An SDN-based firewall shunt for data-intensive science applications

Simeon Miteff

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, May 2016

To Juanita

Abstract

Data-intensive research computing requires the capability to transfer files over long distances at high throughput. Stateful firewalls introduce sufficient packet loss to prevent researchers from fully exploiting high bandwidth-delay network links [25]. To work around this challenge, the science DMZ design [19] trades off stateful packet filtering capability for loss-free forwarding via an ordinary Ethernet switch. We propose a novel extension to the science DMZ design, which uses an SDN-based firewall. This report introduces NFShunt, a firewall based on Linux's Netfilter combined with OpenFlow switching. Implemented as an OpenFlow 1.0 controller coupled to Netfilter's connection tracking, NFShunt allows the bypass-switching policy to be expressed as part of an iptables firewall rule-set. Our implementation is described in detail, and latency of the control-plane mechanism is reported. TCP throughput and packet loss is shown at various round-trip latencies, with comparisons to pure switching, as well as to a high-end Cisco firewall. Cost, as well as operations and maintenance aspects, are compared and analysed. The results support reported observations regarding firewall introduced packet-loss, and indicate that the SDN design of NFShunt is a technically viable and cost-effective approach to enhancing a traditional firewall to meet the performance needs of data-intensive researchers.

Declaration

I declare that this MSc dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of Witwatersrand, Johannesburg. It has not been submitted for any degree or examination in any other University.

Simeon Miteff
Wednesday 4th May, 2016

Acknowledgements

Thank you to: my wife Juanita, for her patience and encouragement, listening and proof reading; Professor Hazelhurst, for his invaluable guidance through my first research project; the SANReN team, CHPC, the CSIR and the South African Department of Science and Technology for funding my studies and purchasing lab equipment, as well as for allowing me to work on this project as part of my day-job. Finally, thank you to my friends, family and colleagues who supported me and showed interest in this work.

Contents

List of Figures

List of Tables

Glossary

ACL Access Control List (ACL) is a stateless packet filter typically provided by hardware switches and routers. 4, 14, 24, 25

API Application Program Interface (API) is a set of functions that form the interface to a software component. 25

ARP Address Resolution Protocol (ARP) is a protocol for resolving network layer addresses to link layer addresses. 80, 81

ASIC Application-Specific Integrated Circuit (ASIC) is a fixed-function integrated circuit designed for a specific application. 13, 16, 18, 19, 22–24, 39

COTS Commercial Off-The-Shelf (COTS) is a complete component that is commercially available. 38, 66, 70

DMZ demilitarised zone (DMZ) (in the context of computer networks) is a sub-network isolated by security measures designed to protect the hosts from threats originating on other networks, as well as protecting other networks from threats if the hosts in the DMZ are compromised. 3–6, 25, 35, 38, 53–56

DPDK Data Plane Development Kit (DPDK) is a library of packet processing software and associated hardware drivers intended for user-space switching. 17, 39

ECN Explicit Congestion Notification (ECN) is used to signal buffer utilisation between routers and end-hosts. 10, 11

FCAPS Fault, Configuration, AAA, Performance and Security management (FCAPS) is the ITU-T Recommendation M.3400, TMN Management Functions [38]. 68

FDT Fast Data Transfer (FDT) is the use of systems and tools optimised for the transfer of large data sets, often over large distances. 2, 5, 6, 9–12, 17, 26, 30, 31, 36, 53–55, 66, 67, 71

FOSS Free and Open Source Software (FOSS) is free software available under an Open Source copyright license. 20, 26, 38, 70, 73

FPGA Field Programmable Gate Array (FPGA) is an integrated circuit technology in which the electrical logical functions are configured and re-configured by software. 6, 22

FTP File Transfer Protocol (FTP) is an application layer protocol for transferring files between hosts on the Internet. 3, 11, 30

HPC High Performance Computing (HPC) is the use of super-computers and computing clusters to execute computationally intensive work-loads. 1, 2, 10, 25, 74

HTTP Hypertext Transfer Protocol (HTTP) is the application-layer protocol of the World Wide Web. 25, 32

ICMP Internet Control Message Protocol (ICMP) is an Internet transport-layer protocol responsible for error messages and testing. 80, 81

IP Internet Protocol (IP) refers to a version of the network layer protocol common to the Internet. 2, 29, 56, 58

IPS Intrusion Prevention System (IPS) is a system that scans network traffic for intrusions and actively destroys offending connections. 4

JSON JavaScript Object Notation (JSON) is a light-weight encoding for documents which is both human and machine readable. 49

LAN Local Area Network (LAN) is a computer network local to a physical building or campus of buildings. 4, 14, 16

LHC Large Hadron Collider (LHC) is the world's largest particle accelerator - housed at CERN. 1

MAC Media Access Control (MAC) is a sub-layer of the data-link layer (layer 2) which is concerned with functions such as addressing, media access control and error detection. 25, 60, 80

MTU Maximum Transmission Unit (MTU) of a protocol layer is the maximum size of a data unit for that layer. 56, 59, 71

NIC Network Interface Controller (NIC) is the compute hardware component that provides an interface to a network. 13, 56–60

NPU Network Processing Unit (NPU) is a multi-core processor providing parallelism intended specifically for a hardware network forwarding plane. 6, 13, 18, 22, 24

OPN Optical Private Network (OPN) is the high-speed data network between CERN and the LHC's tier-one processing centres. 1

OVS open vSwitch (OVS) is an Open Source virtual switch that implements OpenFlow. 20–22, 39–41, 66, 75, 78–81

REST Representational State Transfer (REST) is a stateless architecture for building remote procedure calls on top of HTTP. 25

RISC Reduced Instruction Set Computing (RISC) is a microprocessor architecture based on a simple instruction set and optimised for high speed code execution. 18

RTT Round-Trip Time (RTT) is the bi-directional delay in a network path between two nodes. 10, 57, 58, 60, 61, 63, 67

SDN Software Defined Networking (SDN) is a network architecture where "the control and data planes are decoupled, network intelligence and state are logically centralised, and the underlying network infrastructure is abstracted from the applications" [30]. 6, 16, 18, 20, 22, 24, 36, 54, 73–75

SKA Square Kilometre Array (SKA) will be the world's largest radio-obsevatory, with radio-telescopes planned to be built in South Africa and Australia. 1, 57

SNMP Simple Network Management Protocol (SNMP) is a datagram-based protocol for monitoring an managing network elements. 25, 68

TCAM Ternary Content-Addressable Memory (TCAM) is a type of memory specialised for rapid table lookups based on addresses, including the option for wild-card bits. 24

TCP Transmission Control Protocol (TCP) is the reliable connection-oriented network transport protocol used by the majority of Internet applications. 2–6, 9–11, 14, 16, 22, 23, 25, 29, 32, 40, 41, 45, 47, 49, 54–61, 66, 67, 71, 73, 74

TLS Transport Layer Security (TLS) is a session layer protocol that provides encryption and authentication for transport layer connections. 69

XML Extensible Markup Language (XML) is an encoding for documents which is intended to be both human and machine readable. 47

Publications

Parts of this dissertation appeared in a conference paper entitled *NFShunt: a Linux firewall with OpenFlow-enabled hardware bypass*, at the IEEE Conference on Network Function Virtualization and Software Defined Networks 2015 [55].

Chapter 1

Introduction

1.1 Importance of data-intensive science

Gordon Bell argues that data-intensive computing is the basis for a new paradigm of science [36]. The idea of e-Science is that data-exploration is a new scientific method that unifies theory, experimentation and simulation. Cyber-infrastructure is therefore critical to modern science, and the network is the central component that moves data between computing resources (and indirectly to the researchers).

The Large Hadron Collider (LHC) [14] serves as an example of a scientific instrument and its associated experiments with data-intensive infrastructure requirements. The distributed manner in which data produced at the LHC is analysed led to new architectures for network provisioning and security, for example, the LHC Optical Private Network (OPN) [8]. The distributed processing strategy itself was developed to cope with unprecedented volumes of scientific data, where network transfers were measured in tens of gigabits per second [75].

Early work to understand the network needs of the Square Kilometre Array (SKA) [21] radio-telescope has identified a new large-data frontier, this time in the order of hundreds of gigabits per second [39]. Just as a novel approach was applied for the needs of the LHC, so will each successive data-intensive experiment need to find efficient ways to distribute, store and analyse data.

Gorton et al. list astronomy and cyber-security as applications that exhibit the characteristics of being both data and computationally intensive. Since High Performance Computing (HPC) facilities are themselves exposed to cyber-security threats [32], the need to apply cyber-security measures to HPC compounds the total complexity of data-intensive applications.

McMahon and Hutchison [52] suggest that standard security measures (such as a firewall) can be applied to protect HPC infrastructure, and propose a security architecture based on such standard components. Quite contrary to this work, ESNet [25] reports that packet loss in high speed networking is often caused by standard firewalls.

According to ESNet [25], firewalls are supposed to behave transparently for legitimate traffic but, due to scalability constraints in the special case of large network transfers, they do not do so. When high latency transfers are attempted through such firewalls, the Transmission Control Protocol (TCP) stack throttles a connection's window size because packet loss is interpreted as network congestion and, as a result the data throughput achieved is much less than the nominal network capacity. Given the high cost of international connectivity, this is a practical and important efficiency problem for globally distributed data-intensive research infrastructures.

Since network latency is bound by the speed of light, the solution is either to mitigate the effect of packet loss on applications by adapting their use of the network, or to eliminate the cause of the loss itself.

If the critical role of high speed networks in data-intensive computing is at odds with deployed network security measures, then it is important to study this apparent conflict, and propose solutions.

1.2 State of the art

High performance network security and Fast Data Transfer (FDT) are both subjects of active research. This section describes the relevant state of the art.

1.2.1 Hardware acceleration

According to ESNet [22], hardware firewalls are capable of stateful filtering (see section 2.4) of Internet Protocol (IP) packets at line rate when employing parallelised architecture, which spreads individual network flows (unique connections between two network endpoints) over a number of specialised network processor cores.

In a typical hardware firewall, each network processor's peak performance is less than the total throughput of the firewall [25]. Normally individual flow throughput is limited elsewhere (often at one of the endpoints) to the extent that this architecture scales well for traffic composed of many small flows (such as typical Internet traffic).

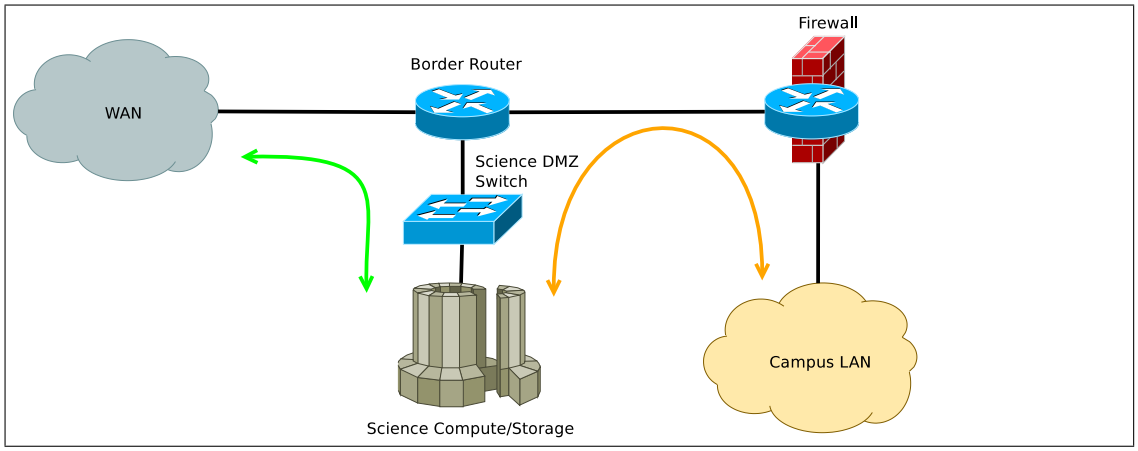

Figure 1.1: Example science DMZ network diagram

1.2.2 FDT-optimized tools

One way to work around poor TCP performance is to spread data transfers over a number of (lower rate) parallel connections (striping). This technique was developed to compensate for end-host TCP window size tuning problems, but could also allow for higher total throughput in high latency transfers exposed to packet loss.

GridFTP [2] is an extension of the File Transfer Protocol (FTP) that employs parallel TCP connections to improve throughput.

1.2.3 Simplified filtering

Another solution is to engineer networks to be suitable for high performance TCP by eliminating the firewall itself (and therefore the lossy component). ESNet proposes a network design called the science demilitarised zone (DMZ) [19].

The idea of a science DMZ is to create a small, fast subnetwork at the edge of the network at each institution (e.g., universities, research labs, etc.), which is devoid of middle-boxes (in other words, it is connected before the firewall). High latency transfers (originating from distant endpoints) are therefore not subjected to the performance-degrading effect of packet loss caused by middle-boxes, while transfers from inside the institutional network (and therefore beyond the border firewall) are low latency – therefore not subject to the same effect.

Figure 1.1 illustrates an example of this design: network traffic represented by the green arrow traverses the "clean" network, while the orange line represents local (low-latency) traffic.

It is argued that computing resources in the science DMZ can be adequately protected

from intruders by making use of simplistic (but scalable) router interface Access Control Lists (ACLs), combined with host security measures [19].

As side benefits, this architecture also alleviates operational conflicts resulting from attempts to shoehorn a firewall security policy designed for protecting enterprise Local Area Networks (LANs) to serve the unusual networking needs of scientific computing applications, and encourages efficient scaling of network elements for the specific needs of each class of network end-host (enterprise LAN versus science computing infrastructure).

1.2.4 Intrusion prevention with shunting

A third alternative is to separate the network traffic belonging to data-intensive science applications from other flows, and only apply security measures to the remainder. This approach makes use of a custom hardware switch called a shunt [31], which is programmed to either bypass or forward traffic via an Intrusion Prevention System (IPS).

An analysis of packet traces from institutions typically running data-intensive applications shows that shunting (the mechanism of hardware bypass-switching of network traffic otherwise processed in software) can reduce the amount of network traffic forwarded via the "slow path" significantly, allowing the system to run at the full speed of the institution's network connection.

More recently, SciPass enhanced shunting to take advantage of OpenFlow-based hardware switching [5]. While the original shunt work aimed to address the problem of IPS scalability, SciPass leveraged bypass switching for the enhancement of science DMZs.

1.3 Limitations of the state of the art

1.3.1 Load balancing

According to ESNet [25], in data-intensive science applications where endpoints are optimised to allow transfers at speeds close to the limit of the underlying network, a hardware firewall architecture which relies on load-balancing (increasing aggregate capacity through parallel processing) over multiple network processors drops packets, resulting in poor TCP performance with high-latency flows.

1.3.2 Specialised protocols

Specialised applications, employing the parallel-TCP connections strategy to improve throughput, suffer from operational and adoption problems:

- Ports and protocols used by non-standard applications are often blocked by the standard policy configured on the same firewall - the limitations of which they are designed to work around [44].

- We speculate that applications using existing (albeit lower performance) protocols are more convenient for some end-users. For example, Secure Copy Protocol (SCP) (a single-flow TCP based file transfer application) is installed by default on most servers, and is integrated with the operating system's authentication and authorisation mechanism [17].

1.3.3 Simplified filtering

A firewall is described as an insurance policy for the manager or executive at an institution accountable for cyber-security [18]. It is understandable then that some institutions are reluctant to adopt the simplified design of the science DMZ (which does not make use of a firewall in the usual sense).

Since no network security measure is absolutely effective, in the case of a breach, it could be difficult for the responsible officer to explain to the board (or relevant authority) why there was no firewall in place.

Given this human bias against simplified filtering, making use of a firewall that delivers suitable performance for FDT is perhaps the path of least institutional resistance.

1.3.4 Shunting

The use of a custom hardware platform for shunting allows for flexibility in the exact operation of the shunt itself, but limits the practicality of wide-spread deployment.

Using standardised off-the-shelf hardware for the shunting component would permit convenient substitutions, and take advantage of the economy of scale available due to mass production of such components. SciPass [5] enhances shunting by employing a standard hardware fast-path.

Shunting and SciPass both address scalability of intrusion detection. SciPass includes the capability to shunt trusted connections around a traditional firewall based on IDS signatures, but neither system attempts to implement standard firewall interfaces or semantics with a single policy for both slow and fast path switching.

1.4 A new approach

Previous work with OpenFlow in the FDT context has applied it to the management of (stateless) access control lists [72], and the implementations of shunting focused on intrusion detection applications [5,31]. Similarly, previous work to add hardware-offload acceleration for the Linux Netfilter firewall (see section 2.9) has relied on a custom kernel module communicating with specialised hardware, such as a Field Programmable Gate Array (FPGA) [15] or a Network Processing Unit (NPU) [1].

We research a shunting strategy for firewalls, which on the forwarding plane is similar to the science DMZ, but the addition of a control plane driven by a stateful firewall ruleset results in functionality very similar to a traditional firewall. The contribution of our research is the hybrid shunting firewall design, a prototype implementation (NFShunt) and an analysis of the prototype's performance.

Our hybrid approach has an advantage of both the science DMZ and stateful firewall solutions, in that it would eliminate some of the trade-off between security and performance, and be implementable in real-world applications by making use of existing, well-understood and tested off-the-shelf components. We use OpenFlow to decouple the control logic of the firewall from switch hardware — the central principle of Software Defined Networking (SDN) (see section 2.5).

It is important to note that stateless bypass switching inevitably compromises the ability of the system to check the validity of packet headers against expected protocol state, as well as other packet filtering capabilities (such as application-layer inspection), subsequent to shifting the traffic to the fast-path. This limitation is common to our prototype hybrid firewall and similar designs, such as SciPass. Table 1.1 summarises the performance and security trade-offs offered by the hybrid approach and its alternatives. We only consider a use-case involving single or small numbers of TCP connections.

