.0.242.191:13306 ....S.    1    46 ...
...
```

Table 7.13: Top 10 Institute IPs Marked as Port Scanned

IP	Owner	Event Count/Month
192.0.2.190	Aplana	375
192.0.2.210	Ocrington	107
192.0.2.130	Seblor	76
192.0.2.211	Ocrington	75
192.0.2.189	Aplana	50
192.0.2.100	Core	36
192.0.2.194	Ocron	34
192.0.2.75	Core	31
192.0.2.76	Core	10
192.0.2.170	Aplana	8

scanning by the 23.229.5.16/29 subnet block. As 192.0.2.191 is not assigned to a physical host on the network, it yielded no responses to any of the SYN packets destined to it. This most likely led to scanning being stopped on this destination IP, and hence why it does not appear in the top 10 most scanned destinations list in Table 7.13. A snippet of a scan directed at 192.0.2.191 from the 23.229.5.16/29 block depicting the lack of response can be viewed in Listing 7.3.

The top 10 source IPs detected to have performed vertical scans against the school networks when excluding known false positives (this being DNS responses), can be viewed in Table 7.14. There are two points of interest found in this list that one should note. The first is that the type of scan seen in each case in this list is that of the SYN scan (discussed above), and other than a change in the port number originating from 163.172.117.47 and 163.172.117.39, the method is exactly the same. The other point to note is that Meridanhq Inc. and Server Mania both exist under the larger organization subnet block allocated to B2 Net Solutions Inc.[23] and both use the same source port to perform their scans. This may be an indication of correlation in the tool used to perform these port scans.

Considering the results collected by this port scan detection module, with respect to the expected false positives detected, and the list of vertical scans detected, this processor module is shown to work under real-world conditions. This being said, there are some modifications that can be made to it to improve its utility.

Currently this processor module detects vertical scans, but fixing the destination IP and looking for variation in ports connecting to the fixed IP. If one were to swap these two roles one would be able to detect horizontal scans[24] as well.

[23] Whois - last-modified: 2016-02-22.

[24] Port scans that scan a single port across multiple IPs

Table 7.14: Top 10 External IPs Marked as Port Scanners

IP	Owner	Source Port	Event Count/Month
23.229.5.19	Meridanhq Inc.	46345/TCP	23
23.229.5.21	Meridanhq Inc.	46345/TCP	18
23.229.5.20	Meridanhq Inc.	46345/TCP	17
69.58.0.148	Server Mania	46345/TCP	15
69.58.0.147	Server Mania	46345/TCP	15
23.229.5.22	Meridanhq Inc.	46345/TCP	12
163.172.117.47	Online.net	22767/TCP	11
69.58.0.149	Server Mania	46345/TCP	10
69.58.0.150	Server Mania	46345/TCP	10
163.172.117.39	Online.net	22767/TCP	8

7.7 Sudden Port Bandwidth Change

Sudden increase or decrease of network traffic (in terms of bytes transmitted) to or from an IP on a given destination port is a good indicator of change within a network. Detection of such change does not mean something malicious has occurred, simply that the network's characteristic has changed in some manner. In the context of this network, this can be due to a service coming up or going down, a new popular game being played by students or, the use of a NAT translating internal IPs to ports.

Configuration of this processor module differed from the configuration used in Section 6.6 in that a window of 1,000 records was used for each port-IP pair. Furthermore, this processor module monitored all ports on the 192.0.0.0/16 subnet block which encompassed all NATs, as well as the *Core* and *Firewall*, on the monitored network. Ultimately, the two reasons all ports were chosen to be monitored on this subnetwork is because there exists multiple NATs on the monitored network, and that the port range that the NATs translate to is unknown. Memory was not seen to be an issue in this configuration as the number of IPs that were allocated in this network block were few.

Table 7.15 shows an overview of results collected for the entirety of the 17 months the network was monitored. One should note that this table is broken down into two columns which show the number of sudden bandwidth changes detected on inbound connections, and on outbound connections.

The symmetry between incoming and outgoing detections is to be expected. This is due to new connections being opened or connections closing, and would cause a change in both outgoing

Table 7.15: Sudden Port Traffic Increase/Decrease Detected in Real Use Case

Year	Month	Incoming	Outgoing	Total Count
2015	Jul	2,396,273	2,431,612	4,827,885
	Aug	2,538,480	2,539,781	5,078,261
	Sep	3,468,545	3,497,653	6,966,198
	Oct	3,741,679	3,795,305	7,536,983
	Nov	4,056,708	4,070,593	8,127,301
	Dec	1,121,278	1,121,580	2,242,858
2016	Jan	2,292,211	2,316,838	4,609,049
	Feb	2,429,327	2,445,843	4,875,170
	Mar	1,862,852	1,842,106	3,704,958
	Apr	1,695,911	1,710,402	3,406,313
	May	1,874,149	1,884,385	3,758,534
	Jun	1,934,862	1,941,650	3,876,512
	Jul	4,114,110	4,163,390	8,277,500
	Aug	3,749,106	3,768,360	7,517,466
	Sep	2,969,620	2,981,875	5,951,495
	Oct	5,050,888	5,115,863	10,166,751
	Nov	2,253,920	2,246,000	4,499,920

and incoming communications on the given port-IP pair. Paying attention to December of 2015 one can see a sudden detection count drop off when compared to that of the previous 5 months and following 3 months. This would be due to the December school holidays closing the schools and thus drastically reducing the user base of the monitored network. The knock on effect of this would be a reduction in events on the network, and thus a reduction in anomalous events detected by this processor module. This suggests that this module is working correctly as it lines up with the expected behaviour of the network.

Focussing again on October 2016, one can see the number of bandwidth increase and decrease anomalies by inbound and outbound connections in Tables 7.16 and 7.17. The symmetrical results of port 53/UDP in both inbound and outbound connections, and its place on both lists, is expected due to the manner in which DNS works. Some requests are larger than others, and at first this would cause anomalies to be detected in the number of bytes transmitted. However, as a long enough history is formed for the port on the monitored IPs, it would reduce the number of anomalies forming on it. This strongly points to this processor module's correct operation. A point of particular interest in both Tables 7.16 and 7.17 is the top port for detected anomalies, being port 45554/TCP. The reason this port is interesting is that it is an unassigned port according to IANA Cotton *et al.* (2011). Digging into the original NetFlow records one will find IP 192.0.2.189 sourcing a port scan from port 45554/TCP. This port had a sudden rise

Table 7.16: Top 10 Bandwidth Anomalies Detected by Destination Port for October 2016 (Inbound Connections)

Port	Increases	Decreases	Total Anomalies
45554/TCP	269	23,964	24,233
23/TCP	64	23,232	23,296
23/UDP	5	19,828	19,833
35030/TCP	781	16,036	16,817
53/UDP	4,290	10,303	14,593
14008/TCP	33	10,296	10,329
25/TCP	489	8,033	8,522
80/TCP	622	7,050	7,672
6889/TCP	35	6,307	6,342
53/TCP	695	5,185	5,880
29871/TCP	6	5,056	5,062

Table 7.17: Top 10 Bandwidth Anomalies Detected by Source Port for October 2016 (Outbound Connections)

Port	Increases	Decreases	Total Anomalies
45554/TCP	107	21,623	21,730
35030/TCP	825	11,039	11,864
25/TCP	248	8,068	8,316
6889/UDP	3	7,837	7,840
6889/TCP	5	7,633	7,638
14008/TCP	30	6,651	6,681
80/TCP	251	5,694	5,945
53/UDP	2,037	3,057	5,094
45554/UDP	110	3,462	3,572
53/TCP	10	3,365	3,375

Listing 7.4: Snippet of Original NetFlow Logs Detected as Part of Port Scan (Target 5.8.32.90 and 185.89.102.59)

```
Proto  Src IP Addr:Port         Dst IP Addr:Port  Flags Pkts Bytes ...
...
  TCP 192.0.2.189:45554 ->   78.31.67.12:51052 .AP.SF    9  2838 ...
  TCP 192.0.2.189:45554 ->     5.8.32.90:47575 .APRSF    7   354 ...
  TCP 192.0.2.189:45554 ->     5.8.32.90:57656 .APRSF    7   354 ...
...
  TCP 192.0.2.189:45554 -> 185.89.102.59:41241 .A..S.    2   120 ...
  TCP 192.0.2.189:45554 ->     5.8.32.90:46154 .APRSF    7   354 ...
  TCP 192.0.2.189:45554 ->     5.8.32.90:36128 .APRSF    7   354 ...
...
```

in use between the 10th of October 2016 and the 5th of July 2017, and was seen used for port scanning activity[25]. A snippet of these records can be found in Listing 7.4.

Most of the byte fields recorded by these NetFlow logs capturing the port scanning activity have the size 354. These NetFlow logs with the recorded 354 byte field make up 98.7% of all NetFlow logs recorded in this port-IP pair. The remaining byte counts seen are 120 bytes, 180 bytes, and less often than these two is 2,838 bytes. As records containing 384 byte fields make up the majority of all the records seen for this port-IP pair, this will set the mean of the 1,000 stored records close to 384. Furthermore, this will create a relatively small standard deviation for this port-IP pair. This means that when a record with byte field value of 120, 180 or 2,838 is received by this processor module, it will be seen as a sudden decrease or increase, as it will fall outside of 3 standard deviations from the mean.

Lastly, as the byte values 120 and 180 are seen more often than 2,838, this will cause more sudden decreases to be recorded by the processor module than sudden increases. This result strongly supports the correct functionality of this processor module.

A downfall of this processor module is that it only considers received logs in the context it was received in, with respect to previously stored records for a port-IP pair. This means that a record may be classified as anomaly but, when considered again after a new set of records have arrived, that anomalous log may be completely legitimate. With this in mind, a mechanism for reclassification of logs during their stay in the record window should be considered for implementation. This being said, the processor resource overhead for this implementation may make this solution unviable for large record windows.

7.8 Source IP Anomaly

Source IP anomalies in this research are defined as packets sourced from a network with source IPs that do not reside within the subnet block which the network uses. In the context of these tests, these are packets that sourced from the schools, and are destined to the external network with source IPs that do not reside in the 192.0.2.0/24 subnet block. This is a problem, as generation of these packets should not be possible under correct internal network configuration; the IP seen is effectively a bogon[26] to the internal network.

Such generation of packets can be accounted to:

[25] https://isc.sans.edu/port.html?port=45554 (Accessed 19th October 2017)
[26] http://www.outpost9.com/reference/jargon/jargon_17.html

Table 7.18: Source IP Anomalies Detected by Real Use Case

Year	Month	Detection Count
2015	Jul	353,359
	Aug	978,641
	Sep	744,476
	Oct	732,866
	Nov	625,334
	Dec	461,980
2016	Jan	638,408
	Feb	320,306
	Mar	462,449
	Apr	506,207
	May	230,349
	Jun	121,031
	Jul	1,109,196
	Aug	1,219,034
	Sep	610,035
	Oct	1,497,732
	Nov	301,647

Misconfiguration: Systems that are misconfigured, particularly with respect to the source IP, can generate packets with source IPs not found within the allocated network block.

Malformed Packets: Packets that are malformed or damaged in communication can see the source IP modified to an IP which lies outside of the allocated IP block.

Spoofed Packets: These are packets that are intentionally crafted with a source IP outside of the network block in order to masquerade as another system to act on behalf of it, or to hide the original system's actions.

This processor module was configured to monitor NetFlows sourced from the school's network and destined to external network. This means that network flows were only monitored in one direction. This should be kept in mind when reviewing these results.

Table 7.18 shows the results found for each month in terms of the number of network flows containing source IPs not belonging to the 192.0.2.0/24 subnet block, and sourced from the school networks. It was found that 73.27% of all anomalous source IPs were due to broadcasts or Local Internet Registry (LIR) traffic. Top contributors to this traffic can be viewed in Table 7.19.

What is particularly interesting about these records in Table 7.19 is that each of the sources were

Table 7.19: Top Source IP Anomalies Detected

Src IP	Dst IP	Src Owner	Traffic Type	Count
87.247.5.126	87.247.21.224	2DAY Telecom LLP	LIR	18,697
86.228.245.237	86.228.245.237	Orange S.A.	LIR	18,644
87.158.65.4	87.158.80.219	Deutsche Telekom	LIR	18,643
86.37.207.106	86.37.223.204	Qatar Foundation for Education	LIR	18,620
87.193.31.2	0.6.223.0	QC Internet Services	LIR	18,540

Table 7.20: Top Source IP Anomalies Detected (Broadcast and LIRs filtered out)

Src IP	Dst IP	Src Owner	IP Description	Detection Count
192.168.200.20	239.255.255.100	IANA	Private Network	14,892
198.51.100.218	Multiple Hosts	Internet Solutions	ISP	950
198.51.100.220	Multiple Hosts	Internet Solutions	ISP	23
-	Other	-	-	< 5

only ever destined to a single destination IP. This was further investigated and it was found that the first 1,021 occurrences of broadcast group or LIR source IPs were only ever destined to a single IP. This IP in each case was unique, and belonged to either a broadcast group, or LIR.

Removing LIRs and broadcast groups from the results gathered, one will find 3 points of interest and are shown in summary by Table 7.20. The first point to notice in this result subset is that any anomalous source IP with a detection count less than 5 was shown to only contain ICMP traffic. Due to the manner in which the network flows were captured in this dataset, the contents of these ICMP messages are unknown. However, there was never more than 5 packets observed with an average packet size of 178 bytes.

The next point that will be detailed is the existence of 192.168.0.0/16[27] traffic on the school networks. The IP 192.168.200.20 was the only IP used in the 192.168.0.0/16 subnet block and it destined all of its traffic to 239.255.255.100. This destination IP belongs to the subnet block 224.0.0.0/4 which is reserved for multicasting by IANA Internet Assigned Numbers Authority (2017). Upon further investigation into the device generating this network traffic, it was found that a misconfigured network switch was reporting to the the specified multicast group. This was a useful result generated by this processor module.

The last point to note in Table 7.20 is the two IPs owned by the same ISP[28] and communicating to multiple IPs. Upon further investigation of these sources it was found that destination ports

[27] Reserved by IANA for Private-Use.

[28] The original IPs were blinded to TEST-NET-2.

Table 7.21: Vulnerability Fingerprints Detected by Real Use Case

Year	Month	Detection Count
2015	Jul	8,510
	Aug	21,778
	Sep	22,145
	Oct	23,812
	Nov	12,949
	Dec	21,394
2016	Jan	21,396
	Feb	12,095
	Mar	8,404
	Apr	13,943
	May	6,781
	Jun	8,368
	Jul	19,278
	Aug	21,336
	Sep	10,268
	Oct	16,169
	Nov	5,337

mostly consisted of 53/UDP (DNS), 80/TCP (HTTP) and 443/TCP (HTTPS). This was most likely due to outage onto the SANReN network at some point and this was a backup connection to ensure some degree of Internet access was maintained on the schools' networks.

7.9 Vulnerability Fingerprint Detection

Before diving into the results presented in this section, one must first take note of a few characteristics of exploits that exist on the Internet[29]. It is a very difficult task to clamp down on a general class of exploitation as, one has to bear in mind a few points. The first and most obvious is the version of the exploit being used, or the version of software the attacker is attempting to exploit. A change in these would in most cases cause a change in packet sizes and possibly order of flows. This would cause the NetFlow records to vary from the original fingerprint. It is also common that multiple versions of an exploit exist live on the Internet at a given point, for example consider the Conficker worm's history (Irwin, 2011; Microsoft, 2009). This is common practice, as cybercrime is often compared to a warfare, where each side of the war is

[29] In the wild so to speak.

continuously trying to counteract each others advancements, and so naturally characteristics in exploits and defence mechanisms change.

Secondly, there exist exploits that simply act as carriers for a variable payload. This can be thought of as a two phase effort, where each phase is disjoint. The exploit seeks to simply gain access to a system and upload a payload, where as the payload actually performs the malicious activity; be it stealing information, deleting something, etc. Depending on what an attacker's end goal is, the payload can vary vastly for the use of a single exploit (Khrer *et al.*, 2014). This has to be considered when fingerprinting an exploit, as the transfer of the payload makes up part of the exploit's fingerprint (particularly relating to the byte and packet count).

Lastly, and more subtly, one has to consider the language of the region being exploited. If one simply changes the text presented to a potential target in an exploit from one language to another, without actually changing any functionality of the exploit, this alone could change the size of the exploit and thus the fingerprint. However, this would not change ordering of the NetFlow records.

These are all factors that need to be considered when creating a fingerprint detector, and is the reason why detection rates for small sets of fingerprints are usually low (Cox *et al.*, 2006). This approach should not be discarded though, as it can still be used to fingerprint unmodifiable parts of an exploit. For example, if one considers the *ntp_mon_list* (Czyz *et al.*, 2014) exploit fingerprint in Table 6.18, one will notice that this exploit cannot be modified. It has to make a monitor list request to an NTP server which takes a specific shape and form. It is a DDoS attack, so seeing a large amount of these requests in a short period of time means that one can safely assume that this form of DDoS attack is occurring.

Leading into the results one must note that the fingerprints used in this module were the same generated in Section 6.8. The number of exploits detected can be viewed in Table 7.21.

Referring to Table 7.22 one can see a break down of how many times each exploit was detected over the entirety of the 17 months. Of all the exploit fingerprints detected, all but 1 was due to the *ntp_mon_list* exploit. This other vulnerability was a *samba_usermap_script* exploit and it occurred in July of 2015. This resultant singular detected fingerprint, other than *ntp_mon_list*, does not come as a surprise due to the reasons discussed above.