The research does not address a comprehensive threat-model for science DMZs, cater for non-TCP applications, or examine connection-rate performance.

Approach	**Performance**	**Security**
Access Control Lists	High	Low
Traditional firewall	Low	High
Hybrid (shunting) firewall	High	Medium

Table 1.1: Comparison of Science DMZ protection mechanims

1.5 Overview of the thesis

This report is structured as follows:

Chapter 2 provides the background to our research: we review relevant technologies and critically analyse related work in the literature. Chapter 3 describes the research problem, and our prototype firewall's design and implementation. Chapter 4 covers our research method for experimental evaluation, and reports the results of our experiments. Chapter 5 presents our analysis of the prototype implementation and evaluation. Finally, in chapter 6, we draw conclusions from our study and explore future work.

Chapter 2

Background

2.1 Introduction

The performance of network applications in the presence of network packet filtering is a function of interrelated effects: the transport layer protocol implementation (e.g., Transmission Control Protocol (TCP)) reacts to network conditions, that are in turn affected by the architecture of the network packet filter. It is, therefore, necessary to understand the theoretical background and practical implementation of packet forwarding and filtering, in addition to the network traffic profile of the applications of relevance to the research.

When examining different implementations of packet forwarding and filtering devices, it is important to consider the separation between the following: the mechanisms that handle individual packets, namely the forwarding plane (or data path); and the control mechanisms that maintain data structures used by the forwarding plane to make decisions about how to handle packets, namely the control plane.

Our hybrid firewall combines a bypass switching (shunting) strategy with the use of a standard software interface for controlling the hardware component. Each of these technologies (as well as existing work to combine them in similar architectures) are explored in the sections that follow.

2.2 Fast data transfer

Network use-cases for data intensive science require Fast Data Transfer (FDT). More specifically:

1. Large (sometimes Peta-byte-sized) data sets are moved between different locations

on the network [36]. For example, raw experimental data may be collected by sensors or instruments at one location, processed at a second location (on a supercomputer), and the output then analysed by scientists at a third location.

2. Unlike commodity Internet traffic, which is typically comprised of many simultaneous (relatively low speed) TCP flows, the data is often transferred between single endpoint systems, and maximum throughput is required for single TCP sessions.

The distribution of connection data volumes for various types of network traffic follows a heavy-tailed distribution [41], meaning that a small number of connections account for a large proportion of the total data transferred, while the majority of the connections transfer a small amount of data. Gonzales et al. [31] apply their shunting technique to large connections in six different network traffic data sets gathered from universities, research labs and a super computing centre. They find that the heavy tail flow effect is even more prevalent where the use of the network tends towards the specialised applications of High Performance Computing (HPC).

2.2.1 TCP performance challenges

The TCP protocol interprets packet loss as congestion on the network path between end-points, resulting in packets being dropped from router interface queues, and uses this information to adjust the rate at which data is transmitted to match the capacity available.

Unfortunately, queue drops resulting from link congestion are not necessarily the only causes of loss. Using the model for TCP performance developed by Mathis et al. [49], it can be seen that the combination of modest packet loss rates on high bandwidth-latency product links results in pathological inefficiency on links that are not fully utilised.

Among the variants of TCP, some implementations are adapted for high speed (for example: HighSpeed TCP [29], Scalable TCP [40] and FAST TCP [81]). Difficulties in the design of TCP congestion control algorithms are: achieving high throughput in diverse (and variable) network conditions while also maintaining *fairness* (the equal sharing of bottleneck link capacity among all TCP connections), and avoiding biases among connections according to Round-Trip Time (RTT).

Explicit Congestion Notification (ECN) extends TCP to distinguish between error drops and queue drops [70], which is promising for supporting FDT with non-congestive packet loss. While ECN is now commonly supported in both end-host operating systems, as well as network equipment, it must be enabled both on the hosts and on the underlying network to be effective. This constraint has been problematic for the use of ECN in long

distance inter-domain network applications: a recent study by Kühlewind et al. [42] showed that the majority of TCP connections on the Internet still cannot use ECN.

In addition to the bootstrapping problem between host and router support, the adoption of ECN has been further hampered by the interference of poorly considered firewall policies [79]. This is incongruous with the fact that ECN might otherwise permit the use of firewalls for FDT.

The latency constraint is often an unavoidable physical constraint that is a consequence of the global nature of modern collaborative big science projects. Clearly loss-free networks are critical for the use of single TCP connections in high-bandwidth applications.

2.2.2 Alternative protocols for FDT

Proposals exist for new protocols that perform well despite packet loss in high bandwidth-delay product networks.

One example of an application-layer protocol is GridFTP [2], which extends the File Transfer Protocol (FTP) to suit the needs of FDT. One GridFTP feature is the ability to establish multiple parallel TCP connections between a pair of servers. A pool of servers containing the same data can also be used in parallel (in *striping* mode), futher improving the scalability of GridFTP-based systems by accessing the independent storage backends simultaneously. GridFTP protocol is also used in the Globus Toolkit and the Globus Online service [3]. Parallel TCP connections can be effective at working around throughput limitations (due to packet loss and poor host tuning) because individual congestion windows are smaller while the aggregate throughput is increased.

Alternatives to TCP for large file transfers include various UDP-based protocols, such as UDT, and commercial products such as Aspera's FAST; MTP/IP from Data Expedition; and TIXEL's RWTP. Dart notes [17] that these protocols deal well with congestive packet loss, but their performance in high latency un-congested links (typical research network) is less clear.

Despite their advantages, alternative protocols face adoption challenges (similar to ECN). GridFTP has been successful in the grid computing community but is not without its own challenges, one being the need to allow non-standard ports on firewalls [44]. UDP-based protocols have been successful for commercial applications but have seen limited deployment in research networks.

We also note that proposals for alternative protocols and applications, compete for adoption, which suggests that no general solution exists. The efforts to propose and then promote alternative protocols are important and valid in their approach, but it is

also important to explore other solutions to the broader FDT problem.

Rather than adapting network applications to suit the network, the science DMZ architecture aims to engineer the network to support all (including non-optimised) applications. Our research focuses on the network-based approach to FDT.

2.3 High speed packet switching

Expansion of the Internet has required the evolution of network technologies to deliver higher speed networks. In this section we explore the state of the art in electronic network packet switching.

2.3.1 Software switching

Forwarding or switching of Internet Protocol packets is performed by a router (or gateway). In addition to making switching decisions (choosing an output interface for each packet), a router must also be able to translate between different network media. Therefore its functions must extend from the physical to the network layer. The basic steps common to all IP routing are:

1. Next-hop (route) lookup,
2. Decrementing the Time To Live (TTL) counter,
3. Recalculating the checksum,
4. Layer 2 header re-writing.

In addition to packet forwarding, a router itself must be able to act end-node to enable it to interact with other nodes for the purposes of administration and monitoring, building routing tables, error handling and so on. The functions of a router can, therefore, be divided into two areas: the forwarding plane and the control plane.

The complexity of control plane functions require software running on a general purpose microprocessor. Early routers also implemented the control and forwarding plane function as subroutines of the same software, as the processing power available was sufficient to support the data rates required. Network protocols were also rapidly evolving, making a software implementation well suited to updates and improvement. Consequently, the hardware architecture of IP routers were similar to that of general purpose computers, utilising a centralised memory and a shared peripheral bus [4].

As data rates on inter-networks increased, optimisations were made to the IP forwarding process, for example: commonly used routes were cached for faster routing decisions, and the actual packet forwarding process could be moved into an interrupt handler to avoid packets being delayed by process scheduling in the router operating system.

2.3.2 Hardware switching

When attempting to scale software IP forwarding to hundreds or even thousands of megabits per second, it becomes apparent that bus and memory bandwidths must be twice the total port capacity of the router, as each packet is copied twice during the routing process [4].

While it is now possible to achieve small-packet 10Gbps line-rate L2 and L3 switching with a small number of interfaces in software [83], we note that the bar for switching performance has been raised to 100Gbps.[1] The calibrated model developed by Meyer et al. [54] predicts that future single-flow performance of software packet processing will remain constrained by single-core performance. To overcome this, modern multi-gigabit networks rely on specialised packet forwarding hardware.

One of the simplest hardware optimisations is to off-load some of the processing tasks to fixed-function circuitry, such as verifying a packet checksum to the receiving network interface controller. Moving beyond this initially proved problematic because the Internet protocol was not designed with a hardware forwarding plane in mind. This led to the development of label switching as a simpler alternative to IP routing, and a means to encapsulate IP and other network and data-link layer protocols in transport networks [61].

Eventually, advances in integrated circuit technology permitted pure hardware forwarding to be implemented, including dedicated longest-prefix route look-up state machines. Most high speed routers now employ Application-Specific Integrated Circuits (ASICs) or occasionally Network Processing Units (NPUs) in the forwarding plane.

In order to keep ASICs simple enough to allow line-rate forwarding in practical implementations, certain features are not catered for in the hardware path, for example: IP options change the length of the header itself and are, therefore, "punted" to a software router implementation running on a general-purpose microprocessor (often the same processor running the router's control plane software). This hybrid fast/slow path

[1] 100Gbps Ethernet Network Interface Controllers (NICs) as well as 32-port 100Gbps 1-RU switches were available as of early 2015.

approach is reminiscent of route caching and re-appears in contemporary network designs (sometimes in combination with route caching).

One of the key advantages afforded by the stateless nature of the Internet protocol is that routing is an embarrassingly parallel workload. In a single router, multiple (distributed) copies of routing tables can, therefore, be used by more than one forwarding engine in parallel, thereby eliminating the need for shared access to memory.

2.4 Traditional firewall designs

Blocking certain connections and allowing others has become a popular measure to protect end-hosts (and consequently the users and organisations) connected to the Internet. This function is performed by a firewall [77].

The state machine used for route lookup (matching a header field against a data structure of values and masks) can be used to implement a simple packet filter (also known as an Access Control List (ACL)), with actions that specify whether a matching packet should be dropped or forwarded. Strictly speaking, a router with ACL capability is then also a firewall, however the capabilities of firewalls have surpassed simplistic packet filtering.

Stateful (or state-aware) firewalls take into account the state of the transport protocol connection, thereby protecting against a class of network attacks that cannot be blocked using (stateless) packet filters. We focus on TCP which, being connection-oriented, is inherently stateful. Figure 2.1 describes the connection states of a TCP session. Other state information associated with a TCP connection includes the congestion control algorithm (not usually a security concern) and sequence numbers, which could be checked in a firewall to defend against spoofing attacks [34]. A simple example of the advantage of stateful firewalling is a typical scenario for firewall configuration, namely the *TCP diode*. Suppose an organisation wishes to implement the following policy:

1. Hosts on the Local Area Network (LAN) (clients) can initiate TCP connections to web servers on the Internet (on port 80).

2. TCP connections from the Internet to the LAN are blocked.

In practice, the above policy would be translated into a firewall rule set. For a stateless packet filter, the following rules could be used (evaluated sequentially):

1. Forward packets where the protocol is TCP, direction is to the Internet, and the TCP destination port is 80.

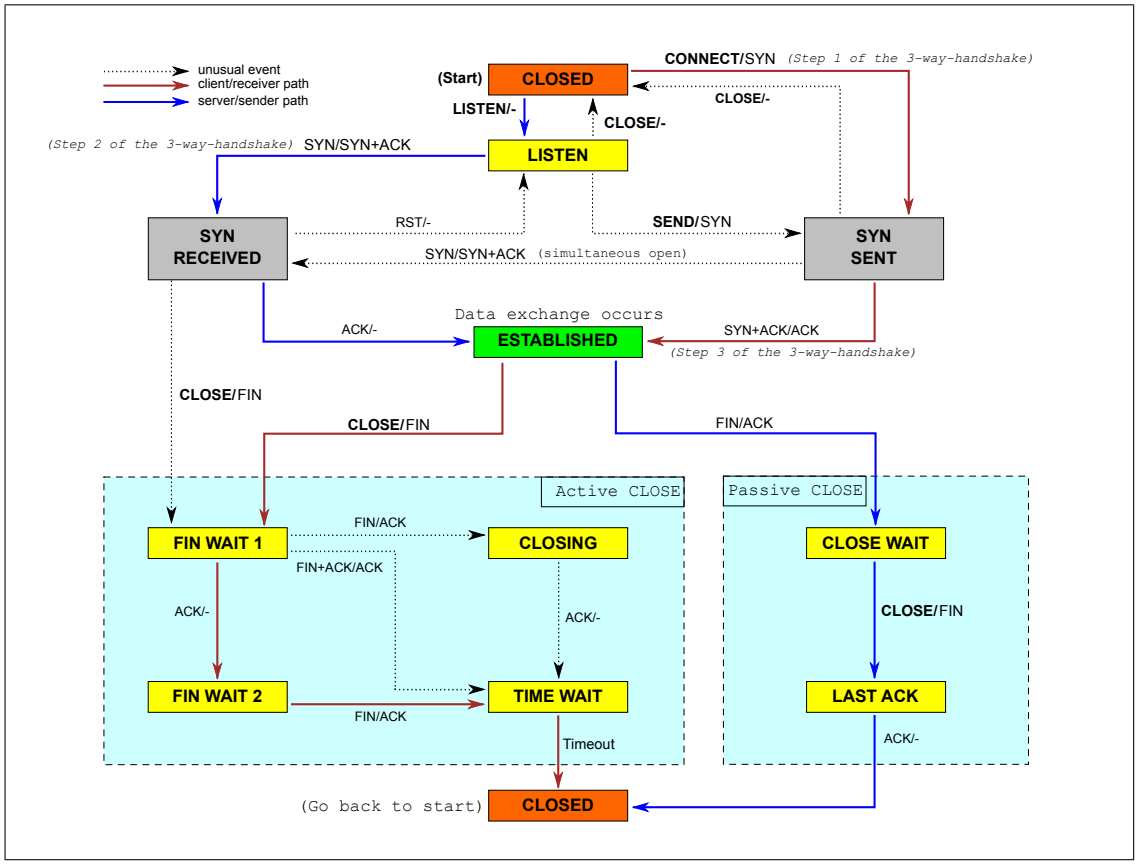

Figure 2.1: TCP connection state diagram (from Sergiodc2, M. Pauley, and Scil100 [73])

2. Forward packets where the protocol is TCP, direction is to the LAN, and the TCP destination port is ephemeral.[2]

3. Drop any packet.

The second rule is necessary to allow the client-to-server direction of the TCP stream, but mapping this rule-set back to the policy reveals that TCP connections from the Internet to the LAN are not blocked if the source port is 80 and the destination port is ephemeral. An attacker's client application (bound to port 80) could connect to a malware server on the LAN.

The second rule would be superfluous with stateful TCP tracking, as return packets would be associated with the connection permitted by the first rule. By removing the second rule, stateful tracking can protect against the attack described above.

While it is useful to examine hybrid and Software Defined Networking (SDN)-based stateless firewalls in our critical analysis of related work, we limit our study of firewall performance to the stateful (state-of-the-art) type.

Stateful packet filtering requires forwarding plane support for functions that are not easily handled at high speeds. For example: IP fragments (which may not contain the original packet's transport layer headers) require re-assembly; an operation that adds extra buffering, lookups and time-out proceeding for each affected packet; whereas an IP router could simply forward fragments without re-assembly. Keeping track of a TCP connection's state requires a connection table update for each packet, therefore, stateful firewalling is not an embarrassingly parallel task if distributed on a per-packet basis.

It follows that state-of-the-art firewalls have more processing to do (per-packet) than IP routers. Given access to the same basic technology (silicon process density and power budget), the increased complexity of a firewall would result in fewer packets being processed in the same time compared to an IP router. We support this argument by first noting that common switch ASICs (so-called merchant silicon) are not capable of supporting sophisticated packet processing [57], and then examining the suitability of the hardware architectures typically used to implement stateful firewalls in the two subsections that follow:

2.4.1 CPU-software firewalls

Modern server operating systems integrate mature host firewall functionality. Host firewalls typically process packets to and from the transport layer on the host itself, however,

[2] dynamically allocated by the client from a range of port numbers not requiring super-user privileges.

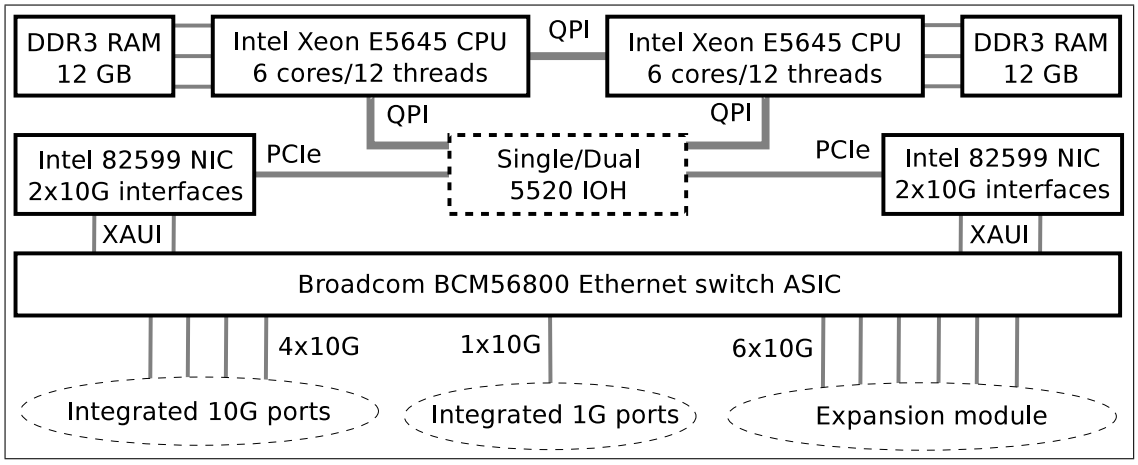

Figure 2.2: Hardware architecture of the Cisco ASA 5585

the operating system network stack can usually also bridge and route packets (switching on layer 2 and layer 3, respectively). Netfilter (the Linux kernel packet filter) is capable of stateful firewalling of both host-terminated as well as routed and bridged connections. We describe Netfilter in detail in section 2.9.