As NTP is also UDP-based, this means that there is no need for an initial handshake, unlike what was observed in the other exploit fingerprints created in Section 6.8.4. The *ntp_mon_list* exploit makes use of a single packet that takes the form of one of three distinct unmodifiable packets. This makes the *ntp_mon_list* exploit used to create an amplification attack very easy

Table 7.22: Count of Each Vulnerability Fingerprint Detected in Dataset

Exploit	Count
ms08_067_shell	0
ms08_067_vnc	0
java_rmi_server	0
distcc_exec	0
samba_symlink_traversal	0
samba_usermap_script	1
unreal_ircd_3281	0
ntp_mon_list	253,962

Table 7.23: Breakdown of Top 10 *ntp_mon_list* Exploit Source IPs

Src IP	Open Ports	Detection Count
41.73.42.10	22, 81	38,183
196.4.160.4	123	27,492
41.73.42.22	-	25,239
185.94.111.1	-	19,782
146.231.129.86	123	10,469
41.79.80.34	80	10,443
134.147.203.115	22, 53, 80	10,402
208.110.72.90	22, 80, 443	7,385
167.114.85.120	22	5,833
104.255.69.6	80, 137, 445, 3389, 5985	5,072

to fingerprint and detect using that fingerprint (Li *et al.*, 2015). All of the other exploits fingerprinted in Section 6.8.4 also upload a payload onto the target. This allows for variation in the fingerprint generated by the exploits, and thus is much harder to detect.

Table 7.23 shows the top 10 sources of the *ntp_mon_list* exploit targeted at the school networks (192.0.2.0/24), broken down as to how many attempts each made over the 17 month monitoring period and what ports they had open [30] at the end of the monitoring period. Although it is very possible that IPs have changed hosts during the entirety of this run, it is interesting to see 41.73.42.22 and 185.94.111.1 having both reported no open ports. This means that the hosts are most likely offline or the IPs are unallocated. This could also suggest the use of these IPs as spoofed IPs when performing the *ntp_mon_list* exploit. There is however not enough information in the NetFlow logs to draw a complete conclusion about this theory.

Investigating into the original NetFlow logs where *ntp_mon_list* exploits were detected, these

[30] https://www.shoadan.io

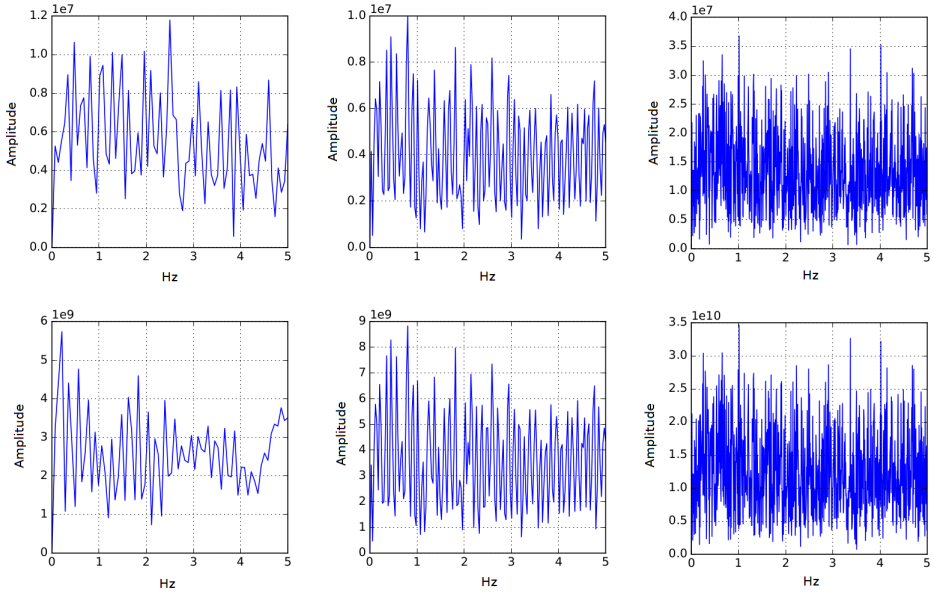

Figure 7.7: Fast Fourier Transform of 24 Hour Period (4th October 2016)

detected logs do follow the fingerprint in which this module was expected to detect. This fact led this testing to the conclusion that this processor module worked, although accuracy is limited without major refinements. One such refinement would be focussing fingerprints more on the immutable parts of an exploit, or exploits that are completely immutable like *ntp_mon_list*.

7.10 Fourier Analysis

Poor results from this FFT processor module were expected from the results found in initial testing in Section 6.3. To re-iterate: Fourier analysis is not usable in this case because Fourier transforms are time domain sensitive, and converting network traffic to netflow records results in the loss of time domain detail, to the point that the data is no longer usable by Fourier analysis techniques.

To better show the persistence of loss of accuracy, one can view a screenshot generated by the FFT processor module during its runtime in Figure 7.7. The information shown in the screenshot follows the same layout as in Section 6.3, and in it there are no frequencies that, when mapped back onto the original 17 month dataset, reveal any specific characteristics of the dataset. Furthermore, screenshots taken by this module directly contradict each others findings in peaks appearing for a frequency in the 24 hour period window at one point of time, not

appearing in a screenshot 5 minutes later. Again, as in Section 6.3 initial testing, no usable information was found to be ascertainable from this processor module.

7.11 Summary

This chapter serves to compliment the findings of tests performed on many of the modules found in **Chapter 6**, through providing testing results generated from NetFlow records produced from real-world network traffic. **Section 7.1** kicks off this chapter by outlining what the recorded 17 months of data looks like on the surface, and gives the basic structure of the network in question.

Continuing from this point, this chapter showed how Bolvedere satisfies its goal of scalability through its ability to be distributed over multiple hosts, or run a on single host, and moreover handle all records introduced to it from this dataset in **Section 7.2**. This was followed by a summary of the time taken for each module to process a record which was calculated in **Section 7.3** and theorized throughputs for the network in which the dataset was collected is calculated.

A more detailed look into Mirai and some events that occurred during the recording of the school network dataset was summarized in **Section 7.4**. Following this, **Section 7.5** discusses results collected from the neural network DDoS detection module. Port related analysis is performed on the recorded dataset in **Sections 7.6** and **7.7**. Results for source IP anomaly detection is discussed in **Section 7.8** and results around vulnerability exploitation detection is detailed in **Section 7.9**. Finally this chapter draws to a close with **Section 7.10** describing the failings of the Fourier analysis module.

8

Hardware Acceleration

AFTER the completion of functional and real-world testing in **Chapters 6** and **7**, Bolvedere was shown to function as intended, supporting 4 of the 5 goals set out in **Section 1.2**. This supports the fact that the collector subsystem in Bolvedere functions correctly, as, if it didn't, the knock-on effect would cause failures in logically subsequent subsystems within Bolvedere. With this fact in mind, this research can now focus on the discussions in **Chapters 3** and **4** around hardware acceleration of Bolvedere components, and aim to fulfil the final goal set out in **Section 1.2**.

This chapter deals with the primary design considerations and implementation of a hardware-accelerated collector for the Bolvedere system. In **Section 8.1**, one can find a summary of the existing software implementation of Bolvedere, why one would want to hardware accelerate this specific subsystem, and how one would go about achieving this. **Section 8.2** approaches the problem of operating the dedicated hardware at the speed of the network interface to which it is connected to, to remove the possibility of it being a bottleneck in the system. A functional block overview is modelled in **Section 8.3** to show how data flows within this hardware design, before discussing reception and transmission of network traffic in **Sections 8.4** and **8.5** respectively. The method in which packet discernment occurs in this hardware implementation of the

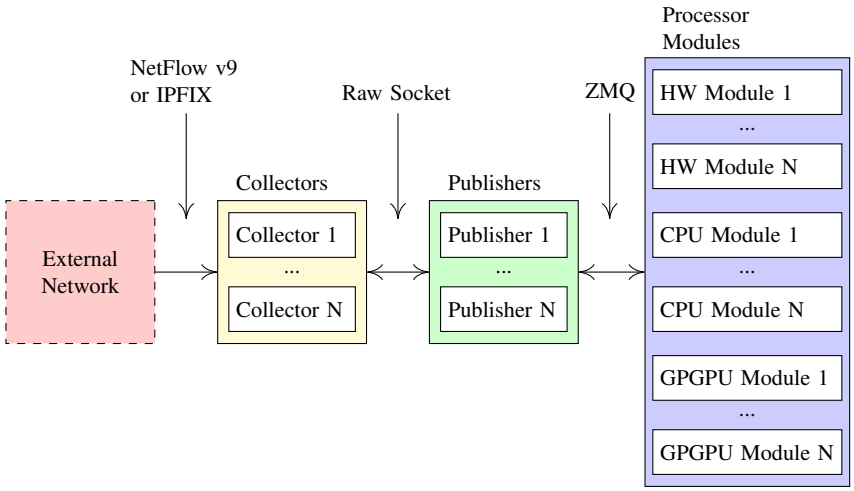

Figure 8.1: Marked Up High-Level System Overview

collector is then discussed in **Section 8.6**, before looking into how template and data records are processed in **Section 8.7**. Lastly, **Section 8.8** steps through a simple functional example of this hardware.

8.1 Design and Implementation

This chapter implements a physical replacement for the collector subsystem of Bolvedere[1]. To better understand where this subsystem fits within Bolvedere, one can refer to Figure 8.1[2]. To revise the logic flow of Bolvedere: Bolvedere is a system that is built on the NetFlow protocol in order to discern anomalous network flows on a network. The flow of logic through the Bolvedere system begins at the collector. This subsystem deals with collecting and recording NetFlow templates for their later use in discerning NetFlow data records. These discerned NetFlow data records are then filtered and reordered into a format that the publisher requires for optimal sequential processing. This configuration is received by the collector from the publisher at the start of the collector's runtime.

Once these filtered and reordered NetFlow data records arrive at the publisher, which is based on the sequential x86 and x64 instruction sets (Intel Corperation, 2016), the publisher can run through these data records from top to bottom, as the format of the records are now known to the publisher. This is due to the filtering and re-ordering performed by the collector.

[1]Please note that this work is previously published as part of Southern Africa Telecommunication Networks and Applications Conference 2015 and 2016 (Herbert and Irwin, 2015, 2016b).

[2]This is a derivative of Figure 4.1 that was placed in this text for convenience to the reader.

Figure 8.2: FPGA Product ID: XC3S500E-FGG320D

From this point, the publisher has two tasks. The first is to apply filters to the data records that are better processed sequentially, and the second is to publish the data to all subscribed processor modules (simply referred to as processors in Figure 4.1). This data is published using ZMQ (as discussed in Chapter 4).

The need for a replacement option for the collector of Bolvedere arises when considering scalability and future-proofing of this system. As NetFlow version 9 data records require the use of data templates in order to discern what the data records mean, overheads are introduced in a sequential processor system. This is due to the sequential architecture-based system not having instructions suited to the task of looking up what a field represents within a data record, and how many bytes that field is (as field size can vary) from the stored template record that discerns the relevant data record.

When considering that each subsystem of Bolvedere is selected from tools that are best suited for the task, using a tool which all these subsystems rely on within Bolvedere that is ineffective will render the optimization of every other subsystem moot. For this reason, it was decided to develop the collector as a hardware-based stream processor for the discernment, filtering and re-ordering of NetFlow data records for the publisher subsystem. This would alleviate the collector's bottleneck at scale by removing the overheads a sequential processor would suffer in performing the same task, as dedicated hardware instructions for this task would now exist.

8.2 Network Link Speed versus FPGA Clock Speed

The FPGA used in this implementation was the XC3S500E-FGG320D (Xilinx, 2013) of the Spartan 3 family with speed grade 4C; this chip is more commonly referred to as the Spartan 3E. This chip has a pin-to-pin time of 3.46 ns (Xilinx, 2013). This is the time taken from introducing an input to a physical pin on the FPGA package, traversing the logic within the FPGA, and then outputting the result on another separate physical pin on the FPGA package.

This was found to be only 13.584% slower than the more expensive[3] Spartan 6 family with a pin-to-pin time of 2.99 ns (Xilinx, 2011). When comparing the hardware emulation space provided by both the Spartan 3 and Spartan 6 family, and both were ample for this project, so cost was considered as a major factor in selecting a device (Xilinx, 2011, 2013).

The maximum frequency at which the selected Spartan 3E FPGA can drive its pins is 289.017 Mhz. The logic used to achieve this is a simple connection between the input pin's input buffer and the output pin's output buffer; the simplest functional logic within an FPGA. If one were to make this logic more complex, the time taken for a signal to propagate from an input pin to an output pin would increase. This is because it takes time for electrons to propagate through logic gates within an FPGA, and so one must account for this.

This hardware implementation introduces the hierarchical structure of the 5-layer network stack. Introduction of this complex logic structure will lead to an increase of time taken for input data to this system to be processed and output to a connected device. If the time taken between processing input exceeds that of the time before the a new input arrives for processing, this will lead to input to this hardware not being processed correctly. This will cause a knock-on effect that will cause all logically subsequent subsystems in Bolvedere to produce incorrect results. For this reason, this implementation needs to make sure that input can be dealt with before new input is introduced into this hardware.

To do this one must first consider what hardware is introducing new data into this system. This is the network PHY (Arregoces and Portolani, 2003) and when considering the clocking speed of this input, one needs to consider the rate at which whole bytes are ready to be interacted with. This consideration has to be made due to the way in which the two networking technologies considered PHYs interact with other hardware. The first technology considered was 100 Mbps Ethernet. This typically uses the Media-Independent Interface (MII) (IEEE Standards Association, 2002) protocol which makes use of a nibble[4] wide transmit bus, and a nibble wide receive bus. In order to meet the network rate of 100 Mbps, this protocol requires nibbles to be transmitted at a 25 Mhz clock rate.

This means that in receiving and sending, a data byte[5] is only ready for processing every second clock cycle of a device using the MII protocol. This means that an FPGA clocked at 12.5 Mhz would be able to receive every byte sent over a 100 Mbps network link.

A 1 Gbps link makes use of an update to the MII protocol, named the Gigabit Media-Independent Interface (GMII) (IEEE 802.3 Working Group, 2000). The key differences between MII and

[3] At the time of writing.
[4] A nibble is 4 bits.
[5] Two nibbles or 8 bits.

Listing 8.1: Synthesis Post-Map Static Timings for Spartan 3 Family FPGA

```
1  ...
2  Derived Constraint Report
3  Derived Constraints for TS_fpga_clk
4  +------------+-------------+---------------+--- ...
5  |            |   Period    | Actual Period |
6  | Constraint | Requirement +---------------+--- ...
7  |            |             |    Direct     |
8  +------------+-------------+---------------+--- ...
9  |fpga_clk    |     20.000ns|        6.000ns|
10 | CLK0_BUF   |     20.000ns|       10.179ns|
11 | CLKDV_BUF  |     40.000ns|       10.596ns|
12 +------------+-------------+---------------+--- ...
13
14 All constraints were met.
15 ...
```

GMII are that GMII uses a byte wide transmit bus, a byte wide receive bus, and is clocked at 125 Mhz to achieve the throughput rate required by a 1 Gbps network link. Although an FPGA can achieve clocking of simple logic at this rate, the logic required to describe how the 5-layer network stack works, as well as use of the payload of these network packets, impedes the maximal clock rate of the FPGA used.

To better understand this, one can refer to Listing 8.1. This listing shows a snippet of the output produced during hardware synthesis for the interface to the network PHY. This snippet refers specifically to the minimum clock period required to ensure proper execution of all logic within the FPGA for this interface. The “Actual Period” shows that, to traverse the longest logic path in the FPGA used for this interface, it would take 10.596 ns.

This means that the logic for the network PHY can be processed without any problems arising due to input timings when dealing with a 100 Mbps network link. The reason for this is that data at 100 Mbps is clocked in at 25 Mhz which gives 40 ns between each input. This is more time than required by the FPGA to process the incoming data.

When using this result to consider a 1 Gbps PHY, this 10.596 ns is not adequate to support this interface. At 10.596 ns the maximum frequency one can clock this FPGA at is 94.375 Mhz. This is not enough to negotiate with a 1 Gbps network link which requires 125 Mhz. Resynthesizing this design for a Spartan 6 FPGA leads to the results found in Listing 8.2. The time required to complete the logic traversal of the PHY is 9.877 ns. This means that the maximum speed that one can run the Spartan 6 FPGA at with this PHY logic is 101.245 Mhz. This is again too slow to negotiate with a 1 Gbps network link.

Listing 8.2: Synthesis Post-Map Static Timings for Spartan 6 Family FPGA

```
...
Derived Constraint Report
Derived Constraints for TS_fpga_clk
+------------+-------------+---------------+--- ...
|            |   Period    | Actual Period |
| Constraint | Requirement +---------------+--- ...
|            |             |    Direct     |
+------------+-------------+---------------+--- ...
|fpga_clk    |     20.000ns|        6.000ns|
| CLK0_BUF   |     20.000ns|        9.189ns|
| CLKDV_BUF  |     40.000ns|        9.877ns|
+------------+-------------+---------------+--- ...

All constraints were met.
...
```

When considering 100 Mbps link rates, the requirements of the FPGA are made more favourable by the fact that a byte only has to be dealt with every 2 clock cycles. This means that the MII protocol only requires a byte to be dealt with every 80 ns, meaning that the clocking rate for whole bytes coming in and out of the FPGA can be set at 12.5 Mhz.