Despite the single-core small-packet-rate limitations of pure software routers and firewalls, which we introduced in section 2.3.2, modern software firewall implementations can scale to high *aggregate* throughputs for medium to large packet sizes.

While our investigation begins with the observation that this architecture is not well suited for FDT [25], we acknowledge that software firewalls could adequately serve many, or perhaps most, other network applications. Some high-end commercial firewalls (including the Cisco ASA tested in our experiments) are implemented on commodity server components in a customised chassis. Based on a Cisco presentation [64] and logs from the system tested, we inferred the internal architecture of the Cisco ASA 5585, illustrated in figure 2.2.

High performance packet processing on general-purpose hardware relies on directing packets of each transport flow to the same core of one or more multi-core CPUs. A thread for each core is configured to service a different packet queue on a multi-queue network interface. Drivers and network stacks employ polling (instead of interrupts), and eliminate copying of buffers to improve throughput and reduce latency. Most recently, user-space forwarding plane implementations such as Data Plane Development Kit (DPDK) [20] eliminated processing bottlenecks in the operating system to achieve even higher throughput.

The primary advantage of CPU-based packet processing is the highest flexibility

available for implementing complex functions such as stateful packet filtering or payload inspection [68].

2.4.2 Network processors

Multi-core processors providing parallelism intended specifically for hardware network forwarding planes are known as Network Processing Units (NPU). NPUs typically comprise a large number of simplified Reduced Instruction Set Computing (RISC) cores designed for executing event-driven code that performs tasks typical in network equipment. Marketing material for commercial hardware firewalls from ForiNet and Juniper claim to employ NPU-based designs.

The advantage of an NPU over an ASIC is flexibility better suited to complex tasks [33], similar to CPUs but with greater forwarding performance, at the cost of constraints of per-core resources and a more complex programming model.

Casado et al. [13] observed that NPU vendors initially struggled to strike a complexity-flexibility trade-off which was attractive to the market, while more recently Pongrácz et al. [68] (authors from Ericsson – also an NPU vendor) state that, measured in performance per Watt, recent NPUs outperform CPUs even for complex tasks (unfortunately without elaborating on how they reached this conclusion – for example, which tasks were tested).

The literature is lacking in rigorous performance comparisons between NPUs and other architectures for the application of stateful firewalls, but there is consensus on the general (and intuitive) principle that each hardware platform lies on a curve that relates forwarding performance and processing flexibility. McKeown estimates the successive increases in speed between CPUs, NPUs and ASICs to be one order of magnitude [50]. This supports our earlier argument that a hardware firewall would need to trade off between these two attributes of the underlying technology.

2.5 OpenFlow

OpenFlow is a standard that implements the idea of SDN [51]. The control plane functions of network elements can be centralised (to a so-called controller) by providing a remotely programmable interface for the control plane functions, in order to manipulate forwarding plane configuration. While some definitions of SDN extend far beyond fine-grained programmability of packet forwarding behaviour, we focus on OpenFlow as a realisation of SDN.

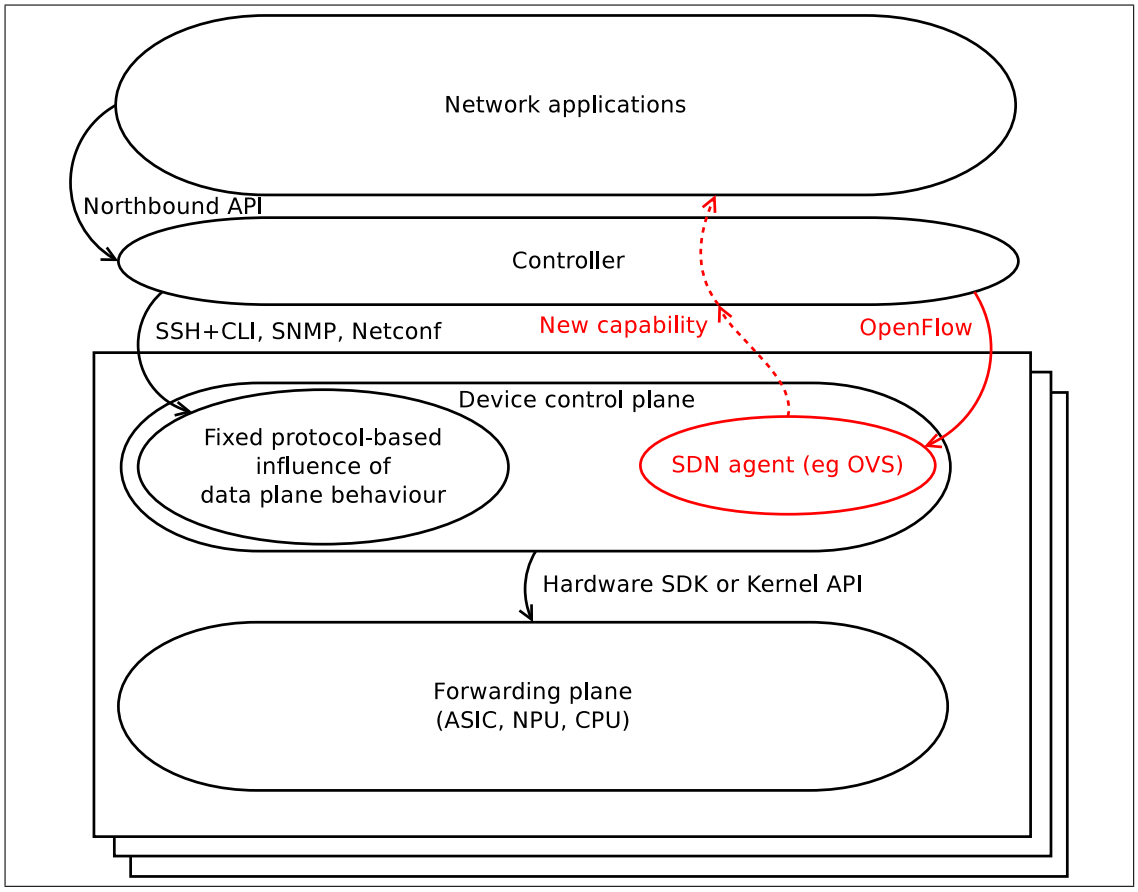

Figure 2.3: Architecture of a Software Defined Network

In an OpenFlow switch, forwarding plane actions are encoded in a series of switching hardware memory tables [63]. Except for wild-card protocol field matching (which was not supported in earlier generation switch ASICs), OpenFlow does not define new forwarding plane behaviour, instead it makes use of the existing functions used by embedded control plane software found in common switching and routing hardware. OpenFlow controllers can manipulate the switching tables inside any OpenFlow-capable switch by managing so-called flow entries. Flow entries can be specified in multiple tables that form a packet processing pipeline. The component responsible for implementing OpenFlow functions in a switch is called the agent.

A flow entry specifies matching rules, counters and actions to be applied to network traffic traversing the OpenFlow switch. This match-action abstraction is the central contribution of OpenFlow. Using the OpenFlow protocol, it is possible for software to remotely control packet switching with a high degree of granularity.

Figure 2.3 shows the interfaces between network devices, controllers and applications. The components and interfaces indicated in red represent the addition of OpenFlow as the basic technology to enable Software Defined Networking.

The pragmatic re-use of existing technology allowed an evolution toward SDN. Much progress has been made, first by the network research community and then by industry, in creating software platforms for OpenFlow controllers. Multiple network equipment manufacturers are now providing OpenFlow capability in their switches and routers.

2.6 open vSwitch

open vSwitch (OVS) is an implementation of OpenFlow which is typically used as a virtual switch.

Virtual switches are deployed in conjunction with server virtualisation technology (so-called hypervisors) to provide network connectivity to virtual machines. OVS is Free and Open Source Software (FOSS), and gained popularity with the rise of hypervisors such as Xen and KVM. While the goal of OVS is to provide a flexible, performant platform for overlay networking and packet classification for virtual machines, it is also both an OpenFlow switch and OpenFlow agent [66].

OVS's multi-layer architecture separates the control plane from the software forwarding path, which in turn has a fast exact-match cache and a slow path that handles cache misses (performing more complex wildcard or longest-prefix matching). The systems design of loosely coupled components has proven useful for switching implementations with different data paths, including both user and kernel-space, as well as hardware offloading (described in section 2.7.2).

The OpenFlow switch that we used in the implementation of our prototype firewall includes OVS as its OpenFlow agent. Consequently, OVS is used extensively in our research, and we therefore describe OVS's architecture and configuration in further detail below.

2.6.1 Architecture

The architecture of the OVS OpenFlow agent component is composed of two user-space server processes, some command-line utilities and a configuration database (illustrated in figure 2.4).

Connections to the OpenFlow controller and interfacing with the forwarding plane (or *data-path* in OVS terminology) is handled by ovs-vswitchd.

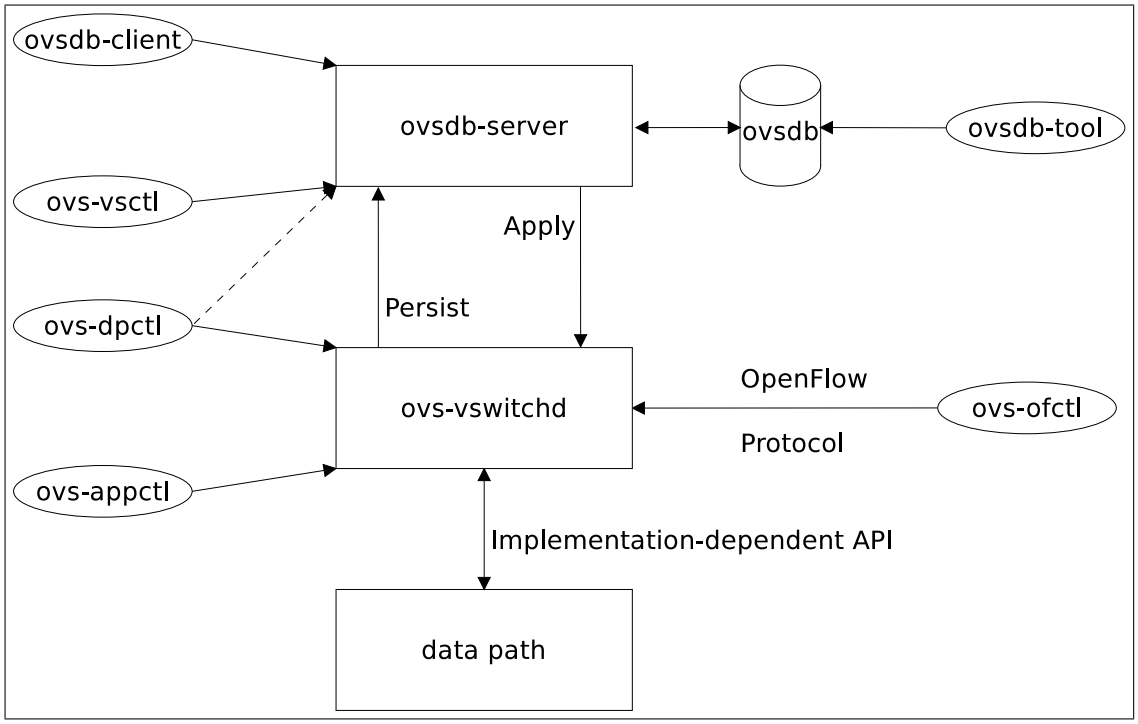

Figure 2.4: OVS agent architecture

2.6.2 OVS utilities

The `ovs-ofctl` utility allows the user to manually manipulate the flow tables of the switch without a controller.

While `ovs-appctl` and `ovs-dpctl` query and modify some run-time state of ovs-vswitchd, the configuration of the OpenFlow switch instances are instead obtained from a separate server process: ovsdb-server. Persistent configuration created by the `ovs-vsctl` utility is stored in the ovsdb file, and applied to ovs-vswitchd via ovsdb-server. The stored configuration can either be retrieved with the `ovsdb-client` utility, or the ovsdb file can be manipulated directly with `ovsdb-tool`.

2.6.3 OVS usage

A detailed example of OVS usage is provided in appedix A.

2.7 Hybrid forwarding

Beginning with flow caching in software-only routers, the designers of switching platforms have often employed the optimisation of switching the forwarding path of packets

between sophisticated-but-slow and simple-but-fast mechanisms, as an optimisation and a work-around to the inherent trade-offs we observed in the preceding sections.

This re-occuring hybrid design pattern of acceleration or bypass switching is applied in the prototype firewall we developed in our research. We explore related work on hybrid forwarding in this section.

2.7.1 Multi-layer virtual switches

Mekky et al. extend OVS, adding so-called *application tables* to enable application-layer processing of flows in the vSwitch without having to send packets to the control plane [53]. Similarly FAST [56] and OpenState [7] add capability for flow state tracking to the virtual switch. By leveraging the TCP options field match type in OVS, FAST can be used to implement a stateful firewall. At the time of writing, hardware switches supporting OpenFlow 1.5 (which includes the TCP options match type) were not yet commercially available.

2.7.2 Co-processor fast-path

Some hybrid architectures partially offload packet processing to NPUs or fixed-function hardware pipelines (co-processors):

Yang and Yonggang [82] demonstrate a hybrid Netfilter-based firewall leveraging a tightly coupled Field Programmable Gate Array (FPGA)-CPU forwarding plane. Their implementation modified the Linux kernel with Netfilter hooks that examine packet headers in the CPU, while the complete packet remains queued in the forwarding plane. Similarly, Chen et al. [15] add hardware-offload acceleration for Netfilter, with a custom kernel module communicating with a data path implemented on NetFPGA [45], and Accardi et al. [1] implement the same scheme on an NPU. In the broad sense, both of these systems are SDN firewalls, but tight coupling to the hardware limits their suitability for vendor-agnostic deployments.

Split SDN Data Plane (SSDP) [57] is another hybrid approach that off-loads complex processing to an NPU subsystem in an ASIC-based hardware switch. The authors demonstrate deep packet inspection as an application. SSDP differs from the above hybrid designs in that it defines an interface for the NPU subsystem to be configured by a single SDN controller, an approach that has the potential for standardised and widespread deployment.

Finally, some ASIC based switches integrate OVS in the control plane, and employ the multi-layer flow-cache approach to accelerate traffic after the initial packets of a flow

are punted to the control plane.

2.7.3 Co-processor slow-path

The inverse approach of hardware off-loading of packet forwarding in a software system, is the off-loading of complex packet processing to software in a hardware switch. While this approach is commonly used for handling packet processing exceptions (punting) without specific concern for the performance impact (as highlighted in section 2.3.2), some designs include software co-processing to add state tracking and application layer processing features to otherwise hardware-only forwarding.

Lu et al. [47] use this approach to enable large routing tables, and packet buffering with a server platform, combined with ServerSwitch [46] – a switch ASIC on a plug-in card. It is not clear what the advantage of ServerSwitch is, compared to leveraging an open control plane interface for ASIC programming and out-of-band forwarding plane connections between the switch and the server (the approach used for our prototype).

2.7.4 Control plane advanced packet processing

One configurable OpenFlow action is for the switch to tunnel packets to and from the controller via so-called packet-in and packet-out messages [63]. It is then possible for the controller to inspect packets directly and perform forwarding decisions in the traditional punt-and-switch architecture employed by many hardware routers.

OpenFlow can simplify the local functions of a switch because certain tasks are handled by the controller, reducing the need, for example, to compute an optimal forwarding table when routing changes occur. This affords equipment vendors the cost savings of using low performance system-on-chip processors to run the OpenFlow agent. Unfortunately, such processors limit the performance of packet-in/packet-out forwarding via the OpenFlow controller.

The second problem with packet-in/packet-out tunnelling is that OpenFlow messages are typically transported over TCP, and when the packets within the tunnel are themselves transporting TCP traffic, interaction of TCP retransmission timers can lead to poor performance [78].

Collings and Liu [16] note scalability constraints when relying on OpenFlow controllers to process forwarding plane traffic. Nevertheless, we note existing proposals for advanced packet processing (such as stateful connection tracking) continue to rely on packet-in/packet-out tunnelling.

2.7.5 Forwarding plane advanced packet processing

For the most part, the evolution of OpenFlow sees the inclusion of new match and action types with each subsequent version of the specification. The state-of-the-art in SDN seeks to extend the capability of the forwarding plane beyond the constraints of OpenFlow in two dimensions: protocol independence and stateful processing.

Protocol independence would allow the structure of packet headers to be defined independently of the hardware structure of the data-plane. P4 [9] proposes the use of a compiler to transform a logical definition of protocols, and the associated processing that can be performed, into configuration for flexible packet switching hardware. A P4 switch is therefore truly software defined (or re-defined at run-time), and can then be controlled by a protocol like OpenFlow during operation. It remains to be seen whether protocol independence can be delivered by a generation of more flexible switch ASICs, or if it will be realised by widespread deployment of NPUs in general-purpose switches.

The implication of protocol independence on the design of firewalls is significant. Once the structure of packets is software defined, the enforcement of network security policies must also be equally flexible and independent of hardware platforms. This would apply even to stateless packet filtering (ACLs).

While stateful packet processing can currently be performed in software (on general purpose CPUs or NPUs), future software defined networks may see the inclusion of simple state machines in switch ASICs, leveraging an open standard to define them (along the lines of OpenState or FAST). These developments are promising for the implementation of high performance firewalls, but we believe careful consideration should be given to the mapping between network flows and expensive hardware resources (such as Ternary Content-Addressable Memory (TCAM) entries) if and when this capability is realised.

2.8 OpenFlow-based firewalls

The OpenFlow specification not only allows flexible control over how packets are forwarded by hardware, but also whether they are switched at all. This enables packet filtering in the switch based on flows programmed by a controller designed to implement a specific security policy. We now explore existing work in this area:

2.8.1 OpenFlow controller firewalls

Some OpenFlow controller frameworks include some firewall functionality. One example is the Floodlight OpenFlow controller [69], which has a component currently under

development that implements ACL functionality and exposes a Representational State Transfer (REST) Application Program Interface (API) for applications to configure filtering policy on an OpenFlow switch. In effect, this work performs translation from firewall-like rules into flow specifications. This follows a trend which has the traditional interfaces for network management and configuration (such as Simple Network Management Protocol (SNMP)) being substituted with popular and light-weight Hypertext Transfer Protocol (HTTP)-based APIs.

Thimble [72] is an OpenFlow controller that improves the management of ACLs specifically for implementations of the science demilitarised zone (DMZ) architecture. A web-based interface is presented to IT support staff to conveniently and systematically add and remove rules that protect the HPC infrastructure located within the DMZ network. Russell [72] also proposes the application of a "default flow" rule that directs traffic not specifically permitted to be switched into the science DMZ to an existing enterprise firewall. Prasad et al. [12] extend the stateless bypass switching in a science DMZ with a policy controlled by application layer gatekeeper middleware.

Due to the lack of stateful connection tracking, neither of these approaches are comparable with the state-of-the-art stateful firewalls.

Shieha [74] modified the layer-2 Media Access Control (MAC) learning switch included in POX, adding a stateful firewall module capable of blocking connections on a five-tuple protocol/address/port basis, or application identification by means of simplistic[3] string matching. Firewall policy is loaded into the controller's memory from a text file and evaluated as new flows are established. Flow tracking is performed as packet-in messages are received. The performance limitation of this approach is described in section 2.7.4.

2.8.2 OpenFlow hybrid firewalls

We make a distinction between the use of the controller to track connection states directly (as described in the preceding section) and an OpenFlow controller managing the flow of packets between distinct slow and fast paths. The primary difference in these strategies is that the individual components of the latter (hybrid) approach are known to have specific performance strengths, whereas the literature raises scalability concerns with the former.

SciPass [5] implements a secure science DMZ using OpenFlow switching controlled by an intrusion detection system. This approach allows for the use of a traditional firewall as the default traffic path, with the ability to accelerate connections identified by the

[3] Each packet is searched individually, without TCP stream reassembly.

IDS (based on traffic mirrored by the switch) as known-good or trusted connections. The use of OpenFlow instead of custom hardware, and the option of firewall bypass, differentiates SciPass from the original shunting work [31].

The separation of the firewall from the shunting policy is natural in SciPass, since IDS signatures are conceptually different from a packet filter set. We expect this design to be well suited to environments already familiar with, and capable of operating and maintaining, intrusion detection systems.

SciPass also has a particularly useful scalability feature: the intrusion detection traffic can be load-balanced among nodes of an IDS cluster. This load-balancing is performed by the OpenFlow switch.

Narisetty [58] describes vArmour, a Floodlight-based OpenFlow controller application offering distributed firewall capability. vArmour appears to implement the shunting approach (described as *steering*). Unfortunately, with the focus of the study being the timing of session off-loading, the details of the vArmour firewall design and the mechanism of its interaction with the controller are not provided. We note that vArmour is proprietary technology, whereas SciPass is FOSS.

In the chapters that follow, we study our own approach to the design of a hybrid OpenFlow firewall. Our prototype shares some similarities with SciPass (apart from the shared objective to enable FDT), but differs in the way that the firewall policy and shunting decisions interact.

2.9 Netfilter - the Linux firewall

Netfilter is Linux's software firewall and is used in our research as the slow-path component of a hybrid system. In this section we describe the structure and function, as well as configuration, of Netfilter.

2.9.1 Netfilter's design

Figure 2.5 indicates the logical flow of Linux packet processing, including Netfilter components. Architecturally, Netfilter is composed of user-space utilities, common components in the Linux kernel, and loadable kernel modules that implement specific firewall functions. Figure 2.6 shows the structure of Netfilter.

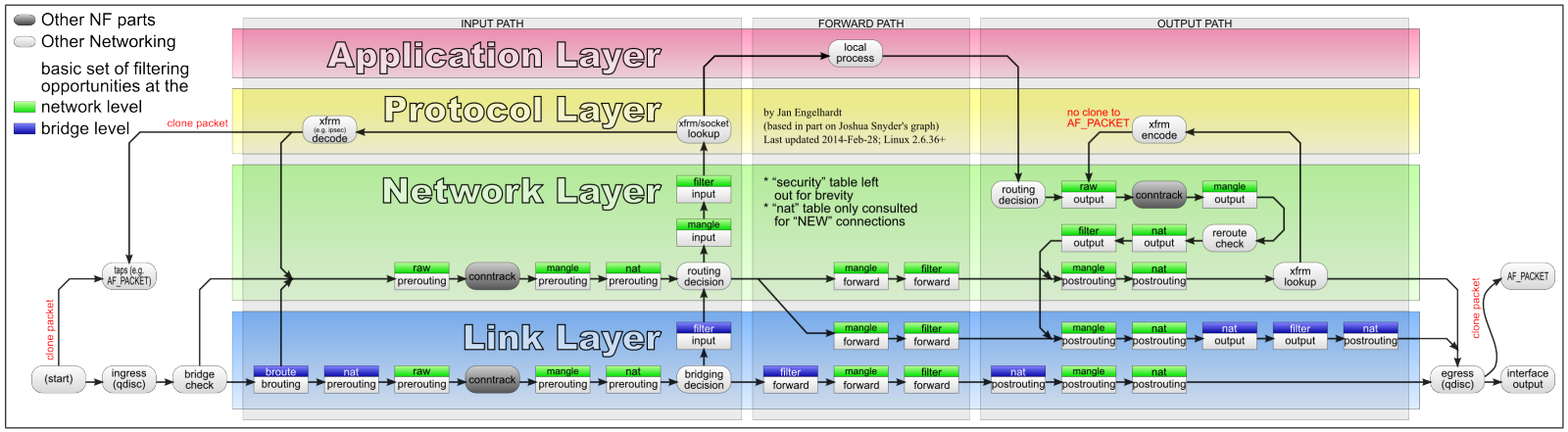

Figure 2.5: Linux Netfilter packet flow diagram (from J. Engelhardt [27])

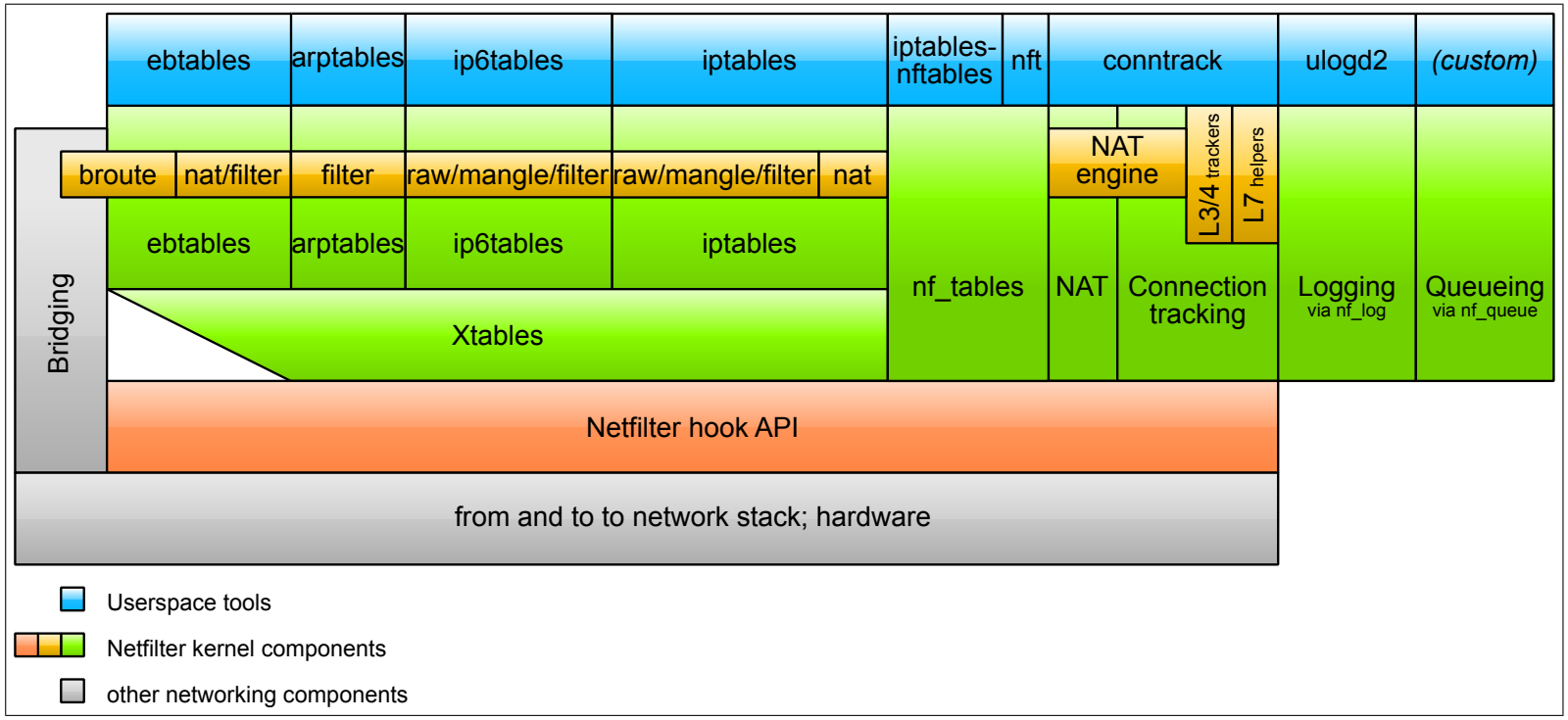

Figure 2.6: Linux Netfilter components (from J. Engelhardt [26])

2.9.2 `iptables` configuration

For Internet Protocol (IP) version 4, the `iptables` utility is primarily used to configure Netfilter firewall rules. Our prototype firewall also uses the `conntrack` command to interface with Netfilter's connection tracking system (described at the end of this section).

Netfilter rules must belong to a *chain* and a *table*. Conceptually, the *chain* evaluated is determined by the path of a packet through the network stack (shown in figure 2.5). The sequence of Netfilter chains and rules applied to each packet is called *traversal*.

Any packet (whether bridged or routed) is first checked against the `PREROUTING` chain. Once a routing or bridging decision determines whether to send the packet to the host's transport layer, or to forward the packet, it is either checked against `INPUT` chain or `FORWARD` chain rules. Packets originating from the host's transport layer are checked by `OUTPUT` chain rules, and forwarded packets are finally checked by `POSTROUTING` chain rules, prior to being sent to the output network interface.

In each Netfilter chain, rules belong to *tables* according to the function (or action) of the rule. The `PREROUTING` chain has `raw`, `mangle` and `nat` tables; `FORWARD` has `mangle` and `filter`; and `POSTROUTING` has `mangle` and `nat` tables.

Similar to OpenFlow flow specifications, `iptables` rules contain matches and actions. Netfilter includes a myriad of match types, ranging from the input or output interface, to full application layer inspection. Actions (specified with the `-J` parameter) can accept or discard a packet, or perform some other state-altering action (such as marking the packet, re-writing a header, or jumping to a different set of rules to continue evaluation).

2.9.3 Netfilter connection tracking

Conntrack enables stateful packet inspection in Netfilter. It does this by first identifying the establishment of connections, and then allocating data structures used to track the expected state of each connection, matching each packet against this state and (when appropriate) updating those states based on already validated packet headers.

States associated with each transport layer connection tracked by Conntrack are:

- `NEW`: initial packets have been received but the firewall has not yet seen bi-directional communication (for example, by completion of a 3-way TCP handshake).
- `ESTABLISHED`: the connection is established and the application-layer payload is now transported in the packets.

- `RELATED`: state applied to packets of a connection that have been determined to be related to another connection through application-layer inspection (for example: the data connection of an FTP session is related to its control connection).
- `INVALID`: applied to any packet that Netfilter cannot otherwise track (including via the `NEW` state).
- `UNTRACKED`: a state given to packets for which connection tracking is disabled by instruction of a firewall rule.

2.10 Traffic generation and testing

With the performance focus of FDT and our objective to improve on the performance of existing firewall architectures, the ability to evaluate network performance is central to the research. In this section we review approaches to network performance testing.

2.10.1 Test standards

Techniques for benchmarking network elements are of interest to both suppliers who wish to market their products and end-users who wish to evaluate the performance of products against their own requirements, as well as for comparison amongst competitors. For this reason, a number of standards and methodologies exist:

1. IETF (RFC) standards for benchmarking of "Network Interconnect Devices": RFC1242 [10] and RFC2544 [11].
2. IETF (RFC) standards specific to the benchmarking of firewalls: RFC2647 [60] and RFC3511 [37].
3. Test methodologies developed by network test equipment manufacturers: e.g., BreakingPoint firewall testing methodology.
4. Test methodologies developed by independent test houses: e.g., NSS Labs firewall testing methodology [62].

These standards and methodologies define aspects such as terminology for the components in test setups, but also of particular interest to our research, the profile of test traffic generated for the purpose of evaluating performance.

One important factor during testing is the distribution of packet sizes, as this has a direct impact on the packet rate for a given data rate. As described in section 2.3, processor-based forwarding performance may suffer at high packet rates.

Another aspect of performance of specific relevance to devices that track connection state is the composition of the test traffic grouped by transport layer header values. Firewall test methodologies are designed to demonstrate the scalability of the device under test in this dimension, and therefore seek to discover the maximum number of concurrent connections supported, as well as the maximum rate at which the firewall can track the establishment of new connections.

When measuring the absolute throughput supported (in bits per second), careful attention is given to ensuring traffic used in this test consists of a large number of concurrent connections (often tens or hundreds of thousands), presumably because this reflects a worst-case scenario for flow state tracking. Unfortunately, none of the standards and methodologies included above specify a test designed to determine the maximum throughput achievable for a single or a small number of connections (typical of FDT applications).

2.10.2 Generating network traffic

Standards for benchmarking network equipment listed in section 2.10.1 require the properties of network traffic used during testing to be reported, but avoid dictating the method to be used for generating traffic. This section examines some test traffic generation techniques that have been employed:

2.10.3 Sampling real traffic

The simplest approach to network traffic generation is to capture traffic in a real network environment that the test is meant to replicate, and then to replay that traffic during experimental runs. Alternatively, live network traffic can be safely copied (mirrored) and used as an input into the system to be evaluated. An example of this technique specifically relevant to our work is found in the evaluation of shunting [31], where the classification of flows is demonstrated.

An advantage of this approach is the accuracy obtained by the use of real traffic, which might otherwise be very difficult to synthesise due to the complexity of behaviour of different implementations of protocols and applications, as well as the challenge of simulating the human (end-user) influence on traffic patterns [28].

While it is possible (given sufficient hardware resources) for bi-directional network traffic to be captured and replayed with accurate timing and without packet loss [28], the scope of the testing possible is limited to passive applications that do not influence the traffic flow. For example, if the device under test was to prevent forwarding of a

packet in the original captured traffic, this would invalidate the state of transport and application layer processes associated with that packet.

2.10.4 Simulating the network layer

If the objective of network testing is to measure the performance of a device functioning at the network layer (such as a router), then the task of generating test traffic need only take into account those attributes relevant to layer three functions (e.g., packet size, protocol, addresses, etc.). In this case, replaying captured traffic is a valid (albeit not very flexible) approach.

Many traffic generation tools (hardware and software) exist. One such tool is Harpoon [76], which is capable of synthesising UDP and TCP flows with a specified set of attributes (including protocol, timing, length and data attributes).

2.10.5 Simulating the application layer

In order to generate test network traffic in an experiment that aims to measure the influence of the network on applications, it is necessary to reproduce (or simulate) the data transmission and reception behaviour of both the application and transport layers.

The need to test the performance of network applications specifically has led to the development of many different load-testing applications. One such tool, SURGE [6], attempts to generate representative Web workloads in order to test HTTP proxies, servers and the underlying layers (such as the network itself).

2.11 Conclusion

The simplifying principle of Internet routing is that intermediate nodes can forward packets independently of the applications using the network. To control packet routing (the network layer) based on transport layer state violates the design principle of separating network protocols into independent layers, but this is precisely what is required to perform flexible, connection state-dependent network filtering.

In this chapter, we have seen how the complexity of routing and filtering influences the speed at which it can be done given state of the art hardware. Previous work has shown how the shunting strategy can be an effective means to improve overall performance, if network flows can be classified into those that require filtering and those that don't.

The emergence of standardised software interfaces to control the forwarding plane of high performance hardware routers (such as OpenFlow) presents an opportunity to

address complex security requirements with a system composed of simpler components. The chapters that follow explore this opportunity.

Chapter 3

An SDN-based shunting firewall

In this chapter we define our research objectives, and document the methodology we used to construct and evaluate our prototype SDN-based shunting firewall. We justify high level design choices, and then describe the design and implementation of NFShunt in detail.

3.1 Research question

In order to leverage the investment in high capacity networks, researchers must be able to use existing, convenient tools and work-flows to move large quantities of data quickly. The designers of network security measures at institutions (connected to research networks) that must deliver on this requirement, should not need to accept a radical departure from currently accepted best-practice designs used in other domains of networking.

Previous work with OpenFlow in this context has applied it to the management of (stateless) access control lists, and the implementations of shunting used intrusion detection for stateful bypass in the science demilitarised zone (DMZ) design (see section 2.8).

In our research, we studied the use of OpenFlow to implement a shunting firewall that provides loss-free network paths for permitted large data transfers (similar to the science DMZ), but allows the use of stateful firewalling for all other traffic by default. Application of this configuration of technologies to stateful firewalls is novel.