To re-enforce the importance of this implementation's stream-based design, if the FPGA system clock is based off of and can support the network link speed, then - because it simply streams the network traffic through itself - it will work as intended irrespective of the link speed used. This means that if one were to clock down the network interface to 10 Mbps, this device would scale down its system clock speed with the network interface, and proceed to work at that clock rate without need for modification or resynthesis of the hardware within the FPGA. Furthermore, if one were to fabricate this logic into an ASIC that supported faster physical logical gating speeds, and thus faster system clock speeds, when attached to a faster network than 100 Mbps it would simply scale up to the clocking rate of that network. This functionality also serves to future-proof this implementation.

8.2.1 Keeping State While Streaming

The point of streaming network traffic is raised on more than one occasion in this text, and the reason why this is possible will now be discussed. The design and implementation of the logic within the FPGA is based on the Register-Transfer Level (RTL) design abstraction (Bening and Foster, 2001). This form of design allows for easier modelling of this system in the form of states. The logic transfer between these states depends on the task the logic is attempting to

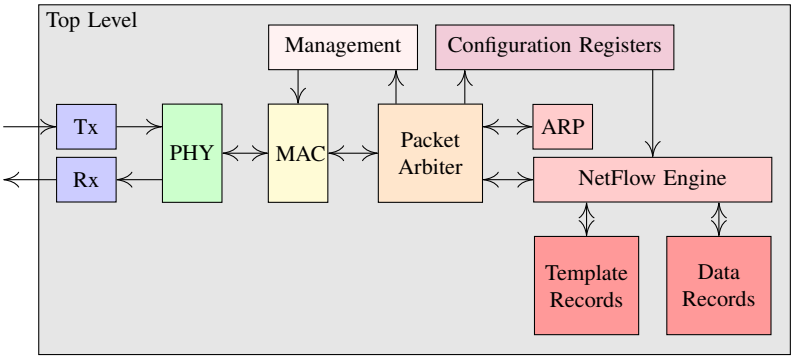

Figure 8.3: High-Level Abstraction of Hardware Functional Blocks and Connections

complete at any given point of execution. Given that the 5-layer network stack (Kozierok, 2005) and NetFlow protocol packets are assembled as layers, one can work through network packets in a stateful manner. This is because the next part of information within a network packet is known, based on a previous part of that packet, and so a state transfer can be made.

In design, this means that incoming network traffic can be streamed, and that the FPGA does not need to buffer any incoming network traffic. Instead, it can look at the byte that has just been received, and transfer state without the need to remember something before that byte, or know something from a byte that has yet to be received. This means that there is a lower memory footprint, as buffering is only required on outbound packets and for the configuration received from the publisher. Furthermore, this device can execute regardless of network link speed, as long as the time for logic to propagate through the devices hardware is less than that of a system clock period.

8.3 Very High-Level Hardware Functional Block Overview

This section will discuss at a high level the life of a network packet through this hardware implementation. How each major functional block achieves its task will be discussed in later sections. Referring to Figure 8.3, one will notice that receipt and transmission of packets are broken into two separate modules. These modules run in parallel to allow support of full-duplex connections (Bartlett *et al.*, 1969). This simply means that this device can receive and transmit network traffic at the same time. The functional block named *PHY* acts as a general manager for the *Rx* and *Tx* functional blocks, and ensures data is streamed to these blocks in a sane manner while maintaining link speed synchronisation.

Once a packet is received it is passed to the *MAC* functional block. This block ensures that

that packets are addressed to and from the device's physical address. This functional block is also tasked with discerning whether an arriving packet is an Address Resolution Protocol (ARP) (Plummer, 1982) packet or an IPv4 (Postel, 1981a) packet. As the use of NetFlow in Bolvedere is primarily focussed at IPv4, if the packet does not satisfy the conditions of use of either the ARP or IPv4 protocols, the packet is discarded.

Packets successfully received and passed by the *MAC* functional block are then given to the *Packet Arbiter*. This functional block is tasked with identification of ARP type, configuration reception and NetFlow packets. Once the type of packet is identified, this module passes on the payload of the packet to the relevant attached functional block. Also, it is note worthy that as NetFlow in Bolvedere is only dealt with via UDP (Postel, 1980), any packet that is IPv4 but not of type UDP at the transport layer is explicitly ignored.

Configuration comes in two flavours in this hardware implementation, the first being *Management* and the second being *Configuration Registers*. The *Management* functional block deals with everything relating to the network. This includes which publisher the device is talking to, the address of that publisher, as well as the IP, port and hardware address of this hardware device. The *Configuration Registers* exist to deal with the re-ordering and filtering of NetFlow data records. Simply, this is where the configuration received from the publisher is stored and used to re-order and filter NetFlow data records into the form that the publisher expects for optimal sequential processing.

The *ARP* functional block does little more than generate appropriate ARP responses for the *Packet Arbiter* to transmit. This is handled through a series of flags that are raised to the *ARP* block by the Packet Arbiter.

The last three functional blocks deal with the NetFlow protocol. The *NetFlow Engine* simply discerns whether a flow set within the NetFlow packet contains template records or data records, and then passes that part of the payload to the relevant functional block. After the relevant functional block has finished processing its chunk of the payload, the *NetFlow Engine* checks whether the entire packet has been processed. If it has, processing of the NetFlow packet is complete, if not the next flow set is discerned and passed to the relevant block for continued processing of the NetFlow packet's payload.

Processed template records are stored and used in conjunction with the configuration stored in the *Configuration Registers* to process data records. Once enough data records have been processed; the resultant reordered and filtered records are transmitted to the publisher.

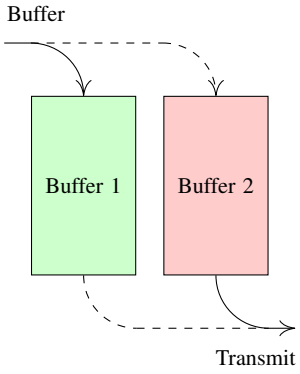

Figure 8.4: Double Buffered Transmission Buffers

8.4 Receiving Network Traffic

Network traffic received is assembled into bytes at the *Rx* functional block, before being handed on to the next connected block (the *PHY*). This is done by storing the first nibble off the wire, and appending the second nibble to the first when the second is received, to form the full byte. This byte is then presented to the *PHY* for processing, which ensures the network traffic follows the Ethernet (Healey, 1983) standard. This includes ensuring correct inter-frame gap, intact preamble and correct frame checksums.

8.5 Transmitting Network Traffic

Transmission of network traffic, like receiving, is done a nibble at a time; this concept was covered in Section 8.4. The main consideration that was made in the *Tx* functional block was that of fully saturating the available link's bandwidth. It was quickly realised that transmitting a packet without knowing its length, or what was in the payload, would require massive overheads in terms of the checksum and length fields within both the IPv4 and UDP headers. Thus, the use of a buffer was chosen to mitigate this overhead.

This approach, however, introduced two new issues. The first was the need for a memory structure that could support a network packet, and the second was that the buffer could not be modified while writing it out to the network link. This meant that the use of a single buffer to transmit network traffic would result in, at best, only half the link's bandwidth being utilized for transmission. These two issues were solved through implementing a double buffer for network

transmission using First In, First Out (FIFO) (Dan and Towsley, 1990) memory structures, which easily fit within the Spartan 3E's block RAM.

Using a double buffer meant that a new network packet could be written to the one buffer while the other buffer is being transmitted. Once the transmitting buffer is finished transmitting, the buffer that was just written to can transmit, while the buffer that just transmitted gets buffered with a new packet. This form of buffering means that the network link can be continuously be written to. This idea is visually illustrated in Figure 8.4.

The use of a FIFO leads to an issue arising with the IPv4 and UDP checksums in that both were unknown until the entire payload was buffered into the FIFO. This was overcome through creation of special checksum registers within each buffer, which were loaded with these relevant checksums when the packet is entirely buffered.

At transmission time, the fact that the IPv4 and UDP checksums are always located in the same byte indexes within the FIFO was used to pause reading from the FIFO momentarily at these times, and instead transmit the bytes from the relevant checksum registers.

8.6 Packet Discernment and Future-Proofing

Packet discernment is handled by multiple functional blocks at different levels of this hardware's implementation (this has been discussed previously in section 8.3). The reason for this design decision was twofold; the first was to keep functional blocks dealing with only the parts that they are concerned with. This was due to the cycle of development, as if a block did not work it could be replaced without the need to modify other blocks.

The second reason was that this modularity allowed one to take functional blocks out of this system's design, and place them into other systems, given that the inputs to the functional block follows what the block expects. This led design of this hardware in a manner that separated the *NetFlow Engine* (referring back to Figure 8.3) from the networking components of the device. In short, development in this manner meant that the *NetFlow Engine* hardware could be taken and placed into other System on Chip (SoC) designs with little overhead, giving those SoCs the same capability as this hardware.

8.7 NetFlow Template and Data Record Processing

The processing of NetFlow records within this system, and where this processing occurs, was discussed in Chapter 4. This section will now discuss in further detail the manner in which these NetFlow packets are processed in this hardware. A NetFlow packet is made up of a header and one or more flow sets. Each of these flow sets contain their own header, and from this the length of the flow set, as well as whether the flow set contains template records or data records, is presented. If one has the template for given a data record flow set, the number of records within that flow set can be worked out. Each template record within a template FlowSet contains its own header stating how many fields are within the template record. A template field is made up of an ID and byte count, each recorded as 16 bit unsigned integers. This means that the length of a template record is 4 bytes multiplied by the number of fields stated in the template record header.

The *NetFlow Engine* functional block (referring back to Figure 8.3) handles discernment of whether a packet is actually a NetFlow packet, and if it is, the discernment of the NetFlow record types. This record type discernment is done on a flow set to flow set basis. Discernment of a NetFlow template flow set is less complex than that of data record discernment, due to the requirement of the template to make sense of a data record.

If a template flow set is detected, that flow set is then streamed to the *Template Records* functional block for processing. This functional block has a finite number of templates it can store, and so template ejection also has to be considered to make space for newer templates. This functional block also has to deal with the possible event of a duplicate template ID being received (a new template with the same template ID as one already stored).

Dealing with template ejection first, this is handled via a Least Recently Used (LRU) (Lee *et al.*, 2001) algorithm. When a new NetFlow template record arrives and there is no space for it, the template which hasn't been used for the longest time is ejected and replaced with this new template record. When receiving a duplicate template record ID, this template is considered as a template record update according to the NetFlow protocol (Claise, 2004). This is handled by writing over the existing record of the same ID with the new duplicate record's field data.

NetFlow data records are only processable if there exists a NetFlow template record of that data record's type. Relating a data record to template record is performed on a flow set basis, and is done through use of the flow set ID. Only one kind of data record can exist within a data record flow set, and the flow set ID determines this type. If the flow set's ID exists in the template IDs stored in this hardware at runtime, then that template ID is used to discern all data records in

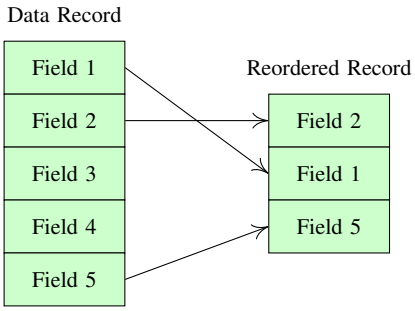

Figure 8.5: Example Filter and Reorder of NetFlow Fields in a Data Record

the current data record flow set. If it does not exist, the length of the flow set is then used to skip over and ignore that flow set. Processing of the next flow set continues from here if the end of the packet has not been reached.

8.8 A Simple Functional Example at Runtime

Data records are filtered and reordered according the the *Configuration Register's* functional block, a simple example of which is provided in Figure 8.5. This configuration is collected from the publisher at boot time of the device and written into the *Configuration Registers*. These registers define which data record fields are wanted by the publisher, and the order in which the publisher wants these records. This is done by writing to a reordered buffer in a location specified by the configuration when a field wanted by the publisher is found in a data record. Once that data record is processed, the reordered buffer is pushed into the buffer FIFO for passing onto the publisher via the network link.

8.9 Summary

In this chapter, an alternate implementation to Bolvedere's collector is proposed. This implementation is focused on an in-hardware solution, leveraging the strength of the FPGA architecture. As a whole, this implementation takes the weaknesses of the CPU-based sequential architecture discussed in **Chapters 3** and **4** and implements specific hardware-based functionality to mitigate these CPU-based overheads.

As the goal of the collector is known by this point in this document, this chapter starts off with **Section 8.1** focusing on why such a hardware implementation of the collector is needed, rather

than what the collector subsystem aims to achieve as a whole. The hardware implementation's ability to keep up with network link speeds is then shown in **Section 8.2**. Once the FPGA timings are shown to meet specification, **Section 8.3** presents an overview of the design of the required hardware, as well as a high-level overview of how each functional block will be connected inside the FPGA. The method of data reception and transmission is then detailed in **Sections 8.4** and **8.5** respectively, and once this concept is understood, the method in which this custom hardware discerns NetFlow packets is brought to light in **Section 8.6**.

Once a packet is identified as a NetFlow packet that is addressed to this hardware-based collector, further discernment is required in order to tell what kind of records the packet contains (template or data). If it is a template record, it is simply stored by the FPGA for later use in discerning data records. If it is a data record the relevant template needs to be fetched, and the entire data record filtered and re-ordered as the publisher's requirements. This functional block implementation is presented in **Section 8.7** and tested in the closing of this chapter in **Section 8.8**.

Following this chapter, **Chapter 9** performs testing against the existing software implemented collector, to show improvement gained through a hardware-accelerated implementation. The tests performed in this chapter show correct functionality, the hardware collector's ability to stream at link speed, and a direct throughput comparison to the software collector implementation.

9

Hardware Equivalence Testing

THIS implementation sought to reimplement the Bolvedere collector subsystem in hardware as a stream processor. This means that this hardware implementation should be able to produce correct results at the full bandwidth the network link speed provided. In this chapter it was also decided that it was worth showing the performance increase gained by the publisher when it processes re-ordered NetFlow data records in a known order. This reasoning led to 3 major tests, which are detailed below

After this testing environment is defined in **Section 9.1**, the first test performed in **Section 9.2** will indicate whether results produced by this hardware implementation are correct, and thus equivalent to the its software counterpart; answering the question of whether this works. The second test in **Section 9.3** will question the ability of the stream processor design, and will test whether this device can saturate an entire link's bandwidth. The third and final test performed in **Section 9.4** does not relate to the device, but will instead show the performance gain by the publisher when processing data record fields in a known order as per configuration given to the collector.

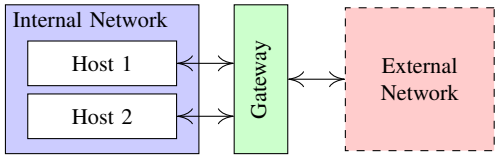

Figure 9.1: Physical Configuration of NetFlow Monitored Simple Networks

9.1 Environment

Testing was performed at 100 Mbps link speed. Link speeds of 1 Gbps and above were not considered were not used due to limitations of the FPGA, as discussed in Section 8.2. Three simple networks of three physical hosts were created to generate network traffic. One system acted as a gateway to an external network, while the other two systems resided within that gateway's internal network; Figure 9.1 shows the example layout. These gateways ran Softflowd (Miller, 2016) and were configured to export all network flows generated to the hardware collector. The hardware collector in turn was configured to pass on its filtered and reordered results of these NetFlow data records to the publisher, which was configured to duplicate the received bytes to the terminal.

Network traffic is created by custom-written bots that execute on the hosts within the gateway's internal network. These bots work by sleeping for a set time before loading a given starting URL from the Internet, creating a list of all URL links on the returned page, and then choosing one, before restarting the process of sleeping and loading that URL. All systems used in testing were running Ubuntu Linux 14.04 LTS [1], and the version of Softflowd used by the gateways was 0.9.9.

9.2 Correct Filtering and Reordering of NetFlow

Three things need to be considered in order for the output produced by a Bolvedere collector to be correct:

- Only the fields required exist in the output.
- The fields required are placed in the correct place in the output.

[1] http://www.ubuntu.com/

Table 9.1: Softflowd's Recorded IPv4 Template Fields In Order

Field Name	Field description
IP_SRC_ADDR	IPv4 source address
IP_DST_ADDR	IPv4 destination address
LAST_SWITCHED	System uptime since the last packet of this flow was switched
FIRST_SWITCHED	System uptime since the first packet of this flow was switched
BYTES	Byte count
PKTS	Packet count
INPUT_SNMP	Input interface index
OUTPUT_SNMP	Output interface index
L4_SRC_PORT	TCP/UDP source port
L4_DST_PORT	TCP/UDP destination port
PROTOCOL	IP protocol used
TCP_FLAGS	Count of all flags seen for this flow
IP_PROTOCOL_VERSION	Version of the IP protocol used

Table 9.2: Kept and Reordered Fields from Hardware Collector

Field Name	Field description
BYTES	Byte count
PKTS	Packet count
IP_SRC_ADDR	IPv4 source address
MIN_PKT_LNGTH	Minimum packet length seen in a flow
IP_DST_ADDR	IPv4 destination address
MAX_PKT_LNGTH	Maximum packet length seen in a flow

- Fields that do not exist in the original data record have the correct place holder put in place, this being every bit set to 1.

To show that this hardware collector satisfies all three of these points, only a single test is necessary. Softflowd's template that describes IPv4 network flows is represented by the same 13 fields as in the order found in Chapter 7. For ease of reading, this list can be found in Table 9.1. For this test, the hardware was configured to filter and reorder the data records generated by Softflowd into the form found in Table 9.2.