The primary problem is to develop a firewall architecture that meets the performance requirements of data-intensive science.

Additional goals of the research were to explore the following questions:

1. How can generic and standardised off-the-shelf components be composed to create a high performance (hybrid) firewall?

2. What is the packet-loss performance of a hybrid firewall (is it equivalent to a "clean network path")?

3. How would a hybrid Software Defined Networking (SDN)-based shunting firewall compare to a traditional firewall in terms of price-performance ratio?

4. What are the operational and maintenance benefits and drawbacks of a hybrid firewall?

3.2 Research approach

The following approach was used in the research:

1. An SDN-based hybrid (shunting) firewall was designed. This design was required to be practical to implement and offer significant improvements over existing designs.

2. A prototype firewall (NFShunt) was constructed based on the SDN hybrid design to demonstrate its feasibility.

3. The performance of NFShunt was compared to that of a representative traditional hardware firewall, and both systems were evaluated for their suitability in Fast Data Transfer (FDT) scenarios. The experiment described in section 4.1 was designed to perform this evaluation.

4. An analysis of the experimental results, as well as other performance metrics such as cost, complexity and flexibility, was performed and documented in sufficient detail to allow the conclusions of the research to inform decision making in real-world applications.

3.3 Prototype architecture

NFShunt is a layer 2 (transparent) firewall composed of two interconnected components, as illustrated in figure 3.1.

The shunting controller and the slow-path are co-located on the same physical server, while the OpenFlow switch provides external interfaces and performs hardware bypass-switching. The slow-path is based on the Linux kernel's built-in Ethernet bridging function, combined with Netfilter (the standard Linux firewall).

Each Linux Ethernet bridge port is physically connected to a corresponding "slow" port on the OpenFlow switch. One-to-one mapping between external (network) facing

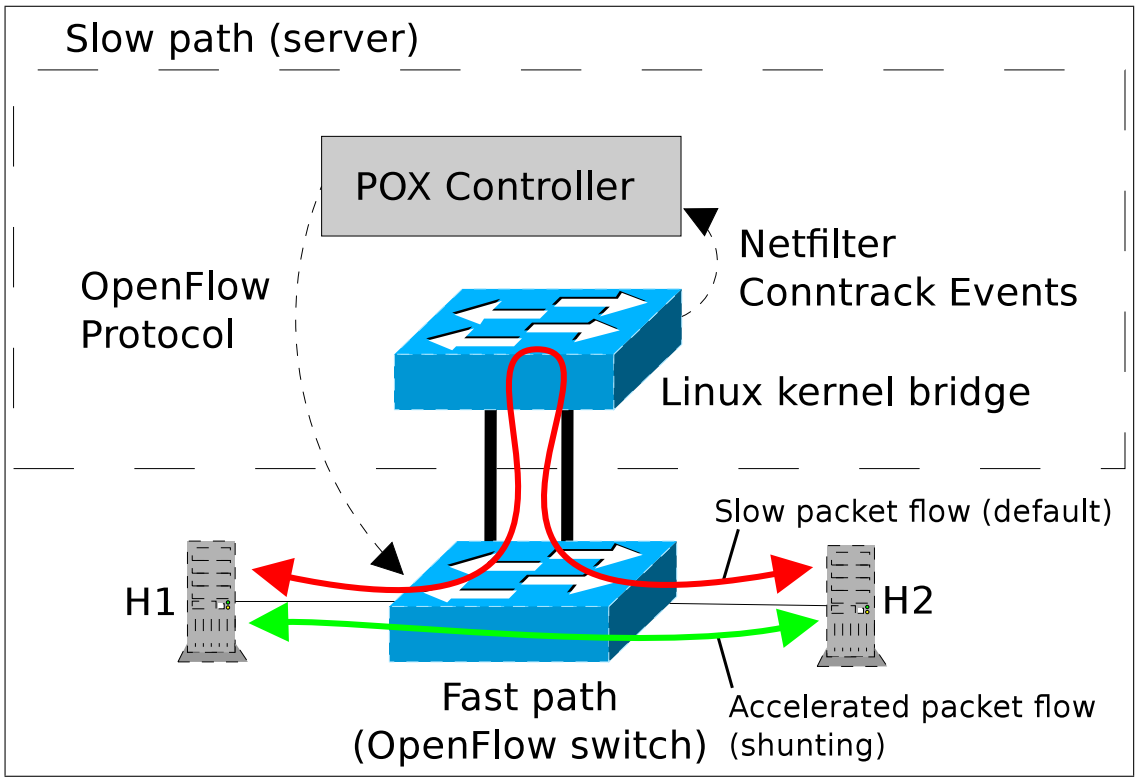

Figure 3.1: NFShunt architecture

ports and "slow" ports is statically configured during the installation of the firewall. Communication between the shunting controller and the OpenFlow agent is also configured during the prototype's setup.

No packets are forwarded by the OpenFlow switch until it receives programming from the shunting controller at run-time. When the controller starts, it installs low priority flow specifications that direct all non-shunted packets via the Linux Ethernet bridge. These flow table entries remain for the execution of the prototype.

Per-connection flow entries are installed dynamically during run-time to implement the firewall's bypass shunting according to iptables firewall policy. These entries are removed by the switch, a configurable number of seconds after the last packet matching the flow specification for the connection is switched.

3.4 Design choices

The hybrid design, based on bypass switching combined with a flexible software firewall, was chosen to address the primary research problem of enabling loss-free stateful forwarding suited to the performance requirements of data intensive research. In chapter two, we explored the strengths and weaknesses of both pure-software and ACL-based packet filters. Our design finds a compromise between the two, and consequently inherits at least some of their weaknesses. Once trusted connections are bypass-switched, the prototype is no longer able to detect security policy violations (e.g., through application layer inspection) due to the stateless packet forwarding performed in hardware. We note that, unlike ACL filters in science DMZs, the hybrid design allows for stateful tracking of connection establishment. Similar to ACLs and the original shunting work, the argument applies that packets that comprise the data-transfer portion of the connection are often encrypted, and therefore inspecting them offers limited value.

3.4.1 The toolkit approach

Unlike previous work that built hybrid firewalls with custom or proprietary components, our research followed a toolkit approach: NFShunt is (and is based on) Free and Open Source Software (FOSS) combined with standardised Commercial Off-The-Shelf (COTS) hardware components. The primary enabler of this is the adoption of OpenFlow by switch manufacturers. While this implies that *some assembly is required*, thereby increasing the initial complexity of installing and configuring multiple components, we hypothesise that it could offer cost savings and other operational benefits.

3.4.2 Transparent firewall

We chose to implement a transparent (bump-in-the-wire) firewall. This simplified the work significantly by avoiding re-implementation of layer 3 routing semantics in the fast-path. While this choice limits the flexibility of NFShunt's deployment in diverse network environments, a transparent firewall is much easier to add into an existing network design since it requires no layer 3 changes.

3.4.3 Linux Ethernet bridge and Netfilter

The lack of integration with Netfilter excluded alternatives to the built-in Linux kernel bridge function (such as open vSwitch (OVS) and Data Plane Development Kit (DPDK)). Due to its mature implementation and widespread use, the Linux bridge is a simple and reasonably performant choice for the slow forwarding path. The ability to monitor and manipulate Netfilter's connection tracking directly makes it the ideal choice for a hybrid firewall design.

3.4.4 Integrated firewall and shunting policy

The objective of designing a firewall required the mechanism for expressing shunting policy to be coherent with the traditional firewall ruleset. We explored mechanisms to annotate Netfilter Conntrack objects with shunting policy expressed through iptables rules. While this adds complexity in the design, it is closer to what firewall administrators would expect from a traditional firewall.

3.5 Low-level design

NFShunt was implemented in Python based on the POX OpenFlow controller library. The slow-path forwarding plane is a Linux 3.2.0 kernel bridge, while the fast-path (an OpenFlow Ethernet switch) is used for all external connections.

The prototype makes use of a Pica8 P-3290 top-of-rack Ethernet switch (based on a Broadcom switch Application-Specific Integrated Circuit (ASIC) and customised OVS as the OpenFlow agent). The switch is equipped with 48 1000Base-T and four 10GBase SFP+ ports. For the prototype firewall slow-path, we used a fit-PC3 Pro with four Intel 82574L-based 1000Base-T Ethernet NICs connected to the switch.
NFShunt's implementation is structured in five modules:

1. The **controller core logic** triggers by-passing of flows based on input from the configuration and slow-path interface.

2. The **slow-path interface** processes Netfilter connection tracking events.

3. The **fast-path interface** communicates with the OpenFlow switch.

4. The **configuration interface** adapts NFshunt to the instance-specific details of the network and firewall policy.

5. The **logging interface** caters for troubleshooting and performance monitoring of the prototype.

The remainder of this chapter describes manual configuration required for the prototype to function, as well as the design and implementation of each module. We start with the interface modules and conclude with a description of the core logic, which ties together all the functions of NFShunt.

3.5.1 Fast path configuration

Some basic configuration of the OpenFlow switch (the fast path) is required for the shunting prototype. In addition to the OpenFlow protocol itself, the Open Networking Foundation has defined a standard protocol (OF-CONFIG) to allow remote (and potentially automated) configuration of OpenFlow switches. It is therefore possible for the prototype to add the necessary configuration to the switch based on the information already available from the configuration module. Unfortunately, the firmware of the Pica8 switch available during development of the prototype did not support OF-CONFIG.

Manual configuration of the Pica8 switch was therefore necessary, and OVS commands were used for this purpose.

Configuring the prototype with OVS commands

The first step is to create an OVS bridge instance with an associated controller (192.0.2.1:6633 is the Transmission Control Protocol (TCP) endpoint of the shunting controller):

```
ovs-vsctl add-br br0 -- set bridge br0 datapath_type=pica8
ovs-vsctl set-controller br0 tcp:192.0.2.1:6633
```

The intent of the prototype is for all forwarding to be under the control of the slow path. When the control connection fails, the OVS bridge instance should not revert to any default behaviour (such layer 2 switching), since this would not be secure. The

bridge is therefore configured not to forward frames unless explicitly programmed to do so by the controller:

```
ovs-vsctl set-fail-mode br0 secure
```

Next, two slow path ports and two fast path ports are added into the bridge. The slow and fast path ports are matched by the slow path configuration, as described in the next section. In this example, two 1000Base-Ethernet ports (10 and 11) are used for the slow path, and two 10GBase-Ethernet ports (49 and 50) are used for the fast path:

```
ovs-vsctl add-port br0 ge-1/1/10 -- set Interface ge-1/1/10 type=pica8
ovs-vsctl add-port br0 ge-1/1/11 -- set Interface ge-1/1/11 type=pica8
ovs-vsctl add-port br0 te-1/1/49 -- set Interface te-1/1/49 type=pica8
ovs-vsctl add-port br0 te-1/1/50 -- set Interface te-1/1/50 type=pica8
```

With the above configuration, the switch will attempt periodically to establish a connection to the shunting controller and, when it does, it will receive flow configuration to switch frames between the fast path ports via the slow path. This configuration is persistently stored in the OVS configuration database of the switch's OpenFlow agent, therefore the fast path configuration only needs to be performed once (during installation of the firewall).

3.5.2 Slow path configuration

The controller communicates with Netfilter via the user-space interface (Netlink) of Netfilter's connection-tracking module (Conntrack).

The operation of the prototype requires specific configuration of Netfilter using the `iptables` command line utility. These configurations ensure that information about TCP flows present in the Linux kernel (due to the fact that packets associated with those TCP flows are seen by the slow path in the kernel) are made available to the shunting controller.

iptables configuration for NFShunt

Transport layer connections are automatically tracked by Netfilter once the Conntrack kernel module is loaded:

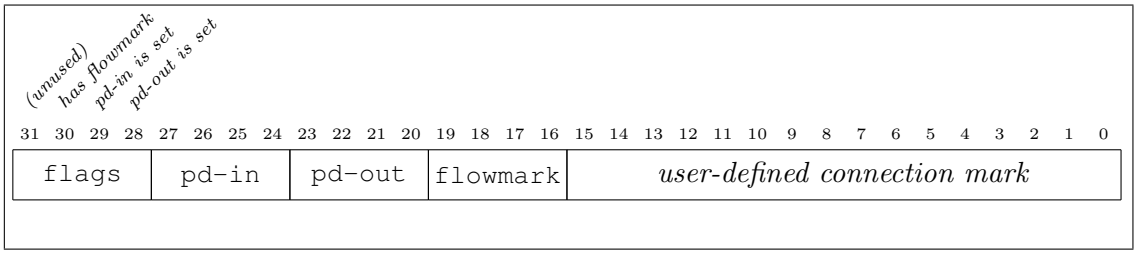

Figure 3.2: Connection mark bit fields used by NFShunt

```
modprobe ip_conntrack
```

Since NFShunt functions as a transparent (layer 2) firewall, it is also necessary to ensure that Netfilter is configured to inspect bridged packets (in addition to routed packets) passing through the Linux kernel. This is the default configuration on most Linux systems, but it can be explicitly configured as follows:

```
sysctl -w net.bridge.bridge-nf-call-iptables=1
```

Since the system administrator is expected to configure the default slow-path firewall policy as a set of Netfilter rules, NFShunt was designed to integrate the expression of shunting policy into the same rule-set. This is achieved through the use of Netfilter's packet mark and connection mark modules.

The mark extensions define both matching extensions and rule targets. The packet mark extension permits the administrator to assign any 32-bit value to a special field in the data structure associated with each packet processed by the Linux kernel (using the mark target). This value can be read by other rules using the packet match extension, and can also be accessed by the connection mark extension, which enables marking of connections identified by the Conntrack module.

In order to permit the administrator to continue using the mark extensions for other purposes, the prototype uses only the 16 most significant bits of the mark value, and masks each operation to leave the lower 16 bits unmodified.

The prototype uses mark bits to store the required shunting action (in the `flowmark` field), as well as flags and information about the flow's ingress and egress kernel bridge physical ports (in the `pd-in` and `pd-out` fields). Figure 3.2 documents the layout of the 32-bit mark field.

Only the packet marking target is required for the rules that express shunting policy:

Three configurable values are defined to express the various supported shunting actions: ignore (to do nothing with the connection), shunt (to bypass the connection via the fast path), and block (to install a rule to drop the connection in the fast path). This value must be encoded into the 4 bits from bit 16 to bit 19 (allowing for expansion of the prototype's actions to 16).

While the generic (logical) flow of matching on, and applying actions to, packets in the slow-path is described in section 2.9, the logic employed for the prototype is further described as a flow diagram in figure 3.3. The dotted lines indicate the link between actions encoding information in `PREROUTING` which is later matched in `POSTROUTING`.

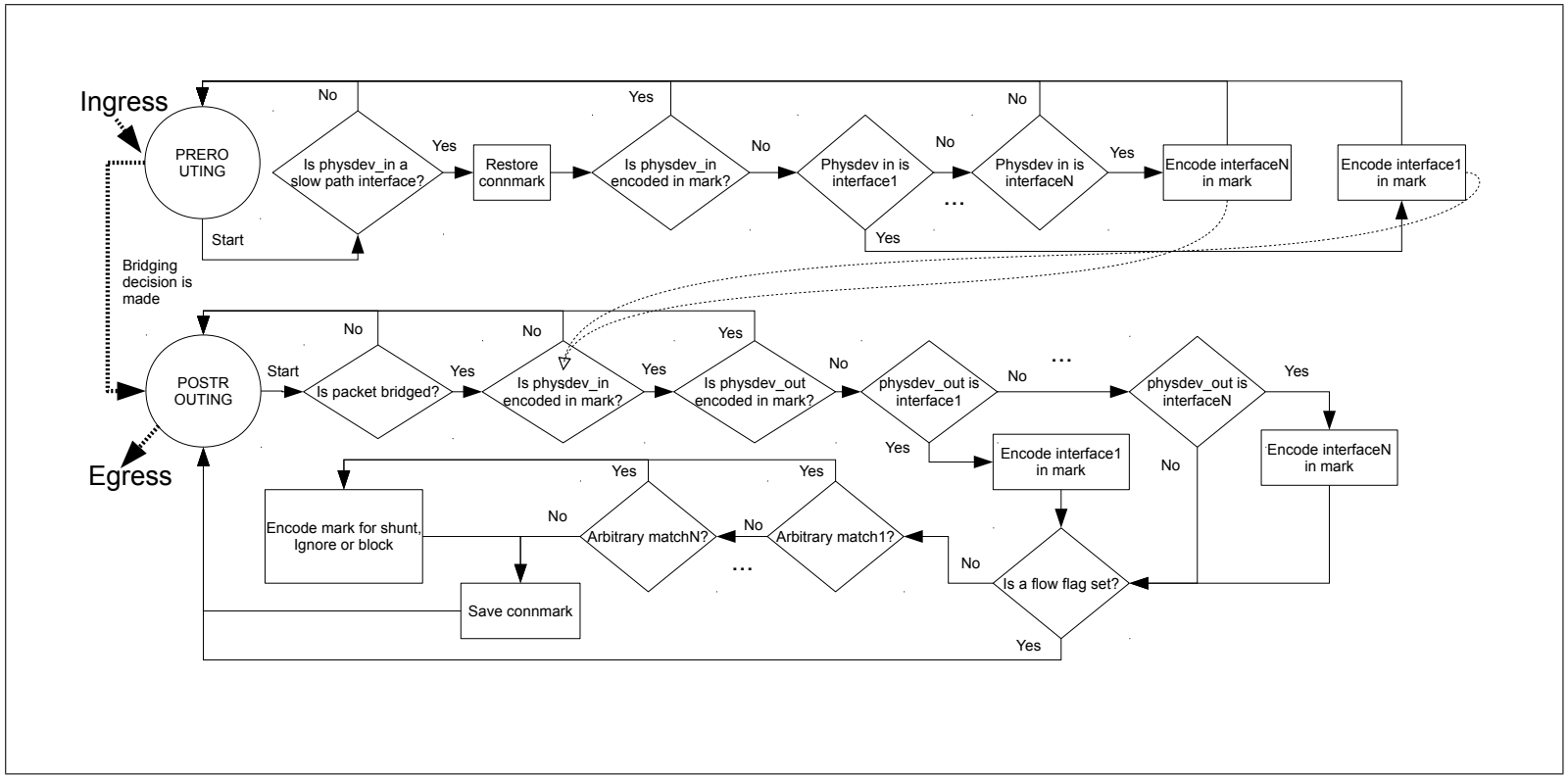

Figure 3.3: NFShunt Netfilter rule flow diagram

NFShunt makes use of the `mangle` table in the `PREROUTING` and `POSTROUTING` chains to mark both packets and connections for the purpose of shunting. In order to simplify integration with an existing `iptables` rule-set, a dedicated chain is used for shunting policy rules (`NFSHUNT_POLICY`), which is indirectly evaluated from the `POSTROUTING` chain. An example rule follows:

```
iptables -t mangle -A NFSHUNT_POLICY -p tcp --dport 5000
    -m conntrack --ctstate RELATED,ESTABLISHED -j MARK
    --set-xmark 0x10010000/0x100f0000
```

The above rule sets a value of 1 (defined in the default controller configuration to trigger a shunt) when a TCP flow is matched with destination port 5000.

We now describe the rules necessary for the functioning of the prototype (in addition to shunting policy):

Netfilter's rule traversal is directed from the built-in `PREROUTING` chain of the mangle table to a chain defined for the prototype's pre-routing rules: `NFSHUNT_PRE`:

```
iptables -t mangle -A PREROUTING -j NFSHUNT_PRE
```

The first rule in `NFSHUNT_PRE` copies the connection mark from the connection tracking object associated with the packet being inspected (if one exists) to the packet mark (the *restore* operation):

```
iptables -t mangle -A NFSHUNT_PRE -j CONNMARK --restore-mark
```

In order to match the correct fast-path ports to the slow-path ports on which the connection enters and exits the prototype prior to shunting, it is also necessary to encode the slow path bridge input and output interfaces into the mark. This is done in two stages (before and after the kernel's bridging decision).

A rule checks bit 30 of the mark value (part of the flag field), to determine whether the input interface has already been recorded in the mark. If not, it directs rule traversal to the `NFSHUNT_PRE_PD_IN` chain, which has one rule to match every possible input interface. These rules alter the mark value at bits 24 to 27 to a value unique to that interface (also configured in the shunting controller), as well as setting the flag checked

Doodle

by the previous rule. In the example rules below, the interface `p1p1` corresponds to a mark of 1, and `p1p2` to a mark of 2:

```
iptables -t mangle -A NFSHUNT_PRE_PD_IN -m physdev --physdev-in p1p1
    -j MARK --set-xmark 0x41000000/0x4f000000
iptables -t mangle -A NFSHUNT_PRE_PD_IN -m physdev --physdev-in p1p2
    -j MARK --set-xmark 0x42000000/0x4f000000
```

Rule traversal then returns to the default Netfilter tables and chains, in which the administrator may have defined any other firewall configuration. Once this reaches the `POSTROUTING` chain for the mangle table, another rule again directs traversal to a chain defined for prototype's use, `NFSHUNT_POST`, which contains three rules. The first again checks a flag at bit 29 of the mark value to test whether the output interface is recorded, and if not, directs traversal to the `NFSHUNT_POST_PD_OUT` chain which marks bits 20 to 23:

```
iptables -t mangle -A NFSHUNT_POST_PD_OUT -m physdev
    --physdev-is-bridged --physdev-out p1p1 -j MARK
    --set-xmark 0x20100000/0x20f00000
iptables -t mangle -A NFSHUNT_POST_PD_OUT -m physdev
    --physdev-is-bridged --physdev-out p1p2 -j MARK
    --set-xmark 0x20200000/0x20f00000
```

In this case, it is necessary to include a check to discriminate bridged from routed packets, due to the design of Netfilter's logic (hence the use of `--physdev-is-bridged`).

Next, `NFSHUNT_POST` directs evaluation to the `NFSHUNT_POLICY` chain described above, and finally it copies the mark from the packet back to the connection tracking mark (the *save* operation):

```
iptables -t mangle -A NFSHUNT_POST -j CONNMARK --save-mark
```

3.6 Prototype controller implementation

3.6.1 Slow path interface

Interaction with Conntrack is via the `conntrack` user-space utility. The shunting controller starts an instance of `conntrack` with the `-E` parameter inside a thread, and

then reads a sequence of connection tracking events from the standard output stream in Extensible Markup Language (XML) format.

Connection tracking events describe the creation, destruction or modification of connection tracking objects in the kernel. When an event includes the mark attribute (obtained from the iptables connection mark), the controller examines it in further detail. Certain events are ignored as they cannot be used to trigger shunting. These include destruction or modification events where the state of the TCP flow changes to `FIN_WAIT`, `LAST_ACK` or `TIME_WAIT`. If the flow is not ignored, and it contains the necessary layer 4 header information (TCP ports), the controller checks for the presence of a matching action in the flow mark.

The flags, flow mark and information about the slow path physical input and output interfaces are used by the controller's core logic to make shunting decisions.

In addition to monitoring connection tracking events, the controller core must also be able to delete connection tracking objects for flows that have been shunted or blocked. This is achieved by executing the `conntrack` utility with the `-D` parameter and specifying the 5-tuple describing the TCP connection.

3.6.2 Fast path interface

Interaction with the fast path is via the POX OpenFlow controller framework. The prototype itself is implemented as a module within the framework; so while, from the point of view of the design, OpenFlow is a module, the implementation calls POX functions directly from the core logic of the prototype.

Upon start-up, POX listens for connections from OpenFlow switches on the default TCP port for this protocol. The fast-path switch is configured with the IP address of the out-of-band connection to the slow path server (where the controller runs). When the switch connects to the controller, a connection handler runs which performs the following actions:

1. If the controller is configured to delete existing flow entries, it does so by sending a flow modification to the switch. This allows the prototype to begin managing the fast path with a known state.

2. If the default packet flow is configured via the slow path (`default_no_shunting`), then the controller iterates through the port groups specified in the configuration file, and sends two flow modifications to add flows between the slow and fast path ports (one flow per direction). This is equivalent to logical patching between the slow and fast path ports.

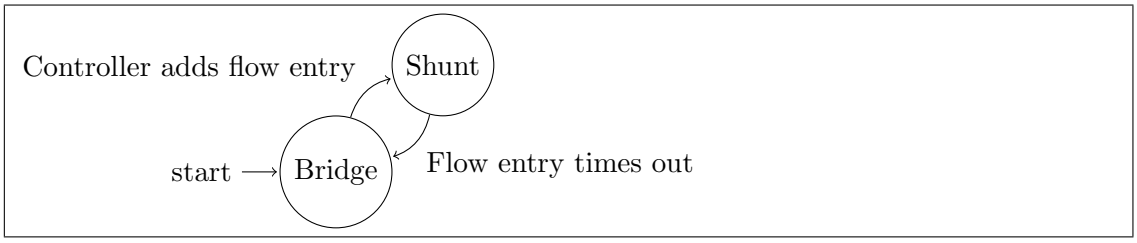

Figure 3.4: Per-connection state machine of the forwarding path

3. A thread handling the slow-path interface is started.

Additional event handlers generate logging entries for flow statistics, flow removal events and connection tear-down.

The core logic of the controller calls POX functions directly to add flow-entries for shunting and blocking.

3.6.3 Controller core logic

The core of the shunting controller receives events from the slow-path module, which have been validated to contain the meta-data added by the Netfilter rules configured for the functioning of the prototype.

If the action indicated by the flow mark is to shunt, the controller will add two flow specifications (one for each direction of the flow) via the fast-path interface to by-pass the slow-path. These flow specifications include layer 2, 3 and 4 header match fields, an action to output via the corresponding fast path ports, and the configured idle time-out. For blocking actions, the only difference in the specification is that the output action is omitted (which leads to an implicit packet drop).

Finally, the controller includes the instruction for the switch to inform the controller when flows are deleted (for informational purposes), and each flow is annotated (in the switch) with a cookie value that corresponds to the connection tracking object ID from the slow path that triggered the shunting action.

The state of the fast path, therefore, transitions between shunting or blocking and forwarding packets via the slow-path. Figure 3.4 illustrates how flow entry programming and time-out transition between the two states on a per-connection basis.

After a shunt or block is installed, the connection tracking object is deleted via the slow-path interface. The reason for this is three-fold:

1. Since the slow-path will no longer forward packets for the flow, the connection tracking object serves no purpose for Netfilter, hence deleting it frees up valuable

kernel memory.

2. If a flow stalls for longer than the idle-timeout of the flow specifications implementing the shunt, and then resumes, subsequent packets will re-appear on the slow path. At this point, it is necessary for the controller to re-install the shunt, but the state of the connection tracking will not necessarily change in a manner that will trigger an event visible via the slow-path interface. Deleting the entry forces the slow-path to re-create the entry, which is guaranteed to be visible to the controller.

3. If the Linux kernel is configured to be strict in the stateful tracking of TCP sequence numbers, the shunting action will result in the connection tracking object's state being invalid once the connection proceeds via the fast path. In this scenario, if a connection-resumption is attempted (as described above), then Netfilter could terminate the connection.

3.6.4 Configuration module

Installation-specific information is provided by the configuration module. This functionality is implemented by reading a single text configuration file at start-up of the controller. The location of the configuration file can be specified as a command-line parameter during start-up of the controller:

```
sudo ./pox.py nfshunt --configurationfile /path/to/file.json
```

If not specified, the controller will attempt to open a file named `nfshunt.json` in the current directory when the controller is started. Note that the shunting controller must run with root privileges to access and modify Netfilter connection tracking information (thus the use of the `sudo` command).

JavaScript Object Notation (JSON) was chosen as the configuration file syntax, as it is both simple for administrators to read and write, and it supports nested structures. The standard Python library's built in-support for JSON parsing simplifies the code and reduces dependencies.
The following general configuration parameters are available:

- `delete_flows_on_startup` (default is true): causes the controller to clear the fast path flow table on start-up.

- `delete_flows_on_shutdown` (default is true): same as above, but runs at shutdown.
- `default_shunt_timeout` (default is 10 seconds): sets the idle timeout for shunt flow entries. Making this longer increases the flow table contention, but reduces the load on the controller for connections that stall.
- `default_block_timeout` (default is 10 seconds): same as above, but specifically for block actions (as opposed to shunt actions).
- `default_no_shunting` (default is true): if this is true, then the controller will configure flows to send traffic via the slow path by default (at start-up), and the *ignore* action does nothing (hence ignored flows will continue via the slow path). Setting this to false only makes sense if `delete_flows_on_startup` is false, and the fast path is pre-programmed for some useful forwarding action.

Two nested sections must exist in the configuration file:

1. `ports` defines a list of groups that tie together three parameters:
   (a) The `fast` port on the OpenFlow switch: the port that connects to the network.
   (b) The `slow` port on the OpenFlow switch connecting to the bridge interface on the slow path to be used for traffic to and from the `fast` port.
   (c) `physdevin` is the unique number configured in the iptables rules for marking of traffic entering via the bridge interface on the slow path, to be used for traffic to and from the `fast` port.
2. `mark_actions` maps numbers configured in the iptables rules for marking flows for the corresponding actions of shunting, blocking and ignoring.

3.6.5 Logging module

Logging information generated by the shunting controller is intended to be used in one of two modes: by default, the logging level provides messages that would be of interest to an administrator. If the log level is set to DEBUG for the NFShunt POX component (by appending `log.level --nfshunt=DEBUG` to the command line parameters), extra messages will be emitted:

- Information about flow events from Netfilter Conntrack that are ignored as they are deemed irrelevant.
- Flow statistics dumped from the fast-path after each flow programming action.

In addition to troubleshooting, the intent of the DEBUG logging mode is to output information about the functioning of the controller that will be recorded for the purpose of evaluating its performance as part of the research.

Additional logging behaviour can be configured by making use of POX's built in capabilities. Examples of this would be to change the logging prefix format by appending additional command line parameters:

```
log --format="[%(asctime)s]␣%(module)s␣%(levelname)s␣%(message)s"
```

Or redirection of log messages to a file:

```
log --file=nfshunt-pox.log
```

3.7 Conclusion

In this chapter, we defined research questions and a research approach that required the construction of a prototype hybrid firewall. The design, implementation and configuration of our prototype was described in detail. In the next chapter, we proceed to evaluate our design through experimentation.

Chapter 4

Experimentation

In the previous chapter, we outlined our research approach and described the construction of the NFshunt. However, this alone does not answer the research questions posed in section 3.1. In this chapter, we describe our experimental methodology and report the results of the experiments.

We proceeded to evaluate our prototype's performance, firstly to determine if the implementation works correctly (validating the design's feasibility), and whether it is suitable for the Fast Data Transfer (FDT) use-cases normally catered for by science demilitarised zone (DMZ)s.

Experiments were designed to compare the prototype's network performance to that of a high speed firewall employing a traditional design. We ensured that our experiments would produce the data required for our analysis (which, in turn, addresses our research questions), by mapping out our research approach in Table 4.1.

	Research question	Data required	Experimental method	Analysis
1	How can generic and standardised off-the-shelf components be composed to create a high performance (hybrid) firewall?	Evidence that the shunting mechanism works in an implementation of the design.	Implement the prototype and test the shunting mechanism with generated network connections.	If Transmission Control Protocol (TCP) connections can be tracked by Netfilter during establishment, and then be offloaded to the OpenFlow switch, the design is validated.
2	What is the packet-loss performance of a hybrid firewall (is it equivalent to a *clean network path*?)	Packet loss rates for both the prototype and an environment similar to the science DMZ.	Generate FDT-like load, and measure packet loss for both the prototype and a direct-switching configuration.	Test the hypothesis that the packet loss of the prototype is equivalent to direct-switching.
3	How would a hybrid Software Defined Networking (SDN)-based shunting firewall compare to a traditional firewall in terms of price-performance ratio?	Packet loss rates for both the prototype and a firewall representative of its class.	Generate FDT-like load and measure packet loss for both the traditional firewall and a direct-switching configuration.	Test the hypothesis that throughput performance of the prototype is greater than the traditional firewall.
		Estimates of cost for all equipment components.	Implement the prototype, then estimate total cost based on the bill of materials.	Compare the hardware costs, consider price/performance.
4	What are the operational and maintenance benefits and drawbacks of a hybrid firewall?	Insight into the requirements on an operator of the prototype.	Document the configuration, troubleshooting and securing of the prototype during the experiments above.	Compare the aspects of the prototype's operations to traditional firewall operations.
5	Have we developed a firewall architecture that meets the performance requirements of data-intensive science?	Outcome of the analysis for research questions 1, 2 and 3 above.		If the analysis of 1, 2 and 3 favour NFShunt, we conclude that the overall research objective has been met.

Table 4.1: Research approach: mapping questions to method and analysis

4.1 Experimental methodology

The objective of the science DMZ design is to create a loss-free network path based on the observation [22] that a firewall cannot perform loss-free forwarding for permitted TCP connections in FDT scenarios. In our performance evaluation, we focused our attention on the TCP performance so that we could relate the results back to the science DMZ use-case. Replaying of captured traffic or network layer simulation were not suitable methods for test traffic generation when evaluating TCP performance. We therefore chose application-layer load generation, which indirectly produces network test traffic.

4.1.1 Experimental design choices

Our experiments were designed to emulate a typical data-intensive science infrastructure scenario. Due to established best common practices for maximising network performance with high bandwidth-delay products, the experiments evaluate approximate best-case performance with moderately difficult circumstances. Our objective for the experiments was an unbiased comparison, which required an upper bound to the performance tuning that was done. We sought to achieve this balance through specific experimental design choices:

- **Focus on TCP:** TCP was selected as the transport protocol for test traffic, since the congestion control mechanism would be affected by the packet-loss performance of the tested firewalls. TCP is also the most widely used transport protocol, therefore the choice reflects realistic applications.

- **Large packets:** Jumbo frames are commonly used to reduce the packet processing rate, and tuning this parameter in our experiment was a sensible optimisation. We expect that smaller frames will magnify the effect that any difference in packet-loss performance will have on the results of the comparison.

- **Limited hardware tuning:** the traditional firewall was tuned for single-flow performance according to the advice of the equipment vendor (described in subsection 4.1.3). Since NFShunt aims to utilise generic hardware, no model-specific tuning was performed on the slow and fast path components.

- **Minimal firewall rule-set:** Both firewalls were tested with the default policy. Specifically excluding the complexity of the rule-set from the factors in the experiments created a best-case scenario for both, and established a performance baseline. While this is not realistic, we expect that a complex rule-set will favour

the shunting design and magnify the performance difference we hypothesised (because it would increase the utilisation of CPU and bandwidth required for state tracking – a problem that shunting avoids).

The traffic-generating servers used in the experimental setup were also tuned to emulate a typical end-host in a science DMZ (described in sub-section 4.2.2).

These choices support the research objective of exploring the limitations of traditional firewalls and our hybrid firewall design.

4.1.2 Lab equipment

Traffic generation (including synthetic delay) relied on two Dell Poweredge servers running Linux, each equipped with Intel 82599-based dual-port 10GBase-Ethernet Network Interface Controllers (NICs). We used a Cisco ASA 5585 firewall for comparison to the prototype firewall. Interface counter values on the Cisco were recorded by a script accessing the system via the serial console instead of in-band management via the test network (which would have complicated packet loss measurements).