This configuration will reorder the incoming Softflowd generated NetFlow data records, filter out unrequired data records, and utilise the default place holders for the minimum and maximum IP packet lengths, as the incoming data records do not contain these fields.

Listing 9.1: Reordered Data Record Output as Seen by Publisher

```
...
Packet Payload Hex Out:
00 00 03 FD 00 00 00 03  // Byte Length, Packet Count
9D 48 00 02 FF FF FF FF  // Source IP, Min Packet Length
68 10 41 CB FF FF FF FF  // Destination IP, Max Packet Length
...
Packet Payload Hex Out:
00 00 2B DF 00 00 00 70
9D 48 00 11 FF FF FF FF
C7 10 9C E6 FF FF FF FF
...
```

Table 9.3: Values of Reordered Fields from Bytes Received

Field	Result 1	Result 2
Byte Count	1021	11,231
Packet Count	3	112
Source IP	157.72.0.2	157.72.0.17
Minimum Packet Length	Place holder	Place holder
Destination IP	104.16.65.203	199.16.156.230
Maximum Packet Length	Place holder	Place holder

9.2.1 Results

Results were collected from known 200 NetFlow data records. A an extract of the output generated by the publisher showing the received payload bytes can be viewed in Listing 9.1. One can translate these results by first understanding that the bytes streamed out of the collector are in the order presented in Table 9.2. Furthermore, each field is 4 bytes in length. Given the bytes returned are in order and can be read from left to right, line by by line, this means that the first 4 bytes are the byte count, followed by the packet count, source IP, minimum packet length, destination IP and finally the maximum packet length.

When considering Listing 9.1, it is important to note that both minimum and maximum packet length fields take on the place-holder value 0xFFFFFFFF. This is due to the fact that Softflowd does not record these two fields. Following this, when the collector the collector attempts to draw these fields out of a NetFlow record received from Softflowd, it cannot find them, and so replaces both of them with the place-holder values 0xFFFFFFFF.

A conversion from these byte values to integers can be found in Table 9.3. Upon comparing the output generated by the publisher receiving the reordered NetFlow records from the collector, to

the original known 200 NetFlow data records, all were found to be correctly discerned, filtered and reordered.

9.3 Streaming at Link Speed

This test involves checking whether this device can process NetFlow data records at full link speed. Keeping track of whether a NetFlow log was missed or not was done by counting the number of logs received at the publisher. If one knows the number of data records transmitted to the collector, one simply needs to count how many processed records arrived at the publisher, whether they arrived in order and that all reordered NetFlow records arrived intact to determine a result for this test. This test made use of the same configuration as the test performed in Section 9.2.

Difficulties in this test arose when attempting to saturate the connected network link to the collector. The reason for this is that NetFlow data records are exported to the collector when a network flow ends or times out[2]. When a data record is exported is a very difficult event to time, and this is a task made even more difficult by trying to saturate a 100 Mbps link with these legitimate NetFlow records.

To solve this problem, 100 legitimate NetFlow packets were recorded and then randomly replayed onto the network link attached to the hardware collector, with no delay between packets. This process was used to replay 200,000 NetFlow packets to the hardware collector, and resultant reordered packets destined to the publisher were then counted by the publisher. Furthermore, the order in which the 200,000 NetFlow packets were transmitted in was recorded for later use in verification of results.

9.3.1 Results

This test resulted in the reception of 200,000 reordered NetFlow data record packets at the publisher. Output from the publisher can be viewed in Listing 9.2. The publisher for this test had packet counting enabled[3], and displays this value between '[' and ']' characters in its output. The snippet shown in Listing 9.2 shows two records taken from the output. The first output is selected randomly, while the second is specifically chosen and is the last packet received in the

[2]No network traffic is seen in the flow for a predefined period of time.
[3]This feature is turned off by default to save CPU resources.

Listing 9.2: Reordered Data Record Output as Seen by Publisher with Count Output Enabled

```
...
[147477] Packet Payload Hex Out:
00 00 5A B2 00 00 00 2F
9D 48 00 11 FF FF FF FF
C7 10 9C E6 FF FF FF FF
...
[200000] Packet Payload Hex Out:
00 00 13 0A 00 00 00 06
92 E7 58 04 FF FF FF FF
08 08 08 08 FF FF FF FF
...
```

test. This last packet is the 200,000th packet received by the publisher, and thus shows that all packets transmitted were processed by the collector and, after transmitting to the publisher, were received.

When observing the recorded random order in which the NetFlow packets were transmitted and comparing these results to the collector's configuration and the reordered NetFlow records generated by the collector, all reordered packets were found to be correct. Examples of these are shown in Listing 9.2 and follow the same translation method found in Section 9.2.

9.4 Hardware versus Software Processing Times

These tests show the total time for a Bolvedere system running a software collector and publisher to process a set of records, versus a Bolvedere system running a hardware collector and a software publisher.

Originally, the host's CPU running Bolvedere dealt with discernment of NetFlow data records by itself before distribution to processor modules. Given that there are two components involved in discernment of a data record (these being template and data), and the fact that a sequential processor has no method of viewing both of these data at the same time, it was noted that this would cause a bottleneck in Bolvedere as a whole.

This test was performed by introducing varying-sized duplicate datasets to the completely software-based Bolvedere system, and again to the hardware collector-based Bolvedere system, and timing how long it takes to process each. The time to process a NetFlow packet was timed from point of introduction to the collector, to time the publisher published the relevant records found in the NetFlow packet.

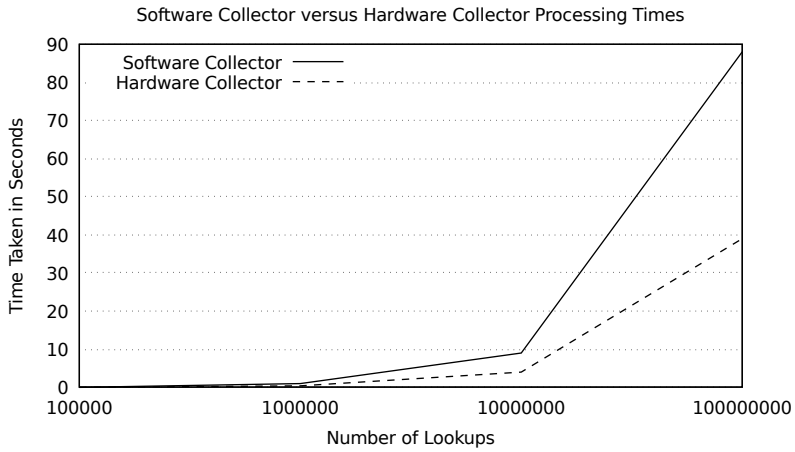

Figure 9.2: Time Taken for Sequential Processing of In-Order versus Out-of-Order NetFlow Data Records

9.4.1 Results

Figure 9.2 shows the time taken to process the varying sized datasets for both software and hardware collector-based Bolvedere implementations. These results strongly motivate the inclusion of an option to process NetFlow records on a hardware-based collector subsystem. This is highlighted in these results, which on average show the hardware collector-based Bolvedere system records taking 44.318% of the processing time of the software collector-based Bolvedere system.

This result is particularly interesting as hardware-accelerated records are processed more than twice as quickly as software processed records. This suggests that the introduction of overheads in software alone require 5% of all processing time when discerning and processing a field in a data record.

These results strongly motivate the use of a hardware-accelerated Bolvedere system. This being said, as both software- and hardware-based collectors produce the same results, depending on one's requirements, one should still have access to both options.

9.5 Summary

As this hardware implementation of the collector subsystem of Bolvedere is designed to be a replacement for the software implementation of it, this hardware implementation needed to be

shown as functionally equivalent to the software implementation of the collector. This chapter achieves this through first defining the environment in which all testing would be done in **Section 9.1**.

Once this testing environment was laid out, testing aimed to check that the ability to filter and reorder packets was equivalent to that of the software collector in **Section 9.2**; achieving this would also ensure that the hardware implementation was functionally correct. The next step was to check that the hardware implementation could keep up with the Ethernet link speed at which it synchronised in **Section 9.3**.

Once this was proven functional, the last step was to note the effect that replacement of the software-based collector would have on Bolvedere as a whole. These results close this chapter and are brought forth in **Section 9.4**, which showed that the use of this hardware implementation more than halves the time taken for Bolvedere's base to publish NetFlow records to modules subscribed to the publisher.

10

Conclusion

THIS research implemented and tested the Bolvedere system; a system that sought to optimise processing and analysis of network flows when using the NetFlow protocol. Bolvedere as a whole was tasked with the distribution and filtering of NetFlow records on an array of hardware configurations. Furthermore, Bolvedere was designed with modularity in mind, enabling its distributed processing mechanisms on parallel architectures.

Finally, hardware acceleration took place in order to provide hardware instructions for discerning and re-ordering of NetFlow data records on an FPGA-based architecture. This reduced the time taken to deliver a new record to processing modules in Bolvedere by 55.682%.

This chapter concludes this research by starting with a summary of this document to this point in **Section 10.1**. Following this summary, a look at the key points that Bolvedere aimed to achieve takes place in **Section 10.2**, which includes compatibility, accessibility, scalability and parallelization. The degree to which the goals outlined in **Chapter 1** are met are then taken into account in **Section 10.3**. The level to which real-world application is achieved by this research is discussed in **Section 10.4**. Finally, this chapter, and research, concludes with a summary of the novel contributions this research made to the field in **Section 10.5** and then a look into future extensions that can be made to this research in **Section 10.6**.

10.1 Document Summary

This document began with a description of the problem statement and outlines the method by which this research aimed to resolve this in **Chapter 1**. Following this, **Chapter 2** brought forth all the background information required for understanding key concepts and the context in which this research takes place. With this knowledge, **Chapter 3** and **Chapter 4** evaluated existing technology's strengths and weaknesses before designing Bolvedere in a manner that could best utilise these current technologies. This includes the ability to horizontally scale this implementation.

Chapter 5 proceeds to design the mechanisms of Bolvedere that would analyse the NetFlow records. This would serve to show that Bolvedere's base system[1] worked correctly and that the system as a whole could be used to analyse NetFlow data. Upon completion of implementation of Bolvedere, **Chapter 6** tested Bolvedere's base system and processor modules in a synthetic environment. This testing was performed in order to verify the correct functionality of Bolvedere.

Building on this synthetic testing, once Bolvedere was deemed correct in functionality **Chapter 7** introduced a real-world dataset consisting of 17 months of school network data. Although not extensive, the analysis of this dataset brought to light many interesting events during these 17 months, and showcased the ability of Bolvedere as a whole.

Upon conclusion of showing the usefulness of Bolvedere as a NetFlow processing and analysis platform, this research refocused its aim on maximising Bolvedere's NetFlow record throughput, and future-proofing its implementation in **Chapter 8**. This was achieved through identifying bottlenecks in Bolvedere's collector system, and hardware-accelerating it through the implementation of the collector in an FPGA.

To ensure that Bolvedere still functioned correctly, testing was performed to show equivalence between the previously implemented software collector and the newly implemented hardware collector in **Chapter 9**. This chapter concluded by benchmarking the hardware collector against the software collector.

[1] Consisting of collector and publisher.

10.2 Key Aspects

Key aspects of this research, which are extracted from this research's goals set out in Section 1.2, are listed below:

1. **Primary Aspects**

   (a) To evaluate the benefits of implementing a dedicated NetFlow processor, and the effects this will have on existing systems.

   (b) Ease of processor module development through code base choice and removal of need to rewrite the initial NetFlow ingress and distribution system (Bolvedere's base).

   (c) The ease of access to this system to allow for its implementation and continued use. This also includes the reuse of existing systems in order to make the adoption of Bolvedere cost-effective.

   (d) Scalability of Bolvedere is taken into account to allow for parallel computation of a task or many tasks that work on common data received by Bolvedere.

2. **Secondary Aspects**

   (a) Real-world application is taken into account, as this research was aimed at being a real-world implementable product.

   (b) Finally, optimisation of this system is taken into account through identification of bottlenecks and mitigation of them through design and implementation of dedicated FPGA-based hardware to perform the bottlenecked task.

This research, in its simplest form, was aimed at optimization of the logic flow within network flow analysis systems. Furthermore, the system developed by this research should be easily implementable, highly compatible and easily accessed. With this in mind, this research looked towards drawing robust conclusions from results yielded during testing, and tying these back to the goals set out at the beginning of this research.

10.2.1 Processor Module Development

With completion of Bolvedere's base system in both software (Chapter 3) and hardware-accelerated form (Chapter 8), and its development for compatibility with a well-supported data distribution platform, the Bolvedere system removes the need for one to write boilerplate code required

simply to get a NetFlow record to a processor module. The manner in which this performed is detailed in Section 4.3. The development of a new processor module for Bolvedere is as simple as importing ZMQ into the module, and subscribing it to the Bolvedere base system that is broadcasting the processed NetFlow data records that are relevant to the processor module in development.

On top of this is the fact that, as ZMQ supports over 30 languages (as detailed in Section 3.2.5), one can choose the best programming language for the task at hand. Extending this fact is that ZMQ is a well-defined and well-documented protocol. This allows one to develop a processor module in a language or on a platform not yet supported by ZMQ. This can be done through the use of sockets to connect to the Bolvedere base, and then performing communication based on ZMQ's documentation to interpret the received data.

In all, this makes for a very easy platform to develop, as the understanding of many complex underlying mechanisms of NetFlow can be omitted. This allows one to place their focus directly on the development of the processor module intended to analyse the relevant NetFlow record data.

10.2.2 Modularity, Scalability and Parallelism

Processor modules perform the bulk of NetFlow data record analysis in Bolvedere. As mentioned before, one can choose the best language for the task at hand. Because of this, one can explore any data analytic that a language supports. This strength is coupled with the manner in which Bolvedere executes. The Bolvedere base system executes disjointly to any of the processor modules listening to the output of the Bolvedere base. This means that modules can be removed and added to the system as a whole, without any affect on any Bolvedere base system, or on any other processor module in the system. This technique was used during both synthetic testing in Chapter 6 and real-world analysis in Chapter 7. This functionality also allows for removal and addition of Bolvedere base modules during execution of the system in its entirety.

Furthermore, this allows one to break up work-loads into more manageable chunks, or distribute processing of modules effectively; one can see this configuration being used directly in Section 7.2. For example, if a work-load is too large for a single module to process, this work-load can be broken down into smaller chunks based on subnets or port range. As Bolvedere is built on ZMQ, the same processor modules can be executed on separate physical hosts attached to the same network. This allows the processor modules to simply connect to the same Bolvedere base using the same ZMQ command that would be used over a network. This logic is also applicable

to Bolvedere base systems, as one can run multiple Bolvedere base systems on a network on which processor modules can choose which subset of these base systems to subscribe to.

On the other end of the scale, if the work-load is small enough that all resources are not used by a host in processing a processor module, one can run multiple processor modules on that host, making use of common hardware in order to achieve their tasks disjointly. This system design inherently allows for ease of scalability as well as parallelism. This was part of the initial design decisions made in Section 3.2 around Bolvedere's inter-system communication. This is due to each processor module executing in its own work space concurrently, and this allows for parallel operation if hardware support allows for it.

10.2.3 System Accessibility

System Accessibility in terms of this project refers to the ability to execute Bolvedere on a system with limited resources available for execution. Cost is also taken into consideration in this evaluation, as the cheaper something is to acquire, the better the adoption rate of that something is. With these two factors in mind, Ubuntu Linux was chosen as the operating system of choice for this implementation. This operating system is both free and easily installable, and is highly compatible with many hardware configurations dating back years in age. This choice was made during the design phase of Bolvedere in Chapter 3, and all testing performed in Chapters 6, 7 and 9 made use of this operating system. This allows for poorer demographics that cannot afford the latest systems to gain access to this research and execute it to some degree. This will increase the potential penetration rate of Bolvedere.

This design choice is two-fold. Maximising compatibility means that adoption of Bolvedere in existing infrastructures and systems is also a cost-effective task. In this case, Bolvedere can use already purchased and in place hosts that are running other systems in order to replace them, or run alongside them if the host's resources permit this. This also allows for ease of scalability in the future with the growth of networks and thus the increase of bandwidth that needs to be processed.

10.2.4 Hardware Acceleration

Hardware acceleration was introduced into this project in Chapter 8, in order to optimise the existing system and future-proof it. The entire hardware design is intended to synchronise and operate at the speed of the network interface to which it is connected, thus allowing the

hardware to scale seamlessly to the speed of the network it is connected to. This is shown to be the case in Section 9.3. Furthermore, this VHDL design can be bundled up into an Intellectual Property Core (IP Core) and included in other networking projects or SoCs. This means that future hardware can choose to natively support Bolvedere in hardware, thus enabling network equipment such as switches and routers to act as Bolvedere bases that processor modules can subscribe to.

It must be restated that this hardware is simply a clone of the functionality existent in the software implementation of the collector, and there is no loss of functionality by not having access to the hardware implementation of the collector. This was shown through using the software-implemented collector as a direct comparison in Chapter 9, and more specifically by the direct comparison of results in Section 9.4. If this was not true, no performance comparison could be justifiably made.

10.3 Evaluation of Research Goals

This sections draws together a conclusion about the degree to which each goal set out in Section 1.2 was achieved. As these goals were labelled previously, they will simply be referred to by their label in the following text.