4.1.3 Lab test configurations

With advice from Cisco technical support engineers, some changes were made to the factory default configuration of the Cisco:

- ASA firewall software was upgraded to version 8.47.
- The firewalling mode was set to transparent.
- The Maximum Transmission Unit (MTU) for interfaces was increased to 9216 bytes.
- "jumbo-frame reservation" was enabled.
- TCP Maximum Segment Size (MSS) clamping was disabled.

Four test configurations were used for experiments:

- **Configuration one** connected the test servers directly, and served to establish the best-case throughput achievable using the experimental hardware. Our experiment required endpoints with sufficient CPU capacity, as well as peripheral bus and memory bus bandwidth, to support near-line rate TCP transfers. During these test runs, Internet Protocol (IP) stack tuning was done to optimise TCP throughput with or without synthetic delay.

- **Configuration two** inserted the Cisco firewall in-line between the test servers via one set of 10Gbase-Ethernet NIC interfaces.

- **Configuration three** connected the Pica8 switch via another set of 10Gbase-Ethernet NICs, and the switch was manually configured for direct forwarding of frames between the test servers without the slow-path interface (no shunting mechanism).

- **Configuration four** allowed the Pica8 switch to be controlled by NFShunt.

The duration of individual TCP throughout tests was 60s, which was sufficient for TCP to reach maximum throughput even at 400ms simulated Round-Trip Time (RTT).

Configurations two through four were used for performance evaluation, which is described in the remainder of this chapter.

4.2 Factors and levels

Round trip delay is an important factor in TCP performance. Many science applications require a small number of high volume, long distance data transfers. For example, the Square Kilometre Array (SKA) will require data transport from a number of African countries to South Africa, and from Australia and South Africa to Europe and North America. For our experiments, synthetic delay values were chosen to represent typical round-trip times for intra-African (100ms), European-South African (200ms) and North American-South African (400ms) connections. These are realistic for the distribution of data from the SKA mid-frequency radio telescope to be constructed in South Africa. Factors (and their respective levels) tested during experiment runs were:

- **Selected middle-box:** three configurations exist to allow the performance comparison objective of the experiment: direct switch, the prototype firewall (NFShunt) and the traditional firewall (Cisco ASA 5585).

- **Synthetic delay:** since the performance reducing effect of packet loss is dramatic in the presence of delay, it is varied (from no delay to long-distance network delays of 100ms, 200ms and 400ms round-trip-times) to observe this effect in the test configurations. The Linux netem [35] queue discipline was used to emulate half of the transmit delay on each of the two test servers.

Tests for each combination of factors and levels was repeated 100 times.

4.2.1 Measurements

The following observable performance characteristics were measured in each experiment run:

- **Absolute TCP throughput**: individual as well as aggregate TCP throughput was measured using the iperf3 network throughput testing utility [24].
- **Dropped packets**: packets dropped between the test endpoints were measured by comparing NIC frame counters on the servers to the frame counters of the device under test. This also allowed for locating the cause of the loss.
- **TCP stack behaviour**: Web100 [48] instrumented Linux kernels were used, and a custom application logged snapshots of the Web100 variables associated with the test connection every 100ms, to allow analysis of the TCP stack behaviour during transfers.

The above measurements were annotated with events relevant to the shunting behaviour of the prototype firewall, in particular, the point in time when a shunting instruction is sent, and the last packet is switched on the slow-path. Network Time Protocol was used to synchronise the system clocks of the two test servers and the slow path server.

4.2.2 Validation of test procedure

Performance tuning was performed on the end-hosts with direct network switching configured via the OpenFlow switch (in order to validate the test procedure prior to the experiments). Host IP stack tuning followed conservative guidelines applicable for modern Linux kernels (optimised to allow for round-trip delays greater than 400ms at 10 Gbit s^{-1} link capacities). Parameters used are listed in Table 4.2. No hardware or driver-specific parameters were changed, as this would deviate from typical data transfer scenarios that the experiments were intended to simulate.

The netem Linux QoS module was configured on the respective network interfaces to introduce fixed transmit delays, which together amounted to the total desired synthetic round-trip delay for each experimental run. It was also necessary to adjust the default QoS buffer of 1,000 to 100,000 packets, in order to prevent packets being dropped in the kernel transmit path.

With this configuration, the testbed was capable of consistent TCP transfers at throughputs in excess of 9 Gbit s^{-1} for 0ms, 100ms and 200ms RTT (included as the *direct*

Tuning parameter	Value
NIC transmit queue length (txqueuelen)	10,000 packets
TCP socket buffer size auto-tuning maximum (tcp_wmem/tcp_rmem)	500 MB
Network interface MTU	9,000 B

Table 4.2: Host tuning for test servers

series of performance tests in the next section). These results provided a performance baseline and context for the experiments that followed.

4.3 Experimental results

In this subsection we report the results of our performance experiments. Mean connection shunt timing, TCP throughput and packet loss values were calculated, and then used to test hypotheses that the performance of the systems we compared differ. We also report the magnitude of performance differences (all of which were found to be statistically significant).

We considered both the speed with which the prototype moves packet forwarding between the slow and fast paths (the shunting mechanism), and the externally observable packet forwarding performance during each test.

4.3.1 Shunting mechanism

The shunting mechanism of the prototype was profiled to determine how quickly flows can be shunted. The timing of events were recorded for shunted TCP connections (measured in seconds elapsed since the originating test server sent the SYN packet establishing the connection). Table 4.3 summarises the results of 100 tests for each combination of factors. Two of the events are timestamps recorded in the shunting controller logs: the time when the controller receives a the connection tracking event from the kernel (via the `conntrack` utility) and the time when all the POX library calls to install shunting flow specifications are executed. The third event is the time at which the last packet is slow-switched, as recorded by a packet capture on the show path server.

Event description	Time from SYN (ms)	Standard deviation (ms)
Controller detected flow mark	3.3	0.30
End of flow programming	58.0	0.69
Last packet slow-switched	75.8	0.04

Table 4.3: Shunting event performance

Synthetic RTT	Test	Mean data rate $\mathrm{Gbit\,s^{-1}}$	Mean packet loss %
None	Direct switching	9.944	–
	Shunting	9.923	0.00
	Cisco ASA5585	9.838	0.05
100 ms	Direct switching	9.678	–
	Shunting	9.690	–
	Cisco ASA5585	5.337	0.10
200 ms	Direct switching	9.332	–
	Shunting	9.334	–
	Cisco ASA5585	4.197	0.21
400 ms	Direct switching	5.978	–
	Shunting	5.957	–
	Cisco ASA5585	3.094	0.21

Table 4.4: Single flow forwarding performance

4.3.2 Forwarding performance

The results of the single-connection TCP performance tests are summarised in Table 4.4. Throughput as measured by the *iperf3* utility is reported, while the packet loss rate is calculated by comparing transmit and receive Ethernet NIC Media Access Control (MAC) frame counters on the test servers.

No packets were dropped in the *direct* series of tests, confirming that the OpenFlow switch is capable of non-blocking line-rate switching. In tests of the prototype and the traditional firewall (where packet loss was observed), the NIC MAC frame counters were compared to the frame counters of the device under test to locate the cause of the loss. Both firewalls were found to be receiving but not forwarding all frames (in other words, unlike direct switching, both firewalls dropped *some* packets).

Throughput tests on the Cisco firewall were repeated with two and four simultaneous connections, in order to study the effects of TCP window scaling and internal load-balancing beyond a single flow. These results are shown in Table 4.5.

Figure 4.1 shows the congestion window and TCP throughput during the first 400 milliseconds of three individual (and independent) experiment runs at 200ms RTT, for the Cisco ASA 5585, NFShunt and direct switching tests. The circles in the congestion window plot indicate retransmissions (each circle's area is scaled to the log of the number of packets retransmitted during a 100ms sample). While this figure does not summarise the results, it provides a visual illustration of the effect packet loss has on the TCP performance of the traditional firewall, compared to direct switching and the prototype firewall. Shunting occurs before the first data point on the graph. We do not directly indicate the timing of packet loss on this graph, as the data was not available from Web100. Peaks in the sawtooth shape of the congestion window plot (for the Cisco test) correspond to short bursts of retransmissions.

	Mean data rate Gbit s^{-1}		
Flows	**No delay**	**100** ms **synthetic RTT**	**200** ms **synthetic RTT**
One	9.838	5.337	4.197
Two	9.975	9.471	8.724
Four	10.011	9.866	9.430

Table 4.5: Cisco - multiple flow forwarding performance

4.3.3 Network performance comparison

We tested the hypothesis that the direct switching performance differs from the prototype's performance, and that the prototype's performance is different to the Cisco firewall's performance. The Wilcoxon rank-sum test was applied to mean throughput and packet loss measurements of the respective devices at different RTT values. We interpret that tests with $p > 0.05$ indicate no significant difference while those with $p < 0.05$ do. Table 4.6 and Table 4.7 summarise the results.

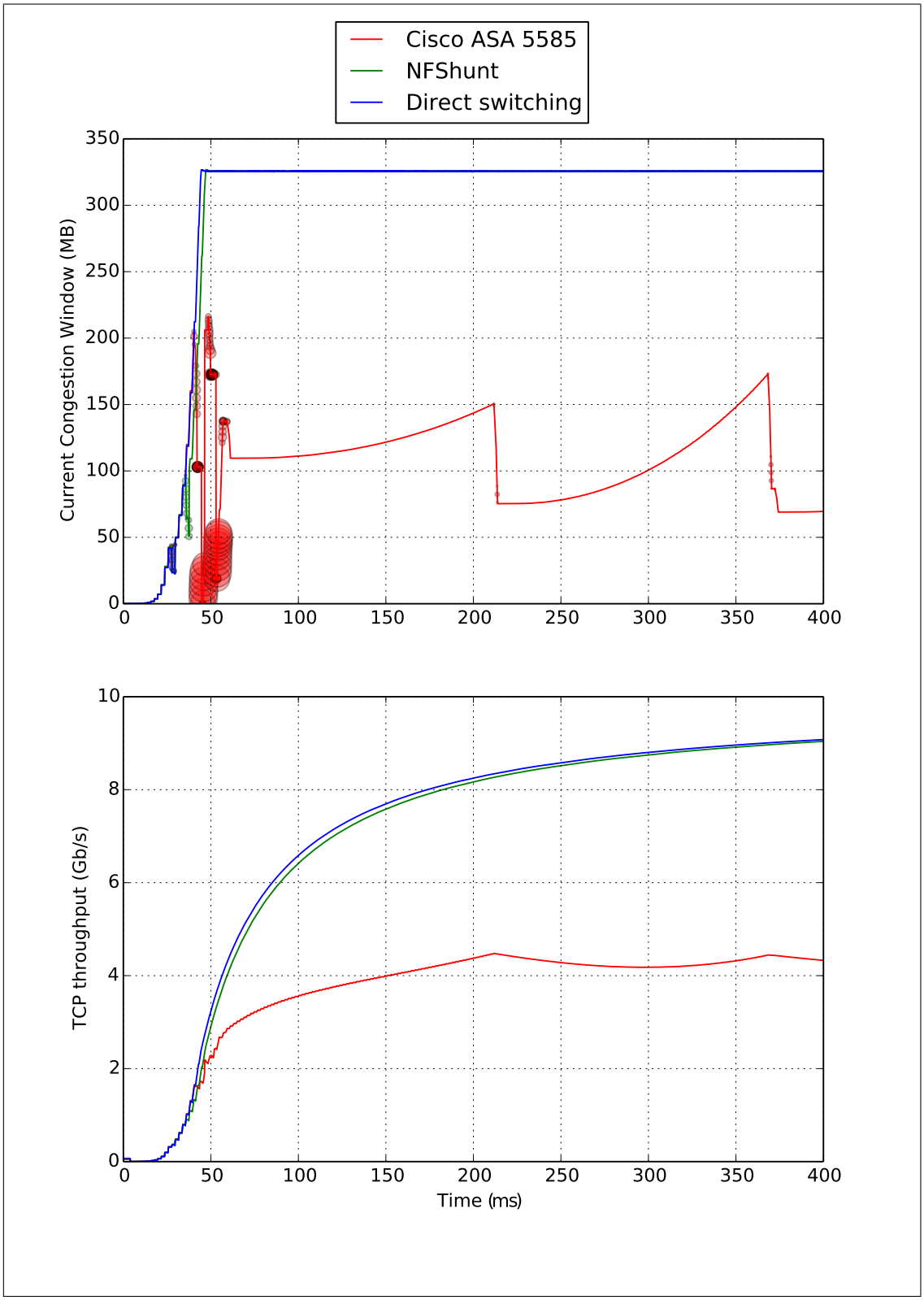

Figure 4.1: Comparison of congestion window and throughput for three independent test runs at 200ms RTT.

RTT	Difference in mean data rate Gbit s^{-1}	Difference in mean packet loss %
None	significant difference of 0.021 Gbit s^{-1}	no significant difference
100 ms	no significant difference	identical (no loss)
200 ms	no significant difference	identical (no loss)
400 ms	no significant difference	identical (no loss)

Table 4.6: Tests of the hypothesis that direct switching and prototype performance differ

RTT	Difference in mean data rate Gbit s^{-1}	Difference in mean packet loss %
None	significant difference of 0.084 Gbit s^{-1}	significant difference of 0.051%
100 ms	significant difference of 4.353 Gbit s^{-1}	significant difference of 0.096%
200 ms	significant difference of 5.137 Gbit s^{-1}	significant difference of 0.207%
400 ms	significant difference of 2.863 Gbit s^{-1}	significant difference of 0.209%

Table 4.7: Tests of the hypothesis that prototype and Cisco ASA 5585 performance differ

Chapter 5

Discussion

In this chapter, we analyse our prototype implementation and experimental evaluation results, with the purpose of answering the research questions posed in section 3.1 and mapped to our research methodology in table 4.1.

5.1 Analysis of the prototype implementation

We consider the feasibility of the implementation in order to provide the analysis required to answer research question one in table 4.1:

Our experimental results show that controlling a shunting mechanism using the Netfilter firewall rule-set works, but we identified some shortcomings of this approach.

The use of connection marking to specify actions makes for a complicated iptables rule-set, and imposes structure that may require existing firewall policy to be re-written by the administrator. A more elegant implementation would add a new iptables action type to Netfilter, allowing rules to be written in the following style (a rule to trigger hardware bypass SSH connections once they are established):

```
iptables -F FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -m tcp
    --dport 22 -j HARDWARE --hardware-action=bypass
```

This would require the development of a kernel module – a non-trivial undertaking for a user-interface improvement!

We found attempting to integrate the POX controller with connection tracking via the user-space Netlink libraries difficult, due to poor Python bindings for the Netlink connection tracking protocol. Instead, our implementation spawns the `conntrack`

userspace tool as a subprocess of the controller.

The use of OpenFlow for the fast-path interface greatly simplified implementation of the prototype. While only OpenFlow 1.0 capability is required for the trivial actions of selecting output ports based on 4-tuple matches, this would have been difficult or impossible to achieve on Commercial Off-The-Shelf (COTS) hardware without OpenFlow. The prototype was also tested successfully with Mininet [43], demonstrating that it is compatible with multiple OpenFlow dataplanes.

Until very recently,[1] OpenFlow lacked a match type for TCP flags [80]. While support for TCP flag matching is available in open vSwitch (OVS), hardware is not yet available with OpenFlow 1.5 support. This prevents the prototype from detecting TCP connection tear-down in the forwarding plane, and necessitates the use of idle-timeouts for flow rules to clean up the fast-path when connections complete. The disadvantage of the idle-timeout approach is that stalled connections will be routed via the slow-path briefly, if and when they resume.

Despite some minor obstacles detailed above, our implementation demonstrates the feasibility of key aspects of NFShunt's design: hardware acceleration, integration into iptables, and the COTS toolkit approach.

5.2 Experimental performance analysis

Research question two in table 4.1 requires the outcome of our hypothesis tests to be analysed to support conclusions about the performance merits of our prototype for Fast Data Transfer (FDT) applications.

Our experiments found that the delay of the slow-path interface (detecting new connections as tracked by the kernel) is small (approximately 3ms). Sending shunting flow specifications to the switch makes up the majority of the delay between the first and the last packet being slow-switched (approximately 60ms out of 75ms).[2] These results are provided in table 4.3. Based on examination of Web100 traces for individual connections (sampled on one of the test servers every 100ms), we conclude that no congestion events are caused by the shunting mechanism, because no such events are reported in the first sample (which would include all the slow-switched packets). These results show that our design can respond fast enough to the establishment of individual connections.

[1] OpenFlow 1.5, published in December 2014 includes a Transmission Control Protocol (TCP) flags match type.

[2] Flow specification programming delay can be attributed the controller itself, the control network, the OpenFlow agent and the switch ASIC, as well as the respective operating systems. The relative contributions of each component was not investigated.

The Pica8 OpenFlow switch used for our experiments was able to forward all connections without packet loss (table 4.4). Despite the host tuning performed, a notable TCP performance drop-off was observed at 400ms. This suggested that TCP throughput would not be the best (albeit direct) measurement of firewall performance in our experiments.[3] We therefore ensured that packet loss could be accurately measured and located to specific network elements.

An explanation offered by ESNet [22] for the tendency of traditional firewalls to drop packets in FDT applications, as well as one of our initial assumptions (section 1.2.1), predicted that the total throughput would be approximately quantised by the maximum per-connection throughput (the quantum being less than the speed of the fastest firewall interface).

When there was no synthetic delay, the Cisco ASA consistently dropped a small percentage of packets (table 4.4), but was able to firewall a single connection at nearly the maximum speed achievable with the test setup (near 10Gbps). From this we infer that the Cisco firewall we tested is not relying on load-balancing multiple connections over processing elements in the forwarding plane to reach *aggregate* 10Gbps throughput.