1. **Primary Goals**

   (a) The modular design of Bolvedere allowed for leveraging of parallel architectures. Furthermore, the nature in which these modules execute allowed them to process their data in a highly concurrent manner, and is best showcased in Chapter 7, where the entire 17-month school network dataset was processed in parallel over multiple hosts. This scalability allowed for ease in ramping up of processing power with the growth of networks and their bandwidth requirements. The disjoint nature of each module in Bolvedere, as well as the base system, allowed for ease of addition and removal of processor modules, based on one's needs and the requirements of the intended network.

   (b) The benefits found through further processing and filtering of NetFlow records showed to be promising, as it made the task of analysing this data more focused on the points that are relevant to what is being analysed. Bolvedere used this fact in Chapter 6 to initially prove that better detection of malicious activity can be achieved through the use of a lens that is more focused on a network's activities.

(c) Processing and analysis of reordered NetFlow records by Bolvedere's base was achieved though development of several processor modules, which were designed in Chapter 5. The multi-language support of ZMQ's allowed for use of the best tool for the job when developing these processor modules, and results produced in both synthetic (Chapter 6) and real-world (Chapter 7) testing support this fact. Through these testing iterations, Bolvedere as a whole was shown to be able to sink NetFlow records, and process and analyse them to produce meaningful information about events on a network.

2. **Secondary Goals**

(a) Testing of Bolvedere on live recorded data in Chapter 7 proved fruitful, as results from the performed analysis brought to light many anomalous events within the 17-month dataset. When further analysis was performed on these results by viewing the original NetFlow records, the events detected by these processor modules proved true[2]. Furthermore, Bolvedere's ability to handle and process a large throughput of NetFlow records helped to prove its potential in live application.

(b) Hardware acceleration of the collector subsystem of Bolvedere was achieved with great success. The collector software and hardware implementations were shown to be interchangeable in Chapter 9, with no effect on the output from the Bolvedere base system. The only difference noted here was a gain in bandwidth throughput in use of the hardware implementation. As previously mentioned, this also added to future-proofing of Bolvedere.

In closing, this research aimed to implement a network flow analysis system specifically targeted at NetFlow version 9 and IPFIX. Having achieved the goals set out in Section 1.2, this research has been concluded to serve as a healthy addition to the research areas involving networks, network analysis and network security.

10.4 Real-World Application

Two of the initial aspects of Bolvedere put forth at design time were accessibility to this system, and its scalability. Commenting on the latter first, it was shown in both the controlled testing (Chapter 6) and the real-world testing (Chapter 7) that the collector, publisher and all processor

[2]There was strong evidence of the occurrence of the detected event.

modules of this system could run within a single host. Furthermore, as all parts of Bolvedere run disjointly, any subsystem of Bolvedere can be run separately on its own host. This allows Bolvedere as a whole to scale out horizontally as hardware resource requirements increase past that which a single host can provide. Furthermore, implementation of an in-hardware collector allowed Bolvedere to increase its overall performance further.

Focusing on accessibility, Bolvedere needed to be accessible to society at large, especially those that cannot afford or do not have access to the hardware needed to run multiple hosts. As mentioned before, the ability to run multiple subsystems of Bolvedere within a single host helped alleviate the barrier to entry in running this system. This was then coupled with the choice between a software- or hardware-based collector. Both hardware and software implementations of the collector are fully featured, and are interchangeable with no functional change other than performance, where the hardware implementation is designed for far higher throughput rates. These design and implementation choices have created a system that is applicable in single-application and multi-application situations. This caters for both someone wanting to just learn how the system works, and a fully scaled-out system churning through multiple gigabits per second of network traffic, as shown to be possible in Section 7.3.

Another major point that one should take note of when considering the use of this system is its simplicity. One need only configure what fields one wishes to see published from the acquired NetFlow records, and start the publisher and collector. The publisher will configure itself and then communicate with and configure any collector that is connected to the publisher. This means that the user need not know the complex workings within the software, or have a deep understanding of hardware logic level programming in order to initiate Bolvedere. One simply needs to input what fields are required and the order in which they wish to have them published to a processor module. This was detailed in Section 4.3.

This leads to the concluding point of Bolvedere. Bolvedere was designed as an easy to access, scalable system that removes, or to a large degree mitigates, the need to understand the complex workings of the NetFlow protocol, hardware logic level and low level C programming. It acts as a complete out-of-the-box system that, beyond configuring what NetFlow fields one wants to work with, requires little to no other configuration or maintenance. This means that the user can more quickly get down to developing a Bolvedere processor module to analyse the NetFlow records that they need to collect meaningful information about a network or, as NetFlow records can be exported to any device with an IP, multiple networks locally or globally.

10.5 Research Contribution

This research has made the following novel contributions:

1. This research developed a hardware-accelerated NetFlow version 9[3] processing system, namely Bolvedere. This research initially sought to develop a platform which would act as a base for NetFlow analysis. The user of Bolvedere would simply specify what part of the NetFlow records they were interested in processing, and Bolvedere's base system would ensure this data was published for the user to use. This removed the need for the user to redevelop and include NetFlow discernment logic in every NetFlow analysis tool they would develop. Instead, the process of developing a NetFlow analysis tool was streamlined by the user only having to focus on development of the analysis tool. This base system that handled the processing, filtering and restructuring of NetFlow records by request of a user, was hardware-accelerated in this research to improve throughput and to future-proof its implementation.

2. This research was limited to processing of NetFlow records (Section 1.3); however, this implementation is able to process any well-defined protocol that follows a template and data record scheme. This functionality is true of both hardware and software implementations of the collector with the rest of the Bolvedere system in its entirety.

10.6 Future Work

Information technology is an ever-growing field and, as such, this research should aim to grow with it. To better streamline this process, some ideas and proposals for future extensions of Bolvedere are listed below. These ideas arose as extensions to the work performed in this research, or because some areas proposed fell out of the scope of this research at the time it was conducted.

- There is always a need for more analysis tools, and in this research, this comes in the form of processor modules. Although there are a few provided with this research, the focus of this research was to implement and ensure the correct functionality of the Bolvedere base system. For this reason, there is a lot of space to grow the analysis side of Bolvedere through development of new processor modules.

[3]NetFlow version 9 is also follows the same format as IPFIX and as such, Bolvedere can also process IPFIX logs.

- As NetFlow works with a pair of records, these being data and template, one can look to use templates to better describe other forms of data. There is a lot of space left open in the NetFlow version 9 and IPFIX standards in which to define custom fields. If one were to use these fields to describe new protocols in development, this would further grow the support base of Bolvedere.

- Continuing on the previous future work, as the NetFlow protocols that make use of templates are well-defined, they can be used to record non-network traffic. If a system that collects other forms of data follows the standard motion of a template describing what is in the data it is transmitting, and then sending data in the format required by the protocol, one can effectively log other forms of data using the NetFlow protocol. This can be directly extended to Bolvedere to perform analysis on this non-network flow data, in order to better analyse this new form of data. This should require minimal to no modification of the Bolvedere system, other than the addition of processor modules to handle this non-network based data.

- The hardware-accelerated implementation of Bolvedere in its current state only supports IPv4. As such, extension to IPv6 would extend the compatibility of the hardware implementation of this system. This would only require modification of the network-specific functional blocks.

- Research into the porting, supporting and testing of Bolvedere on other OSs such as BSD or Windows. This will allow for an even wider access to Bolvedere.

These proposals are not limited to the scope of this implementation only, and implementation on a more suited platform is encouraged if that is the case. This research is not intended to be the pinnacle of all network analysis, and as a collective research community we should strive continuously into new research areas, even if that means rendering obsolete existing implementations; stagnation will only serve to kill this area of research.

With this mindset to grow this research and explore new research ventures, the ever-needed research areas of networks, network security and Computer Science as a whole will continue to grow steadily.

References

Abramson, N. *Development of the alohanet*. *IEEE Transactions onInformation Theory*, 31(2):119–123, 1985. doi:10.1109/TIT.1985.1057021.

Abu Rajab, M., Zarfoss, J., Monrose, F., and Terzis, A. *A multifaceted approach to understanding the botnet phenomenon*. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52. ACM, 2006. doi:10.1145/1177080.1177086.

Amdahl, G. M. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. *Solid-State Circuits Society Newsletter, IEEE*, 12(3):19–20, 2007. doi:10.1145/1465482.1465560.

Argyraki, K. and Cheriton, D. *Network capabilities: The good, the bad and the ugly*. *ACM HotNets-IV*, 139:140, 2005.

Arregoces, M. and Portolani, M. *Data Center Fundamentals*. Cisco Press, 2003. ISBN 978-1-58-714075-4, 495 pages.

Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. *An efficient multicast protocol for content-based publish-subscribe systems*. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 262–272. IEEE, 1999. doi:10.1109/ICDCS.1999.776528.

Barnett, R. J. and Irwin, B. *Towards a taxonomy of network scanning techniques*. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 1–7. ACM, 2008. doi:10.1145/1456659.1456660.

Bartlett, K. A., Scantlebury, R. A., and Wilkinson, P. T. *A note on reliable full-duplex transmission over half-duplex links*. *Communications of the ACM*, 12(5):260–261, 1969. doi:10.1145/362946.362970.

Bening, L. and Foster, H. *Principles of verifiable RTL design*. Springer, 2001. ISBN 978-0-79-237368-1.

Biggs, J. *Hackers release source code for a powerful ddos app called mirai*. Online, October 2016. Accessed 30th June 2017.
URL `https://techcrunch.com/2016/10/10/hackers-release-source-code-for-a-powerful-ddos-app-called-mirai/`

Biham, E. and Seberry, J. *Pypy: another version of python*. Technical report, eSTREAM, ECRYPT Stream Cipher Project, 2006.

Bilge, L. and Dumitras, T. *Before we knew it: an empirical study of zero-day attacks in the real world*. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012. doi:10.1145/2382196.2382284.

Bishop, M. *What is Computer Security? Security & Privacy, IEEE*, 1(1):67–69, 2003. doi:10.1109/MSECP.2003.1176998.

Bonderud, D. *Leaked Mirai Malware Boosts IoT Insecurity Threat Level*. October 2016. Accessed 11 August 2017.
URL `https://securityintelligence.com/news/leaked-mirai-malware-boosts-iot-insecurity-threat-level/`

Boyes, W. *Instrumentation Reference Book*. Butterworth-Heinemann, 3rd edition, 2002. ISBN 978-0-75-068308-1.

Bracewell, R. *The fourier transform and iis applications. New York*, 5, 1965.

Braden, R. *RFC 1122: Requirements for Internet hosts*. 1989. Accessed 27th June 2017.
URL `https://tools.ietf.org/html/rfc1122`

Brasford, G. *The Welchia worm. Global Information Assurance Certification*, page 27, 2004.

Cain, B. *Source-specific multicast for ip*. 2006. Accessed 15th November 2017.
URL `https://tools.ietf.org/html/rfc4607`

Cerf, V. G. and Kahn, R. E. *A protocol for packet network intercommunication. ACM SIGCOMM Computer Communication Review*, 35(2):71–82, 2005. doi:10.1145/1064413.1064423.

Chen, J. C., Hudson, R. E., and Yao, K. *Maximum-likelihood source localization and unknown sensor location estimation for wideband signals in the near-field. IEEE transactions on Signal Processing*, 50(8):1843–1854, 2002. doi:10.1109/TSP.2002.800420.

Cheshire, S. and Krochmal, M. *RFC 6761: Special-Use Domain Names*. 2013. Accessed 10th October 2017.
URL https://tools.ietf.org/html/rfc6761

Chiang, K. and Lloyd, L. *A Case Study of the Rustock Rootkit and Spam Bot*. *HotBots*, 7:10–10, 2007.

Cho, Y. H., Navab, S., and Mangione-Smith, W. H. *Specialized hardware for deep network packet filtering*. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 452–461. Springer, 2002.

Cisco. *NetFlow V9 Export Format*. 2003a. Accessed 13th February 2015.
URL https://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/nfexpfv9.html

Cisco. *NetFlow Version 9 Flow-Record Format*. 2003b. Accessed 19th May 2016.
URL https://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html

Cisco. *Network Address Translation (NAT) FAQ*. 2015. Accessed 4th August 2015.
URL http://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html

Cisco. *The zettabyte era–trends and analysis*. 2017.
URL http://www.cisco.com/c/en/us/solutions/collateral/serviceprovider/visualnetworking-index-vni/VNI_Hyperconnectivity_WP.html

Claise, B. *RFC 3954: Cisco systems NetFlow services export version 9*. 2004. Accessed 29th July 2015.
URL https://www.ietf.org/rfc/rfc3954

CloudFlare, Inc. *Cloudflare protects and accelerates any website online*. 2016. Accessed 19th May 2016.
URL https://www.cloudflare.com/lp/overview/?_bt=91222385652&_bk=cloudflare&_bm=e&_bn=g&gclid=CJWto6PI6MwCFdRuGwodpp0DDQ

Cockshott, P. and Renfrew, K. *SIMD Programming Manual for Linux and Windows*. Springer, 2004. ISBN 978-1-84-996920-8, 11–22 pages.

Collingbourne, P., Donaldson, A. F., Ketema, J., and Qadeer, S. *Interleaving and lock-step semantics for analysis and verification of gpu kernels.* In *ESOP*, pages 270–289. Springer, 2013. doi:10.1007/978-3-642-37036-6.

Coltun, R., Ferguson, D., Moy, J., and Lindem, A. *RFC 5340: OSPF for IPv6.* 2008. Accessed 27th August 2015.
URL `https://tools.ietf.org/html/rfc5340`

Computer Society of the IEEE. *1076-2008 - IEEE Standard VHDL Language Reference Manual.* Technical report, IEEE, 1988.
URL `http://perso.telecom-paristech.fr/~guilley/ENS/20161206/TP/tp_syn/doc/IEEE_VHDL_1076-1987.pdf`

Cotton, M., Eggert, L., Touch, J., Westerlund, M., and Cheshire, S. *Internet Assigned Numbers Authority (IANA) procedures for the management of the service name and transport protocol port number registry.* 2011. Accessed 23rd October 2017.
URL `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`

Cotton, M. and Vegoda, L. *RFC 5735: Special use IPv4 addres.* 2010. Accessed 30th August 2017.
URL `https://tools.ietf.org/html/rfc5735`

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. *N-variant systems: A secretless framework for security through diversity.* In *Usenix Security*, volume 6, pages 105–120. 2006.

Czyz, J., Kallitsis, M., Gharaibeh, M., Papadopoulos, C., Bailey, M., and Karir, M. *Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks.* In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 435–448. ACM, 2014. doi:10.1145/2663716.2663717.

Dabbagh, M., Ghandour, A. J., Fawaz, K., El Hajj, W., and Hajj, H. *Slow port scanning detection.* In *Information Assurance and Security (IAS), 2011 7th International Conference on*, pages 228–233. IEEE, 2011. doi:10.1109/ISIAS.2011.6122824.

Dagum, L. and Enon, R. *OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. doi:10.1109/99.660313.

Dan, A. and Towsley, D. *An approximate analysis of the lru and fifo buffer replacement schemes. SIGMETRICS '90 Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 18:143–152, 1990. doi:10.1145/98460.98525.

Danyliw, R., Meijer, J., and Demchenko, Y. *RFC 5070: Incident Object Description Exchange Format (IODEF).* 2012.
URL `https://tools.ietf.org/html/rfc5070`

Deering, S. *RFC 1112: Host extensions for IP multicasting., August 1989.* 1997. Accessed 16th July 2016.
URL `https://tools.ietf.org/html/rfc1112`

Deering, S. E. *RFC 2460: Internet Protocol, version 6 (IPv6) specificat.* 1998. Accessed 16th July 2016.
URL `https://www.ietf.org/rfc/rfc2460.txt`

Dice, D., Shalev, O., and Shavit, N. *Transactional Locking II.* In *DISC'06 Proceedings of the 20th international conference on Distributed Computing*, pages 194–208. Springer, 2006.

Digital Equipment Corporation, Intel Corporation and Xerox Corporation. *The Ethernet A Local Area Network Data Link Layer and Physical Layer Specifications V2.0.* Technical report, Xerox PARC, 1982.
URL `http://decnet.ipv7.net/docs/dundas/aa-k759b-tk.pdf`

Dougherty, C., Havrilla, J., Hernan, S., and Lindner, M. *W32/Blaster worm.* August 2003. Accessed 9th March 2015.
URL `http://www.cert.org/historical/advisories/CA-2003-20.cfm`

Douligeris, C. and Mitrokotsa, A. *DDoS attacks and defense mechanisms: classification and state-of-the-art. Computer Networks*, 44(5):643–666, 2004. doi:doi.org/10.1016/j.comnet.2003.10.003.

Doulos. *A Brief History of VHDL.* 2014. Accessed 13th February 2015.
URL `https://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/`

Doyen, G., Waldburger, M., Celeda, P., Sperotto, A., and Stiller, B. *Emerging Management Mechanisms for the Future Internet: 7th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2013, Barcelona, Spain, June 25-28, 2013, Proceedings*, volume 7943. Springer, 2013. ISBN 978-3-64-238997-9, 151 pages.

Durumeric, Z., Wustrow, E., and Halderman, J. A. *Zmap: Fast internet-wide scanning and its security applications.* In *USENIX Security Symposium*, volume 8, pages 47–53. 2013.

Eastlake 3rd, D. and Jones, P. *US secure hash algorithm 1 (SHA1).* 2001. Accessed 7th April 2016.
URL `http://www.rfc-editor.org/rfc/rfc3174.txt`

Egevang, K. and Francis, P. *RFC 1631: The IP network address translator (NAT).* 1994. Accessed 23rd October 2017.
URL `https://tools.ietf.org/html/rfc1631`

Embedded Micro. *Embedded micro make technology*. 2015. Accessed 23th June 2015.
URL `https://embeddedmicro.com/tutorials/mojo-fpga-beginners-guide/how-does-an-fpga-work`

Enderton, H. and Enderton, H. B. *A Mathematical Introduction to Logic*. New York: Harcourt Academic Press, second edition edition, 2001. ISBN 978-0-08-057038-9.

Estan, C., Keys, K., Moore, D., and Varghese, G. *Building a better netflow*. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 245–256. ACM, 2004. doi:10.1145/1015467.1015495.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. *The many faces of publish/subscribe*. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003. doi:10.1145/857076.857078.

Facebook. *How does Facebook suggest tags?* February 2016. Accessed 4th February 2016.
URL `https://www.facebook.com/help/122175507864081`

Feinstein, L., Schnackenberg, D., Balupari, R., and Kindred, D. *Statistical approaches to ddos attack detection and response*. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 1, pages 303–314. IEEE, 2003. doi:10.1109/DISCEX.2003.1194894.

Fiat, A. and Naor, M. *Broadcast encryption*. In *Advances in Cryptology-CRYPTO'93*, pages 480–491. Springer, 1994. doi:10.1007/3-540-48329-2_40.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. *RFC 2616: HyperText Transfer Protocol–HTTP/1.1.* 2009. Accessed 10th August 2016.
URL `http://www.rfc.net/rfc2616.html`

Francis, R. J., Rose, J., and Vranesic, Z. G. *Field-Programmable Gate Arrays*. Springer Science & Business Media, 1992. ISBN 978-0-79-239248-4.

Furnell, S., Bryant, P., and Phippen, A. D. *Assessing the security perceptions of personal Internet users*. *Computers & Security*, 26(5):410–417, 2007. doi:10.1016/j.cose.2007.03.001.

Gallagher, S. *Double-dip Internet-of-Things botnet attack felt across the Internet*. Online, October 2016. Accessed 30th June 2017.
URL `https://arstechnica.com/security/2016/10/double-dip-internet-of-things-botnet-attack-felt-across-the-internet/`

Galtsev, A. A. and Sukhov, A. M. *Network attack detection at flow level*. In *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 326–334. Springer, 2011.

Geng, X. and Whinston, A. B. *Defeating Distributed Denial of Service attacks*. *IT Professional*, 2(4):36–42, 2000. doi:10.1109/6294.869381.

Gil, T. M. and Poletto, M. *Multops: a data-structure for bandwidth attack detection*. In *USENIX Security Symposium*. 2001.

Gordon, W. J., Fairhall, A., and Landman, A. *Threats to Information Security-Public Health Implications*. *New England Journal of Medicine*, 377(8):707–709, 2017. doi:10.1056/NEJMp1707212.

Grafarend, E. W. *Linear and nonlinear models: fixed effects, random effects, and mixed models*. de Gruyter, 2006. ISBN 978-3-11-016216-5. Page 556.

Gragido, W. *Understanding indicators of compromise (ioc) part ii*. 2012. Accessed 6th November 2015.
URL `http://blogs.rsa.com/understanding-indicators-of-compromise-ioc-part-i/Ov{ě}{ř}enokedni`

Gu, G., Perdisci, R., Zhang, J., and Lee, W. *Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.* In *USENIX Security Symposium*, volume 5, pages 139–154. 2008.

Gu, G., Porras, P. A., Yegneswaran, V., Fong, M. W., and Lee, W. *Bothunter: Detecting malware infection through ids-driven dialog correlation*. In *Usenix Security*, volume 7, pages 1–16. 2007.

Hagan, M. T., Demuth, H. B., Beale, M. H., and De Jesús, O. *Neural Network Design*, volume 10 of *1*. PWS publishing company Boston, 1996. ISBN 978-0-97-173210-0.

Hall, L. O., Bensaid, A. M., Clarke, L. P., Velthuizen, R. P., Silbiger, M. S., and Bezdek, J. C. *A comparison of neural network and fuzzy clustering techniques in segmenting magnetic resonance images of the brain*. *Neural Networks, IEEE Transactions on*, 3(5):672–682, 1992. doi:10.1109/72.159057.

Halpin, H. *The philosophy of anonymous: Ontological politics without identity*. *Radical Philosophy*, 176:19, 2012.

Hansen, P. B. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer, 2013. ISBN 978-1-47-573472-0.

Haris, S., Ahmad, R., Ghani, M., and Waleed, G. M. *TCP SYN flood detection based on payload analysis*. In *Research and Development (SCOReD), 2010 IEEE Student Conference on*, pages 149–153. IEEE, 2010. doi:10.1109/SCORED.2010.5703991.

Harrop, W. and Armitage, G. *Defining and evaluating greynets (sparse darknets)*. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 344–350. IEEE, 2005. doi:10.1109/LCN.2005.46.

Healey, A. *IEEE 802.3 Ethernet Working Group*. 1983. Accessed 12th September 2015. URL `http://www.ieee802.org/3/`

Hecht-Nielsen, R. *Theory of the backpropagation neural network*. In *Neural Networks. IJCNN., International Joint Conference on*, pages 593–605. IEEE, 1989. doi:10.1109/IJCNN.1989.118638.

Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011. ISBN 978-0-12-370490-0.

Herbert, A. and Irwin, B. *FPGA Based Implementation of a High Performance Scalable NetFlow Filter*. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2015*. 2015.

Herbert, A. and Irwin, B. *Adaptable exploit detection through scalable netflow analysis*. In *Information Security for South Africa (ISSA), 2016*, pages 121–128. IEEE, 2016a. doi:10.1109/ISSA.2016.7802938.

Herbert, A. and Irwin, B. *Towards malicious network activity mitigation through subnet reputation analysis*. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2016*. 2016b.

Herbert, A. and Irwin, B. *Towards Enhanced Threat Intelligence Through NetFlow Distillation*. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2018*. 2018.

Herzberg, B., Bekerman, D., and Zeifman, I. *Breaking down mirai: An iot ddos botnet analysis*. October 2016. Accessed 12th July 2017.
URL `https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html`

Hintjens, P. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013. ISBN 978-1-44-933443-7.

Hodges, A. *Alan Turing: The Enigma of Intelligence*. London: Burnett Books, 1st edition, 1983. ISBN 978-0-09-911641-7. Page 111.

Housely, R., Curran, J., Huston, G., and Conrad, D. *RFC 7020: The Internet Numbers Registry System*. 2013. Accessed 16th July 2017.
URL `https://tools.ietf.org/html/rfc7020`

Huston, G. *NetFlow Packet Version 5 (V5)*. 2006a. Accessed 13th February 2015.
URL `http://netflow.caligare.com/netflow_ v5.htm`

Huston, G. *NetFlow Packet Version 8 (V8)*. 2006b. Accessed 13th February 2015.
URL `http://netflow.caligare.com/netflow_ v8.htm`

Huston, G. *IPv4 Address Report*. February 2014. Accessed 6th February 2014.
URL `http://www.potaroo.net/tools/ipv4/index.html`

IEEE 802.3 Working Group. *Part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications*. *IEEE Std*, 802, 2000. doi:10.1109/IEEESTD.2010.5621025.

IEEE Standards Association. *802.3ae-2002 - IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. IEEE, 2002. ISBN 0-7381-3287-X.

Intel Corperation. *IA-64 Application Instruction Set Architecture Guide*. Technical report, Intel Corperation, 2010.
URL `https://www.csee.umbc.edu/portal/help/architecture/aig.pdf`

Intel Corperation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Technical report, Intel Corperation, 2015.
URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Intel Corperation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Technical report, 2016.

Internet Assigned Numbers Authority. *IANA*. 2017. Accessed 16th October 2017.
URL https://www.iana.org/

Ioannidis, S., Keromytis, A. D., Bellovin, S. M., and Smith, J. M. *Implementing a distributed firewall*. In *Proceedings of the 7th ACM conference on Computer and Communications Security*, pages 190–199. ACM, 2000. doi:10.1145/352600.353052.

Irwin, B. *A framework for the application of network telescope sensors in a global IP network*. Ph.D. thesis, Rhodes University, 2011.

Irwin, B. *A network telescope perspective of the conficker outbreak*. In *Information Security for South Africa (ISSA), 2012*, pages 1–8. IEEE, 2012. doi:10.1109/ISSA.2012.6320455.

Irwin, B. *A Source Analysis of the Conficker Outbreak from a Network Telescope*. *SAIEE Africa Research Journal*, 104(2):38, 2013.

Jacob, N. and Brodley, C. *Offloading IDS Computation to the GPU*. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 371–380. IEEE, 2006. doi:10.1109/ACSAC.2006.35.

Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. *Inferring TCP connection characteristics through passive measurements*. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1582–1592. IEEE, 2004. doi:10.1109/INFCOM.2004.1354571.

JAVAPIPE. *Top 3 DDoS Attacks Toughest To Block*. April 2017. Accessed 27 September 2017.
URL https://javapipe.com/ddos/blog/top-3-ddos-attacks-toughest-to-block/

Jensen, M. and Gruschka, N. *Flooding attack issues of web services and service-oriented architectures.* In *Unsere Jahrestagung - GI - Informatik*, pages 117–122. 2008.

Jin, C., Wang, H., and Shin, K. G. *Hop-count filtering: an effective defense against spoofed ddos traffic*. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 30–41. ACM, 2003. doi:10.1145/948109.948116.

Jo, H.-H., Karsai, M., Kertész, J., and Kaski, K. *Circadian pattern and burstiness in mobile phone communication*. *New Journal of Physics*, 14(1):013055, 2012. doi:10.1088/1367-2630/14/1/013055.

Johnson, P. C., Kapadia, A., Tsang, P. P., and Smith, S. W. *Nymble: Anonymous ip-address blocking*. *Privacy Enhancing Technologies*, pages 113–133, 2007.

Juniper Networks. *Juniper Flow Monitoring*. 2011. Accessed 7th November 2017. URL `https://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf`

Kane, G. and Heinrich, J. *MIPS RISC Architectures*. Prentice-Hall, Inc., 2 edition, 1992. ISBN 978-0-13-590472-5.

Kerr, D. R. and Bruins, B. L. *Network flow switching and flow data export*. June 5 1996. US Patent 6,243,667.

Khrer, M., Hupperich, T., Rossow, C., and Holz, T. *Exit from hell? reducing the impact of amplification ddos attacks.* In *USENIX Security Symposium*, pages 111–125. 2014.

Kim, M., Na, H., Chae, K., Bang, H., and Na, J. *A combined data mining approach for ddos attack detection*. In *Information Networking. Networking Technologies for Broadband and Mobile Networks*, pages 943–950. Springer, 2004.

King, K. F. *Geolocation and federalism on the internet: Cutting internet gambling's gordian knot*. *Colum. Sci. & Tech. L. Rev.*, 11:41–58, 2010.

Kolias, C., Kambourakis, G., Stavrou, A., and Voas, J. *DDoS in the IoT: Mirai and Other Botnets*. *Computer*, 50(7):80–84, 2017. doi:10.1109/MC.2017.201.

Korb, B. *Implementation of POPI Act means companies must secure their information*. *IT Governance and Risk Management, ITWeb*, 10, 2013.

Kosko, B. *Neural Networks and Fuzzy Systems: a dynamical systems approach to machine intelligence/book and disk*. Prentice-Hall International, 1992. ISBN 978-0-13-611435-2.

Kozierok, C. M. *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*. No Starch Press, 2005. ISBN 978-1-59-327095-7.

Krebs, B. *KrebsOnSecurity Hit With Record DDoS*. September 2016. Accessed 11 August 2017.
URL `https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/`

Krebs, B. *Who is anna-senpai, the Mirai worm author*. January 2017. Accessed 11th August 2017.
URL `https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/`

Kumar, S., Jantsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., and Hemani, A. *A network on chip architecture and design methodology*. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 117–124. IEEE, 2002. doi:10.1109/ISVLSI.2002.1016885.

Kuon, I. and Rose, J. *Measuring the gap between FPGAs and ASICs*. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007. doi:10.1109/TCAD.2006.884574.

Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. *Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies*. *IEEE transactions on Computers*, (12):1352–1361, 2001. doi:10.1109/TC.2001.970573.

Lee, S., Min, S.-J., and Eigenmann, R. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization*. *ACM Sigplan Notices*, 44(4):101–110, 2009. doi:10.1145/1594835.1504194.

Leighton, F., Maggs, B., and Rao, S. *Packet routing and job-shop scheduling in O (Congestion+Dilation) steps*. *Combinatorica*, 14(2):167–186, 1994.

Leslie, D. A. *Legal Principles for Combatting Cyberlaundering*. Springer, 2014. ISBN 978-3-31-906415-4. Page 7.

Leyden, J. *Slammer: Why security benefits from proof of concept code*. February 2003. Accessed 11th August 2017.
URL `https://www.theregister.co.uk/2003/02/06/slammer_why_security_benefits/`

Li, J., Berg, S., Zhang, M., Reiher, P., and Wei, T. *Drawbridge: Software-defined DDOS-resistant traffic engineering*. *ACM SIGCOMM Computer Communication Review*, 44(4):591–592, 2015. doi:10.1145/2740070.2631469.

Lim, J. S. *Two-dimensional signal and image processing*. Prentice Hall, 1990. ISBN 978-0-13-935322-2. Page 710.

Liu, Y., Corbett, C., Chiang, K., Archibald, R., Mukherjee, B., and Ghosal, D. *Sidd: A framework for detecting sensitive data exfiltration by an insider attack*. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009. doi: 10.1109/HICSS.2009.390.

Lyon, G. *The Art of Port Scanning*. October 2015. Accessed 31st October 2015.
URL `https://nmap.org/nmap_doc.html`

Lyon, G. F. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009. ISBN 978-0-97-995871-7.

Malkin, G. *RFC 2453: Rip version 2*. 1998. Accessed 2 February 2016.
URL `https://tools.ietf.org/html/rfc2453.html`

Markovic, D., Chang, C., Richards, B., So, H., Nikolic, B., and Brodersen, R. W. *ASIC Design and Verification in an FPGA Environment*. In *Custom Integrated Circuits Conference, 2007. CICC'07. IEEE*, pages 737–740. IEEE, 2007. doi:10.1109/CICC.2007.4405836.

Marple Jr, S. L. *Digital spectral analysis with applications*. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1987, 512 p.*, 1, 1987. doi:10.1121/1.398548.

Matalytski, S. *System and methodology protecting against key logger spyware*. March 30 2006. US Patent App. 11/308,506.

Maynor, D. *Metasploit toolkit for penetration testing, exploit development, and vulnerability research*. Elsevier, 2011. ISBN 978-1-59-749074-0.

Mersereau, R. M. and Oppenheim, A. V. *Digital reconstruction of multidimensional signals from their projections*. *Proceedings of the IEEE*, 62(10):1319–1338, 1974. doi:10.1109/PROC.1974.9625.

Michael, J. Q. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2004. ISBN 978-0-07-282256-4.

Microsoft. *Microsoft Security Bulletin MS01-033 - Critical*. Novemeber 2003a. Accessed 9th March 2015.
URL `https://technet.microsoft.com/library/security/ms01-033`

Microsoft. *Microsoft Security Bulletin MS03-026 - Critical*. July 2003b. Accessed 27th January 2016.
URL https://technet.microsoft.com/library/security/ms03-026

Microsoft. *Microsoft Security Bulletin MS03-039 - Critical*. September 2003c. Accessed 27th January 2016.
URL https://technet.microsoft.com/en-us/library/security/ms03-039.aspx

Microsoft. *Microsoft Security Bulletin MS08-067 - Critical*. October 2008. Accessed 31st October 2015.
URL https://technet.microsoft.com/en-us/library/security/ms08-067.aspx

Microsoft. *Worm:Win32/Conficker.E*. April 2009. Accessed 31st October 2015.
URL https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Worm:Win32/Conficker.E

Miller, D. *Softflowd*. 2016. Accessed 30th April 2016.
URL http://www.mindrot.org/projects/softflowd/

Mills, D. *RFC 904: Exterior Gateway Protocol Formal Specification, DARPA Network Working Group Report*. 1984. Accessed 12th June 2017.
URL https://tools.ietf.org/html/rfc904

Mills, D. *RFC 956: Algorithms for synchronizing network clocks*. 1985. Accessed 30th April 2017.
URL https://tools.ietf.org/html/rfc956

Mirkovic, J. and Reiher, P. *A taxonomy of DDoS attack and DDoS defense mechanisms*. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004. doi:10.1145/997150.997156.

Mohurle, S. and Patil, M. *A brief study of wannacry threat: Ransomware attack 2017*. *International Journal*, 8(5), 2017. doi:10.26483/ijarcs.v8i5.4021.

Moore, D. S. *Chi-square tests*. Technical report, DTIC Document, 1976. Accessed 30th April 2016.
URL http://www.dtic.mil/get-tr-doc/pdf?AD=ADA033287

Moy, J. *RFC 2328: OSPF Version 2*. 1998. Accessed 11th November 2015.
URL https://tools.ietf.org/html/rfc2178

Muniz, J. *Web Penetration Testing with Kali Linux*. Packt Publishing Ltd, 2013. ISBN 978-1-78-216317-6.

Munz, G. and Carle, G. *Real-time analysis of flow data for network attack detection*. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 100–108. IEEE, 2007. doi:10.1109/INM.2007.374774.

Murakami, K., Irie, N., and Tomita, S. *SIMP (Single Instruction stream/Multiple instruction Pipelining): a novel high-speed single-processor architecture*. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 78–85. ACM, 1989. doi:10.1145/74926.74935.

Muuss, M. *The story of the ping program*. 1983. Accessed 20th March 2016.
URL `http://mirrors.pdp-11.ru/_vax/www.bandwidthco.com/whitepapers/netforensics/icmp/The%20Story%20of%20the%20PING%20Program.pdf`

National Instruments. *FPGA Fundamentals*. 2015. Accessed 22th June 2015.
URL `http://www.ni.com/white-paper/6983/en/`

Navas, J. C. and Imielinski, T. *Geocast-geographic addressing and routing*. In *Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*, pages 66–76. ACM, 1997. doi:10.1145/262116.262132.

Nazario, J. and Holz, T. *As the net churns: Fast-flux botnet observations*. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 24–31. IEEE, 2008. doi:10.1109/MALWARE.2008.4690854.

Needham, R. M. *Denial of Service*. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 151–153. ACM, 1993. doi:10.1145/168588.168607.

New Jersey Cybersecurity and Communications Integration Cell. *Mirai Botnet*. December 2016. Accessed 12 July 2017.
URL `https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet`

Nikkel, S. *Network connection speeds reference*. August 2013. Accessed.
URL `http://www.ertyu.org/steven_nikkel/netspeeds.html`

nVidia. *Compute Unified Device Architecture Programming guide*. Technical report, 2007.

O'Gorman, G. and McDonald, G. *Ransomware: A growing menace*. Technical report, Symantec, 2012.

URL http://www.01net.it/whitepaper_library/Symantec_Ransomware_Growing_Menace.pdf

O'Gorman, J., Kearns, D., and Aharoni, M. *Metasploit: The penetration tester's guide*. No Starch Press, 2011. ISBN 978-1-59-327288-3.

Omana, M., Papasso, G., Rossi, D., and Metra, C. *A model for transient fault propagation in combinatorial logic*. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 111–115. IEEE, 2003. doi:10.1109/OLT.2003.1214376.

Openstax CNX. *CPU Structure and Functions*. 2015. Accessed 9 November 2015.
URL http://cnx.org/contents/b87ac62d-79f2-44f2-b4ae-44eb306a5644@1/CPU-Structure-and-Functions

Oracle. *proc(1)*. 2007. Accessed 9 November 2015.
URL http://www-it.desy.de/cgi-bin/man-cgi?proc+4

Oracle. *Java and you, download today*. June 2015. Accessed 16th June 2015.
URL https://www.java.com/en/

Oxford English Dictionary. *Hardware acceleration*. 2015. Accessed 5th August 2015.
URL http://www.oed.com/view/Entry/84197

Oxford University Press. *Oxford dictionaries language matters*. November 2015. Accessed 9 November 2015.
URL http://www.oxforddictionaries.com/definition/english/subsystem

Paar, C. and Pelzl, J. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. ISBN 978-3-64-204100-6.

Paganini, P. *Experts from MalwareMustDie spotted a new ELF trojan backdoor, dubbed ELF Linux/Mirai, which is now targeting IoT devices*. September 2016. Accessed 12th July 2017.
URL http://securityaffairs.co/wordpress/50929/malware/linux-mirai-elf.html

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. *Improving GPGPU concurrency with elastic kernels*. In *ACM SIGPLAN Notices*, volume 48, pages 407–418. ACM, 2013. doi:10.1145/2499368.2451160.

Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. *Characteristics of Internet background radiation*. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 27–40. ACM, 2004. doi:10.1145/1028788.1028794.

Parhami, B. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Inc., 2009. ISBN 978-0-19-512583-2.

Passint, R. S., Oberlin, S. M., and Fromm, E. C. *Messaging facility with hardware tail pointer and software implemented head pointer message queue for distributed memory massively parallel processing system*. December 3 1996. US Patent 5,581,705.

Patterson, D. A. and Hennessey, J. L. *Computer Organization and Design: the Hardware/-Software Interface*. Morgan Kaufmann, 2nd edition, 1998. ISBN 978-0-12-407726-3.

Paxson, V. *Bro: a system for detecting network intruders in real-time*. *Computer networks*, 31(23):2435–2463, 1999. doi:10.1016/S1389-1286(99)00112-7.

Pennefather, S. and Irwin, B. *An exploration of geolocation and traffic visualisation using network flows*. In *Information Security for South Africa (ISSA), 2014*, pages 1–6. IEEE, 2014. doi:10.1109/ISSA.2014.6950507.

Perdisci, R., Lanzi, A., and Lee, W. *Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables*. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 301–310. IEEE, 2008. doi:10.1109/ACSAC.2008.22.

Phaal, P., Panchen, S., and McKee, N. *RFC 3176: InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. September 2001. Accessed 7th November 2017.
URL `https://tools.ietf.org/html/rfc3176`

Piedad, F. and Hawkins, M. *High availability: design, techniques, and processes*. Prentice Hall Professional, 2001. ISBN 978-0-13-096288-1.

Plummer, D. C. *Rfc 826: An ethernet address resolution protocol*. 1982. Accessed 27th August 2015.
URL `https://tools.ietf.org/html/rfc826`

Porras, P., Saidi, H., and Yegneswaran, V. *A multi-perspective analysis of the Storm (peacomm) worm*. Technical report, Technical report, Computer Science Laboratory, SRI International, 2007. doi:10.1.1.126.2517.

Postel, J. *Comments on Internet Protocol and TCP*. August 1977. Accessed 1 August 2015.
URL `http://www.postel.org/ien/txt/ien2.txt`

Postel, J. *RFC 768: User Datagram Protocol*. 1980. Accessed 1st October 2015.
URL `https://tools.ietf.org/html/rfc768`

Postel, J. *RFC 791: Internet Protocol*. 1981a. Accessed 1st October 2015.
URL `https://tools.ietf.org/html/rfc791`

Postel, J. *RFC 792: Internet Control Message Protocol*. 1981b. Accessed 1st October 2015.
URL `https://tools.ietf.org/html/rfc792`

Provos, N. *A virtual honeypot framework.* In *USENIX Security Symposium*, volume 173, pages 1–14. 2004.

Qin, Y., Feng, D. G., Chen, K., and Lian, Y. F. *Research on monitor position in network situation assessment*. In *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pages 1660–1663. IEEE, 2012. doi:10.1109/ICICEE.2012.439.

Quinn, M. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill Higher Education, 2004. ISBN 978-0-07-282256-4.

Quittek, J., Zseby, T., Claise, B., and Zander, S. *RFC 3917: equirements for IP flow information export (IPFIX)*. 2004. Accessed 20th October 2015.
URL `https://tools.ietf.org/html/rfc3917`

Raghavan, B., Vishwanath, K., Ramabhadran, S., Yocum, K., and Snoeren, A. C. *Cloud control with distributed rate limiting*. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 337–348. ACM, 2007. doi:10.1145/1282380.1282419.

Ramanathan, R. *Intel® Multi-Core Processors: Making the Move to Quad-Core and Beyond*. Technical report, Intel, 2006.
URL `https://pdfs.semanticscholar.org/7570/65cc4263ee0c897f61b7facab4f2582c6796.pdf`

Reinders, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007. ISBN 978-0-59-651480-8.

Rekhter, Y., Li, T., and Hares, S. *RFC 4271: Border Gateway Protocol 4*. 2006. Accessed 20th December 2016.
URL `https://tools.ietf.org/html/rfc4271`

Rivest, R. *The MD5 message-digest algorithm*. 1992. Accessed 28th January 2016.
URL `https://tools.ietf.org/html/rfc1321`

Roads, C. *The computer music tutorial*. MIT press, 1996. ISBN 978-0-26-218158-7.

Rochester, N., Holland, J., Haibt, L., and Duda, W. *Tests on a cell assembly theory of the action of the brain, using a large digital computer*. *Information Theory, IRE Transactions on*, 2(3):80–93, 1956. doi:10.1109/TIT.1956.1056810.

Roesch, M. *Snort: Lightweight Intrusion Detection for Networks*. In *Large Installation System Administration*, volume 99, pages 229–238. 1999.

Rudman, L. and Irwin, B. *Characterization and analysis of NTP amplification based DDoS attacks*. In *Information Security for South Africa (ISSA), 2015*, pages 1–5. IEEE, 2015. doi:10.1109/ISSA.2015.7335069.

Rudman, L. and Irwin, B. *Characterization and Analysis of NTP Amplifier Traffic*. *SAIEE Africa Research Journal*, 107(2):54–64, 2016.

Rumelhart, D. E., McClelland, J. L., Group, P. R. *et al. Parallel distributed processing*. MIT press Cambridge, MA, USA:, 1987. ISBN 978-0-26-268053-0.

Sak, H., Senior, A. W., and Beaufays, F. *Long short-term memory recurrent neural network architectures for large scale acoustic modeling.* In *INTERSPEECH*, pages 338–342. 2014.

Sanford, B. *Integrated Graphics Solutions for Graphics-Intensive Applications*. Technical report, 2007.
URL `http://www.techspot.com/news/46773-amd-announces-radeon-hd-7970-claims-fastest-gpu-title.html`

Schuster, M. and Paliwal, K. K. *Bidirectional recurrent neural networks*. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. doi:10.1109/78.650093.

Seagate. *Data Sheet: Barracuda The Port of One*. 2011. Accessed 23rd October 2017.
URL `https://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf`

Setiono, R. and Liu, H. *Neural-network feature selector*. *IEEE transactions on neural networks*, 8(3):654–662, 1997. doi:10.1109/72.572104.

Shannon, C. and Moore, D. *The spread of the Witty Worm*. *Security & Privacy, IEEE*, 2(4):46–50, 2004. doi:10.1109/MSP.2004.59.

Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database system concepts*, volume 4. McGraw-Hill New York, 1997. ISBN 978-0-07-352332-3.

Smith, A. J. *Disk cache-miss ratio analysis and design considerations*. *ACM Transactions on Computer Systems (TOCS)*, 3(3):161–203, 1985. doi:10.1145/3959.3961.

Smith, J. O. *Mathematics of the discrete Fourier transform (DFT): with audio applicaitons*. Julius Smith, 2007. ISBN 978-0-97-456074-8.

Srivastava, A. and Giffin, J. *Tamper-resistant, application-aware blocking of malicious network connections. raid*. In *Recent Advances in Intrusion Detection*, pages 39–58. Springer, 2008.

Statista. *The Statistics Portal: Statistics and Studies from more than 18,000 Sources*. 2015. Accessed 6th August 2015.
URL `http://www.statista.com/statistics/267106/global-market-share-of-pc-processor-manufacturers/`

Stewart, J. *Calculus*. Cegage Learning, 8th edition, 2016. ISBN 978-0-53-849886-9.

Stroustrup, B. *The C++ programming language*. Pearson Education India, 1986. ISBN 978-0-20-170073-2.

Symantec. *Trojan.peacomm*. January 2007a. Accessed 27th January 2016.
URL `http://www.symantec.com/security_response/writeup.jsp?docid=2007-011917-1403-99`

Symantec. *W32.sqlexp.worm*. February 2007b. Accessed 28th January 2016.
URL `https://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99`

Templeton, S. J. and Levitt, K. E. *Detecting spoofed packets*. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 1, pages 164–175. IEEE, 2003. doi:10.1109/DISCEX.2003.1194882.

The Open Group. *The UNIX System*. 2012. Accessed 17th June 2015.
URL `http://www.unix.org/`

The R Foundation. *What is R?* January 2016. Accessed 3rd February 2016.
URL `https://www.r-project.org/about.html`

Thomas, C. H., Charles, L. E., Ronald, R. L., and Clifford, S. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 3rd edition, 2009. ISBN 978-0-26-203293-3.

Trend Micro. *WORMMSBLAST.A*. August 2003. Accessed 27th January 2016. URL `https://www.trendmicro.com/vinfo/us/threat-encyclopedia/archive/malware/worm_msblast.a`

Tseng, Y.-C., Ni, S.-Y., Chen, Y.-S., and Sheu, J.-P. *The broadcast storm problem in a mobile ad hoc network*. *Wireless Networks*, 8(2-3):153–167, 2002. doi:10.1023/A:1013763825347.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. *Simultaneous multithreading: Maximizing on-chip parallelism*. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995. doi:10.1145/223982.224449.

Underwood, K. *FPGAs vs. CPUs: trends in peak floating-point performance*. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180. ACM, 2004. doi:10.1145/968280.968305.

Van Loan, C. *Computational frameworks for the fast Fourier transform*. SIAM, 1992. ISBN 978-0-89-871285-8.

Videla, A. and Williams, J. J. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012. ISBN 978-1-93-518297-9.

Vinoski, S. *Advanced message queuing protocol*. *IEEE Internet Computing*, 10(6), 2006. doi:10.1109/MIC.2006.116.

Walter, J. *Flame attacks: Briefing and indicators of compromise*. *McAfee Labs Report*, 2012.

Wang, B., Wang, B., and Xiong, Q. *The comparison of communication methods between user and kernel space in embedded Linux*. In *Computational Problem-Solving (ICCP), 2010 International Conference on*, pages 234–237. IEEE, 2010. ISBN 978-981-08-6322-7.

Welch, J. L. and Lynch, N. *A new fault-tolerant algorithm for clock synchronization*. *Information and Computation*, 77(1):1–36, 1988. doi:10.1016/0890-5401(88)90043-0.

Willinger, W., Paxson, V., and Taqqu, M. S. *A practical guide to heavy tails: statistical techniques and applications*. Birkhauser, 1998. ISBN 978-3-76-433951-7, 27–53 pages.

Winett, J. M. *Rfc 147: Definition of a socket*. 1971. Accessed 15th November 2017. URL `https://tools.ietf.org/html/rfc147`

Xilinx. *Spartan-6 FPGA Packaging and Pinouts*. 2009. Accessed 23th June 2015. URL `https://www.xilinx.com/support/documentation/user_guides/ug385.pdf`

Xilinx. *Spartan-6 FPGA Data Sheet: DC and Switching Characteristics*. 2011. Accessed 24th July 2015.
URL `https://www.xilinx.com/support/documentation/data_sheets/ds162.pdf`

Xilinx. *Spartan-3E FPGA Family Data Sheet*. July 2013. Accessed 8th June 2016.
URL `http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf`

Xilinx. *Spartan-6 FPGA Configuration*. 2014. Accessed 23th June 2015.
URL `http://www.xilinx.com/support/documentation/user_guides/ug380.pdf`

Yeh, T.-Y. and Patt, Y. N. *Two-level adaptive training branch prediction*. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991. doi:10.1145/123465.123475.

Yen, T.-F. and Reiter, M. K. *Traffic aggregation for malware detection. Detection of Intrusions and Malware, and Vulnerability Assessment 5th International Conference*, pages 207–227, 2008.

Yu, S., Zhou, W., Jia, W., Guo, S., Xiang, Y., and Tang, F. *Discriminating ddos attacks from flash crowds using flow correlation coefficient. Parallel and Distributed Systems, IEEE Transactions on*, 23(6):1073–1080, 2012. doi:10.1109/TPDS.2011.262.

Zemcik, P. *Hardware acceleration of graphics and imaging algorithms using FPGAs*. In *Proceedings of the 18th spring conference on Computer graphics*, pages 25–32. ACM, 2002. doi:10.1145/584458.584463.

Zhang, W., Teng, S., and Fu, X. *Scan attack detection based on distributed cooperative model*. In *Computer Supported Cooperative Work in Design, 2008. CSCWD 2008. 12th International Conference on*, pages 743–748. IEEE, 2008. doi:10.1109/CSCWD.2008.4537071.

Zhang, Z., Li, J., Manikopoulos, C., Jorgenson, J., and Ucles, J. *Hide: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification*. In *IEEE Workshop on Information Assurance and Security*, pages 85–90. 2001.

Zheng, C. and Thain, D. *Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. VTDC '15 Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, pages 31–38, 2015. doi:10.1145/2755979.2755984.

Zhenqi, W. and Xinyu, W. *Netflow based intrusion detection system*. In *MultiMedia and Information Technology, 2008. MMIT'08. International Conference on*, pages 825–828. IEEE, 2008. doi:10.1109/MMIT.2008.213.

Zhuravlev, S., Blagodurov, S., and Fedorova, A. *Addressing shared resource contention in multicore processors via scheduling*. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 129–142. ACM, 2010. doi:10.1145/1736020.1736036.

Zukowski, M., Heman, S., Nes, N., and Boncz, P. *Super-scalar RAM-CPU cache compression*. In *ata Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006. doi:10.1109/ICDE.2006.150.

A

NetFlow Version 5 Packet Formats

Figure A.1: NetFlow Version 5 Packet Header Format

Bytes	Contents	Description
0-1	Version	NetFlow export format version number
2-3	Count	Number of flows exported in this packet (1-30)
4-7	System Uptime	Current time in milliseconds since the export device booted
8-11	Unix Seconds	Current count of seconds since 0000 UTC 1970
12-15	Unix Nanoseconds	Residual nanoseconds since 0000 UTC 1970
16-19	Flow Sequence	Sequence counter of total flows seen
20	Engine Type	Type of flow-switching engine
21	Engine ID	Slot number of the flow-switching engine
22-23	Sampling Interval	First two bits hold the sampling mode; remaining 14 bits hold the sampling mode; remaining 14 bits hold value of sampling interval

Figure A.2: NetFlow Version 5 Record Format

Bytes	Contents	Description
0-3	Source Address	Source IP address
4-7	Destination Address	Destination IP address
8-11	Next Hop	IP address of next hop router
12-13	Input Interface	SNMP index of input interface
14-15	Output Interface	SNMP index of output interface
16-19	Packet Count	Packets in the flow
20-23	Layer 3 Packet Count	Total number of Layer 3 bytes in the packets of the flow
24-27	Start of Flow	SysUptime at start of flow
28-31	End of Flow	SysUptime at the time the last packet of the flow was received
32-33	Source Port	TCP/UDP source port number or equivalent
34-35	Destination Port	TCP/UDP destination port number or equivalent
36	Padding	Unused (zero) bytes
37	TCP Flags	Cumulative OR of TCP flags
38	Protocol	IP protocol type (for example, TCP = 6; UDP = 17)
39	Type of Service	IP type of service (ToS)
40-41	Source AS Number	Autonomous system number of the source, either origin or peer
42-43	Destination AS Number	Autonomous system number of the destination, either origin or peer
44	Source Mask	Source address prefix mask bits
45	Destination Mask	Destination address prefix mask bits
46-47	Padding	Unused (zero) bytes

B

FPGA 6 Input Gate Logic Results

Table B.1: Look-Up Table for Figure 2.11

Input A	Input B	Input C	Input D	Input E	Input F	Output
0	0	0	0	0	0	1
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	0	1	1	1
0	0	0	1	0	0	1
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	1	1	1	1
0	0	1	0	0	0	1
0	0	1	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	1	1	1
0	0	1	1	0	0	1
0	0	1	1	0	1	1

0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	1
0	1	0	0	0	1	1
0	1	0	0	1	0	1
0	1	0	0	1	1	1
0	1	0	1	0	0	1
0	1	0	1	0	1	1
0	1	0	1	1	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	0	1	1
0	1	1	0	1	0	1
0	1	1	0	1	1	1
0	1	1	1	0	0	1
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	0	0	1	1
1	0	0	0	1	0	1
1	0	0	0	1	1	1
1	0	0	1	0	0	1
1	0	0	1	0	1	1
1	0	0	1	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	0
1	0	1	1	1	1	1
1	1	0	0	0	0	0

1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	0	1	0	1	0
1	1	0	1	1	0	0
1	1	0	1	1	1	1
1	1	1	0	0	0	1
1	1	1	0	0	1	1
1	1	1	0	1	0	1
1	1	1	0	1	1	1
1	1	1	1	0	0	1
1	1	1	1	0	1	1
1	1	1	1	1	0	1
1	1	1	1	1	1	0

C

The Process Packet Template

```
1  // ------------------------------------------------------------------
2  //
3  // process_packet: This source exists to process packets for
4  //                 forwarding out ZMQ
5  //
6  // ------------------------------------------------------------------
7
8  // ------------------------------------------------------------------
9  // Includes
10 // ------------------------------------------------------------------
11
12 #include "process_packet.h"
13
14 >>>INCLUDES<<<
15
16 // ------------------------------------------------------------------
17 // Globals
18 // ------------------------------------------------------------------
19
20 >>>OFFSETS<<<
21
22 >>>GLOBALS<<<
23
24 // ------------------------------------------------------------------
25 // Functions
26 // ------------------------------------------------------------------
27
```

```
28 // ---------------------------------------------------------------------
29 //
30 int32_t get_hw_conf(uint8_t **conf_bff)
31 {
32     *conf_bff = conf;
33
34     return conf_count;
35 }
36
37
38
39 // ---------------------------------------------------------------------
40 // Process packet according to templates and send data to ZMQ
41 int32_t process_packet(uint8_t *pkt, int32_t len)
42 {
43     int32_t record_count = len/conf_size;
44     int32_t record_itteration;
45
46     int32_t zmq_buffer_len = 0;
47     uint8_t *zmq_bff_start = malloc(ZMQ_BUFFER_LEN);
48     uint8_t *zmq_buffer    = zmq_bff_start;
49
50     uint8_t *packet;
51
52     memset(zmq_bff_start, 0, ZMQ_BUFFER_LEN);
53
54     for(record_itteration = 0 ;
55         record_itteration < record_count ;
56         record_itteration++)
57     {
58         packet = pkt;
59         //printf("\n");
60 >>>PROCESS<<<
61         pkt += conf_size;
62     }
63
64     send_zmq_data(zmq_bff_start, zmq_buffer_len);
65     free(zmq_bff_start);
66
67     return 0;
68 }
```

D

Storm Worm Bait Mail Subjects and Attachment Names

THESE are the confirmed subjects of emails used for baiting potential users into running the executable required to install the Storm Worm software on a Microsoft OS. These are directly copied from the subject lines of the bait emails used (Symantec, 2007a). These are listed below:

- A killer at 11, he's free at 21 and kill again!
- U.S. Secretary of State Condoleezza Rice has kicked German Chancellor Angela Merkel
- British Muslims Genocide
- Naked teens attack home director.
- 230 dead as storm batters Europe.
- Re: Your text
- Radical Muslim drinking enemies's blood.

- Chinese/Russian missile shot down Russian/Chinese satellite/aircraft
- Saddam Hussein safe and sound!
- Saddam Hussein alive!
- Venezuelan leader: "Let's the War beginning".
- Fidel Castro dead.
- If I Knew
- FBI vs. Facebook
- Love birds
- Touched by Love

The confirmed names of executables attached to these bait emails containing the Storm Worm trojan software are listed below:

- Postcard.exe
- ecard.exe
- FullVideo.exe
- Full Story.exe
- Video.exe
- Read More.exe
- FullClip.exe
- GreetingPostcard.exe
- MoreHere.exe
- FlashPostcard.exe
- GreetingCard.exe
- ClickHere.exe

- ReadMore.exe
- FlashPostcard.exe
- FullNews.exe
- NflStatTracker.exe
- ArcadeWorld.exe
- ArcadeWorldGame.exe
- with_love.exe
- withlove.exe
- love.exe
- frommetoyou.exe
- iheartyou.exe
- fck2008.exe
- fck2009.exe

E

Mirai's IPv4 Subnetwork Exclusion List

THIS is the list of excluded subnetworks for infection defined in the Mirai malware code accompanied by which entity the subnetwork belongs to:

- 127.0.0.0/8 Loopback
- 0.0.0.0/8 Invalid address space
- 3.0.0.0/8 General Electric (GE)
- 15.0.0.0/7 Hewlett-Packard (HP)
- 56.0.0.0/8 US Postal Service
- 10.0.0.0/8 Internal network
- 192.168.0.0/16 Internal network
- 172.16.0.0/14 Internal network
- 100.64.0.0/10 IANA NAT reserved

- 169.254.0.0/16 IANA NAT reserved
- 198.18.0.0/15 IANA Special use
- 224.0.0.0/8 Multicast
- 6.0.0.0/7 Department of Defense
- 11.0.0.0/8 Department of Defense
- 21.0.0.0/8 Department of Defense
- 22.0.0.0/8 Department of Defense
- 26.0.0.0/8 Department of Defense
- 28.0.0.0/7 Department of Defense
- 30.0.0.0/8 Department of Defense
- 33.0.0.0/8 Department of Defense
- 55.0.0.0/8 Department of Defense
- 214.0.0.0/7 Department of Defense

F

Simple Template File for Bolvedere Publisher

Listing F.1: Template File Referred to By Configuration File for Building a Bolvedere System

```
# This file deals with the default NetFlow v9 flows sent by Softflowd
# for the simple conficker and port scan detectors.
# NAME, flow_id, len

# src IP, dst IP, src Port, dst Port, pkt count, byte count

IN_BYTES, 1, 4
{
    *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 2); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 3); zmq_buffer++;
    zmq_buffer_len += 4;
    printf("[Data] bytes communicated       : %08X\n",
           ntohl(*(uint32_t *)relevant_data));
}

IN_PKTS, 2, 4
{
    *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 2); zmq_buffer++;
    *zmq_buffer     = *(relevant_data + 3); zmq_buffer++;
    zmq_buffer_len += 4;
```

```
25      printf("[Data] packets communicated     : %08X\n",
26             ntohl(*(uint32_t *)relevant_data));
27  }
28
29  L4_SRC_PORT, 7, 2
30  {
31      *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
32      *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
33      zmq_buffer_len += 2;
34      printf("[Data] L4 source port           : %04X\n",
35             ntohs(*(uint16_t *)relevant_data));
36  }
37
38  IPV4_SRC_ADDR, 8, 4
39  {
40      *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
41      *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
42      *zmq_buffer     = *(relevant_data + 2); zmq_buffer++;
43      *zmq_buffer     = *(relevant_data + 3); zmq_buffer++;
44      zmq_buffer_len += 4;
45      printf("[Data] IPv4 source address      : %08X\n",
46             ntohl(*(uint32_t *)relevant_data));
47  }
48
49  L4_DST_PORT, 11, 2
50  {
51      *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
52      *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
53      zmq_buffer_len += 2;
54      printf("[Data] L4 destination port      : %04X\n",
55             ntohs(*(uint16_t *)relevant_data));
56  }
57
58  IPV4_DST_ADDR, 12, 4
59  {
60      *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
61      *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
62      *zmq_buffer     = *(relevant_data + 2); zmq_buffer++;
63      *zmq_buffer     = *(relevant_data + 3); zmq_buffer++;
64      zmq_buffer_len += 4;
65      printf("[Data] IPv4 destination address : %08X\n",
66             ntohl(*(uint32_t *)relevant_data));
67  }
68
69  FLOW_SOURCE_ID, 512, 4
70  {
71      *zmq_buffer     = *(relevant_data + 0); zmq_buffer++;
72      *zmq_buffer     = *(relevant_data + 1); zmq_buffer++;
73      *zmq_buffer     = *(relevant_data + 2); zmq_buffer++;
74      *zmq_buffer     = *(relevant_data + 3); zmq_buffer++;
75      zmq_buffer_len += 4;
76      printf("[Data] Flow Source ID           : %08X\n",
77             ntohl(*(uint32_t *)relevant_data));
78  }
```

G

Graphed Statistics for Real-World Dataset

Below one can find the graphs depicting the information found in Tables 7.1 and 7.2.

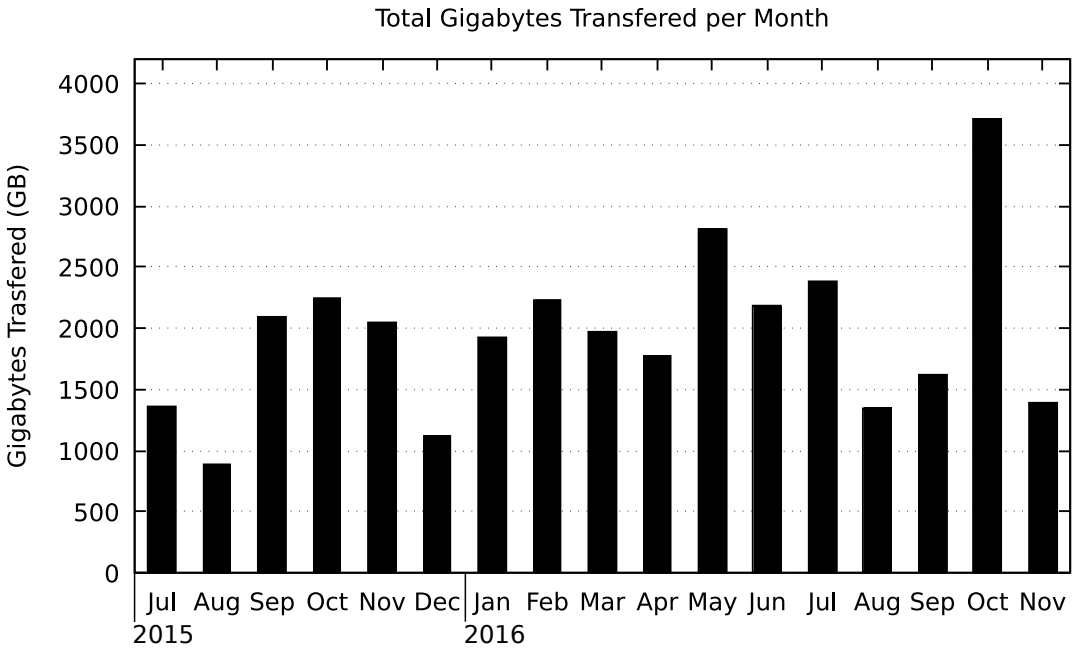

Figure G.1: Number of Gigabytes Transferred per Month

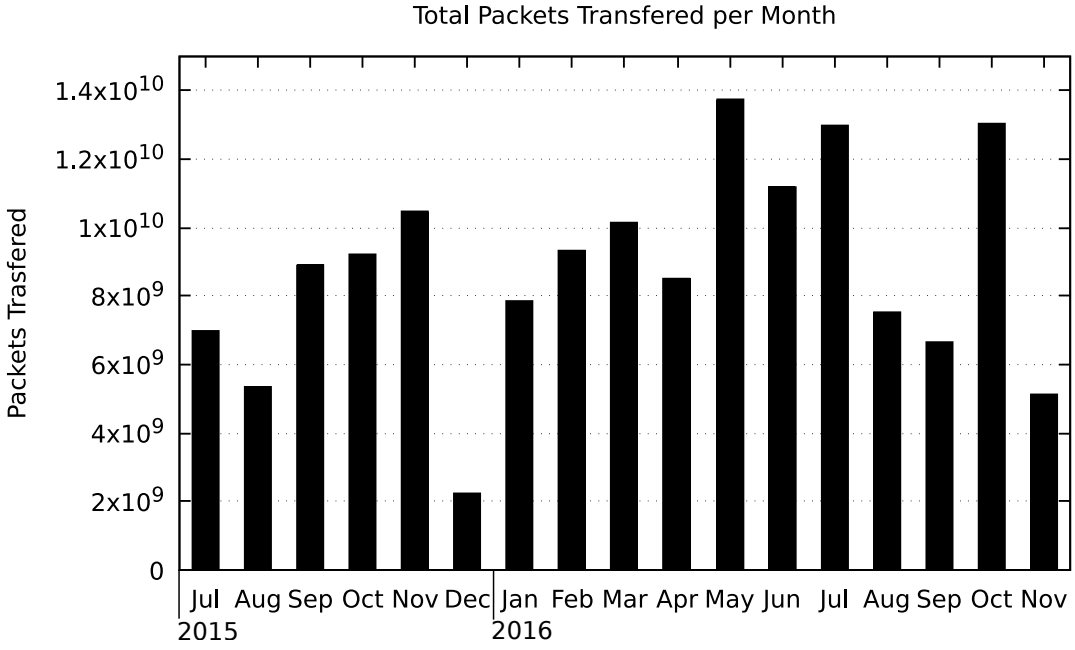

Figure G.2: Number of Packets Transferred per Month

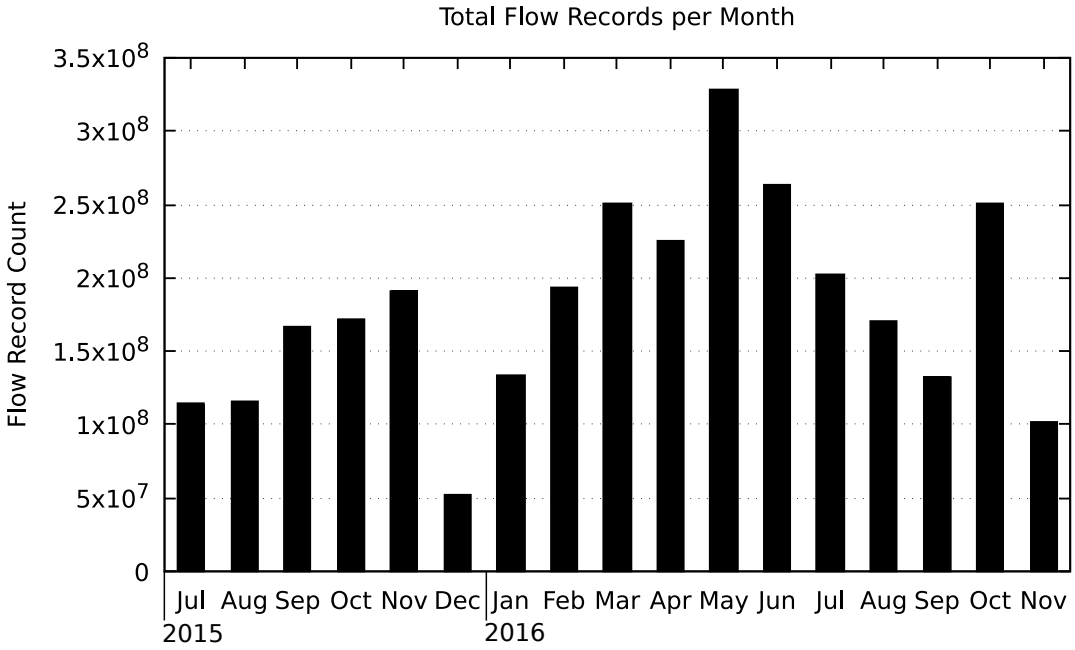

Figure G.3: Number of Flow Records per Month

H

Online Resource Access

Access to online resources is not public and as such should be requested from the author of this document. To contact the author to request access please send an email to phd@eskerfall.com. The repository contains this document and both software and hardware implementations discussed in this research. This repository resides on Bitbucket[1] and can be cloned through the relevant address below where the username is defined by the reader's registered Bitbucket account:

https://username@bitbucket.org/twelfthletter/phd

[1] https://www.bitbucket.org