Transferring data over simultaneous TCP connections was quite effective at overcoming the high latency TCP slow-down due to loss introduced by the firewall (table 4.5), and thus utilising nearly the full link bandwidth at moderate Round-Trip Time (RTT). This supports ESNet's alternative suggestion [23] that the packet loss we found is due to traditional firewall input buffers not being optimised for large flows. This insight also reveals a limitation of our study: the experimental design choice to focus on TCP did not allow more extensive exploration of the FDT problem space (for example, by studying transport protocols that circumvent TCP's limitations).

A very small amount of packet loss was observed in the tests of the prototype with no synthetic delay (table 4.4). While this is not a statistically significant difference from the direct switching tests (according to our interpretation of p-values), a 21Mbps difference in throughput was found to be statistically significant. At 100ms, 200ms and 400ms, there were no differences between the measurements of the prototype and direct switching (table 4.6 and table 4.7). These results are consistent with a model where the slow-path phase of the connection has little effect on performance (with TCP, this phase covers the connection setup, and at worst the slow-start phase of data transfer) because the packet-rate is low enough to software switched without packet loss.

Under load, the difference in packet loss, combined with moderate RTT, results in

[3] As an end-to-end performance measure, TCP throughput could be influenced by factors not controlled in our experments.

a significant degradation of throughput performance of the tested Cisco firewall model to other approaches (such as our NFShunt prototype or that of a science DMZ). This result allows us to answer the research questions: **our hybrid design achieves high performance and exhibits the loss-free forwarding behaviour required for data-intensive science applications**.

5.3 Operations and maintenance analysis

Now we examine technical aspects of the prototype and traditional firewall designs, studied in the context of operations and maintenance. This analysis supports conclusions to research question four (table 4.1). The Fault, Configuration, AAA, Performance and Security management (FCAPS) framework [38] is used to consider the different activities involved in operating firewalls.

5.3.1 Fault management

NFShunt's open architecture allows for more in-depth troubleshooting of the individual components compared to a traditional firewall. Open source software can be audited, and bugs in NFShunt's code can be found and fixed by the end-users. The downside of this tool-kit approach is that administrators would require knowledge to troubleshoot Netfilter, Linux bridging and the OpenFlow agent to locate faults in the firewall itself. A traditional firewall typically offers a unified interface, which aids troubleshooting of simple problems, but could obscure low-level details that only vendor technical support may be able to access and interpret. Compared to pure software firewalling, the shunting approach is more complex: it introduces additional components to the forwarding path, more configuration and another software component. We expect that this additional complexity would result in a less reliable system.

5.3.2 Configuration management

NFShunt does not offer standard interfaces and protocols for network configuration management, like Simple Network Management Protocol (SNMP), or more recently Netconf. Netfilter is, however, very widely used, and many tools and interfaces exist to manage iptables rule-sets. The design that integrates shunting policy in the iptables rules, rather than maintaining a separate configuration, reduces the complexity of managing its configuration. For configuration of the controller – the design has chosen a format that is concise and easy for administrators to both read and write.

5.3.3 Account management

The shunting design implicitly requires network traffic to be accounted separately on the slow and fast paths. While shunted packets are no longer visible to tools that might normally be used on Netfilter firewalls, NFShunt exposes flow statistics via the logging module. It is possible to integrate these statistics into existing reporting and accounting tools to achieve feature parity with a traditional firewall design.

5.3.4 Performance management

Other than allowing the usual process of measurement, planning and provisioning of resources to meet the demands on the network (common with traditional firewalls), NFShunt has the advantage of being able to selectively scale up throughput via shunting. This additional capability is the primary contribution of the prototype, and is the advantage of NFShunt over alternative approaches.

5.3.5 Security management

NFShunt's advantages contribute to the network's resources to implement security without compromising on performance. The security of NFShunt itself is composed of Linux and the OpenFlow agent's security measures.

The NFShunt controller requires administrator level rights to the Linux system for access to the connection tracking module. While not implemented in our prototype, it is possible to compartmentalise functions of the controller to reduce privileges of all components but the slow path interface. Configuration of Netfilter via the `iptables` utility is performed by a user with administrative rights.

The prototype implementation did not make use of OpenFlow protocol security beyond static address configuration. The controller could be enhanced to use Transport Layer Security (TLS) encrypted and authenticated connections for agents that support this capability, however, the Pica8 switch used in our research did not include secure control channel support.

5.4 Price-performance comparison

Finally, consider both the up-front equipment cost, as well as the ongoing costs of operating a firewall, in order to answer research question three as per table 4.1:

5.4.1 Capital cost

Cost of infrastructure is important as high-throughput science expands from the preserve of the few to that of the many.

The capital cost[4] of the Cisco ASA5585-X SSP-60 firewall used for testing varies between R1M and R2M, depending on configuration, licensing and discounts applied. Since this particular firewall's features and capabilities far exceed the requirements of our tested scenario, a smaller configuration was chosen for the purpose of pricing comparison with the prototype. The Cisco ASA5580-20 configured with two 10G interfaces matches the prototype firewall more closely. Table 5.1 shows the capital cost of our prototype compared to the specified traditional firewall.

NFShunt Prototype		Traditional Firewall	
Component	Cost	Component	Cost
FitPC	R7291	Cisco ASA5580-20	R720 000
Pica8 P3290	R32 005		
Total	R38 296	Total	R720 000

Table 5.1: Capital cost comparison

5.4.2 Operational cost

Real world implementations of NFShunt would allow the operational costs to be quantified and compared, but this is not within the scope of our research. Instead, we analyse the technical aspects of our design that would impact operations and maintenance. As such, our research methodology for studying the operational costs is qualitative.

In general, it is expected that the additional complexity of the NFShunt prototype (described in section 3.5) is likely to increase its operational costs relative to operating traditional firewalls (though the absolute cost could still be lower). Some traditional firewall vendors generate additional revenue from their products by charging license fees as the customer requires additional software features or artificially limited capacity. From the customer's perspective, these are operational costs that would not necessarily apply to a hybrid firewall composed of COTS hardware and Free and Open Source Software (FOSS).

[4]ZAR prices at various exchange rates from 12 to 14 ZAR/USD

5.4.3 Analysing cost performance

The above analysis of the costs associated with our toolkit-like design support an answer to the research question: the prototype is clearly low-cost compared to the class-representitive traditional firewall we studied. With higher performance, NFShunt would deliver a lower price-performance ratio. The total cost of ownership is unknown (and will vary, even amongst deployments of traditional firewalls). Based on our analysis in section 5.3, we argue that the cost of operating NFShunt in production would be similar to a traditional firewall, as the benefit of its flexibility comes at the cost of increased complexity.

5.5 Limitations of the research

Some convenient experimental design choices resulted in minor shortcomings of the research:

- If we assume that traditional firewalls optimise buffers for the distribution of packet sizes typical to Internet traffic, then varying the Maximum Transmission Unit (MTU) value in our TCP experiments may have revealed whether the best-practice of using jumbo frames in FDT applications has a detrimental effect on firewall performance.
- Connection-rate testing would likely have shown up processing delay limits of the shunting mechanism which could severely limit the suitability of our design for applications where off-loaded connections are numerous (unlike the FDT scenario). We expect traditional firewalls would out-perform our prototype in such cases, but since we did not study this dimension of scalability, this remains speculation.

The major shortcomings of our research, however, were due to the deliberately restricted scope:

- The lack of a real-world NFShunt deployment case-study restricted our investigation of operational aspects to arguments based on the theoretical implications of our prototype's design. Rich performance data from production application traffic may also have provided further insights and strengthened the reliability of our results.
- Testing high-performance traditional firewalls from a range of models and manufacturers would have allowed general conclusions to be drawn about the performance of firewalls used in data-intensive research infrastructure.

- Finally, broadening the scope of the research to study multiple file transfer tools and transport protocols, as well as to consider the information security threat model of science DMZs, may have generated knowledge useful to end-users and operators.

Chapter 6

Conclusion and future work

In this chapter we provide a conclusion to our research, and explore possible directions for future work.

6.1 Research conclusions

Our test of the Cisco firewall suggests that there is merit in engineering networks for loss-free paths to serve the narrow use-cases addressed by science DMZs (single or small numbers of high bandwidth-delay product connections between research infrastructures). Due to the limited design of the experiments we performed, we cannot generalise this conclusion to all traditional firewalls. We speculate that *some* firewalls are, or with advances in technology will be, capable of stateful packet filtering of single connections at very high speeds, without introducing packet loss.

We conclude that, as an interim measure or an alternative to static separation of end-points into classes protected by stateful and stateless (ACL) packet filters, it is possible to build a hybrid firewall that off-loads trusted connections to stateless hardware switching. Our prototype demonstrates the feasibility of an Software Defined Networking (SDN)-based design, making use of widely used Free and Open Source Software (FOSS) for the stateful slow-path, and vendor-agnostic OpenFlow forwarding for the fast-path. Our experiments verify that the performance of dynamically off-loaded Transmission Control Protocol (TCP) connections benefit from science DMZ-like loss-free forwarding. These results address the research problem, and answer the research questions one and two (posed in 3.1).

Analysing the capital costs of implementing our prototype shows potential for substantial savings over the cost of traditional firewalls with similar performance charac-

teristics (by our estimate, at least an order of magnitude less). The operational benefits of composing OpenFlow switching with Linux's Netfilter are difficult to quantify and may be outweighed by increased complexity. Due to network latency, the science use-case requires loss-free forwarding in countries that are geographically distant from collaborators in Europe and North America. This work is particularly important due to cost-constraints where those countries are also developing or newly industrialised nations. This answers research question three and, to the extent possible within the scope of the study, it suggests possible answers to question four.

Finally, the parallel-flow performance results via a non-ideal network path suggests that improving TCP, or adopting better file transfer tools, should be seriously considered as an alternative to building loss-free networks. Our experimental design did not produce data that allowed us to evaluate and compare firewall performance with non-TCP connections.

6.2 Future work

Considering the problem of Science DMZ security, two avenues for future research are immediately evident:

- Our focus on an SDN-based hybrid firewall implementation makes some assumptions about the information security threat model and application design. A study focused on data-intensive research network use-cases that examines real-world applications, and considers a comprehensive threat model, is called for.

- An investigation into the real-world performance of SDN-enhanced science DMZs (especially operations and maintenance aspects) would be valuable to operators of High Performance Computing (HPC) infrastructure. IDS-driven alternatives such as SciPass offer unique advantages over NFShunt's design, and we hope to see both systems deployed and evaluated in production networks.

The introduction of a Linux driver framework for switch-like devices (switchdev [67]) provides a mechanism to manipulate hardware off-loaded forwarding using standard Linux IP routing and Ethernet bridging utilities. While this functionality appears to be intended for tight integration with local hardware, it may be possible to extend switchdev to support remote forwarding planes via OpenFlow. If Netfilter offloading could then be added to switchdev, it might be possible to re-implement NFShunt in the Linux kernel. Alternatively, an NFShunt-specific Netfilter target would improve the integration with the Linux kernel and allow for the user-interface enhancement suggested in section 5.1.

Similarly, integration between open vSwitch (OVS) and Netfilter connection tracking [65] (expected to be available at the end of 2015) could enable hardware acceleration directly in the OpenFlow agent.

More generally, future work could explore using hybrid SDN designs in different applications, for example:

- Adding a shunting mechanism to routing (as opposed to transparent) firewalls may be useful in certain network designs. Initial work on chaining NFShunt into the forwarding path of larger SDN-enabled systems such as Vandervecken [71] (a fork of RouteFlow [59]) shows promise.

- The use of NFShunt's *block action* (which allows the slow path firewall rules to off-load packets to be dropped to hardware) was briefly tested, but not explored in our research. This function could form part of a denial-of-service mitigation system with applications beyond the science DMZ use-case.

A new field, related to SDN is Network Function Virtualisation (NFV). This is an architecture that replaces dedicated hardware appliances performing fixed network functions (such as firewalls, proxies, etc.) with virtualised servers performing the same functions. Since NFV is based on software implementations running on normal CPUs, it would be subject to similar scalability constraints to traditional firewalls. As with other cloud-based computing architectures, NFV typically relies on a scalability strategy of parallel processing (*scaling-out*). NFShunt could be enhanced to provide a standard interface for a virtualised slow path to off-load forwarding of specific connections to hardware. An implementation of hybrid-NFV such as this could provide a mechanism to *scale-up* the performance of virtual network functions.

Finally, some cutting-edge, SDN-enabled hardware platforms support the tracking of transport layer states in the forwarding plane. Open standards to take advantage of this capability could enhance NFShunt-like designs, if not eliminate the need for state-less packet filtering (and hence science DMZs) entirely.

Appendices

Appendix A

openVSwitch usage

Mininet [43] is a network prototyping tool that uses OS-level virtualisation to allow the creation of logically independent host, router and switch nodes, interconnected by virtual network links.

To demonstrate the use of open vSwitch (OVS) commands, we show the simplest Mininet example (the *minimal topology* illustrated in figure A.1) by running `sudo mn -x`. This creates four terminal windows, one for each node.

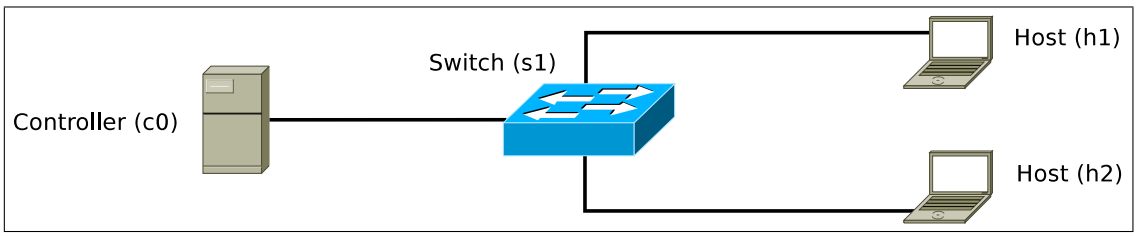

Figure A.1: Mininet's minimal topology

Accessing the switch, we can query the OVS configuration database:

```
switch s1 # ovs-vsctl show
0b8ed0aa-67ac-4405-af13-70249a7e8a96
    Bridge "s1"
        Controller "tcp:127.0.0.1:6633"
            is_connected: true
        Controller "ptcp:6634"
        fail_mode: secure
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1-eth2"
            Interface "s1-eth2"
        Port "s1"
            Interface "s1"
                type: internal
    ovs_version: "2.0.2"
```

In addition to the two ports connecting host h1 and h2, each OVS bridge (switch instance) has an internal port with the same name as the bridge. This port is logically connected to the host IP stack and can be used by the host system to communicate via the bridge.

Since ovs-vswitchd applies this configuration to the data-path, we can obtain similar information by querying the data-path directly using `ovs-dpctl`:

```
switch s1 # ovs-dpctl show
system@ovs-system:
        lookups: hit:3 missed:21 lost:0
        flows: 0
        port 0: ovs-system (internal)
        port 1: s1-eth1
        port 2: s1-eth2
        port 3: s1 (internal)
```

While OVS data-paths require the features to implement OpenFlow forwarding behaviour, OVS abstracts away all the OpenFlow-specific semantics. Note that information regarding the OpenFlow controller and controller-failure mode are absent from the `ovs-dpctl` output.

With `ovs-ofctl`, we can use the OpenFlow protocol to query the capability of the switch:

```
switch s1 # ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST
    SET_NW_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST ENQUEUE
 1(s1-eth1): addr:4a:15:3a:ad:7f:98
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 2(s1-eth2): addr:ba:0f:c6:93:44:f5
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 LOCAL(s1): addr:3a:ce:8d:ef:b6:48
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

Notable in the output is support for multiple OpenFlow tables (`n_tables:254`), as well as the switch capabilities and supported actions.

Querying the switch's OpenFlow table at this point shows that no flow entries have been programmed:

```
switch s1 # ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
```

The *minimal topology* Mininet example configures a reference OpenFlow controller included in OVS for the switch to connect to. This implements a basic layer-2 Media Access Control (MAC)-address learning Ethernet switch that maintains state in the controller and reactively populates the switch with micro-flows.[1] If we generate traffic between h1 and h2 by executing ping, and then query the table again, we can see flow entries added for Address Resolution Protocol (ARP) and Internet Control Message Protocol (ICMP) between the two switch ports:

[1] flow specifications that map to individual transport layer connections (by specifying a 5-tuple of matches.

```
host h1 # ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=5.51 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.623 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.050 ms
...output truncated.

switch s1 # ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=307.399s, table=0, n_packets=9, n_bytes=378,
  idle_timeout=60, idle_age=15, priority=65535,arp,in_port=2,
  vlan_tci=0x0000,dl_src=96:0f:63:93:27:73,dl_dst=06:49:6b:f9:de:08,
  arp_spa=10.0.0.2,arp_tpa=10.0.0.1,arp_op=1 actions=output:1
 cookie=0x0, duration=307.398s, table=0, n_packets=9, n_bytes=378,
  idle_timeout=60, idle_age=15, priority=65535,arp,in_port=1,
  vlan_tci=0x0000,dl_src=06:49:6b:f9:de:08,dl_dst=96:0f:63:93:27:73,
  arp_spa=10.0.0.1,arp_tpa=10.0.0.2,arp_op=2 actions=output:2
 cookie=0x0, duration=312.4s, table=0, n_packets=313, n_bytes=30674,
  idle_timeout=60, idle_age=0, priority=65535,icmp,in_port=2,
  vlan_tci=0x0000,dl_src=96:0f:63:93:27:73,dl_dst=06:49:6b:f9:de:08,
  nw_src=10.0.0.2,nw_dst=10.0.0.1,nw_tos=0,icmp_type=0,icmp_code=0
  actions=output:1
 cookie=0x0, duration=311.401s, table=0, n_packets=312, n_bytes=30576,
  idle_timeout=60, idle_age=0, priority=65535,icmp,in_port=1,
  vlan_tci=0x0000,dl_src=06:49:6b:f9:de:08,dl_dst=96:0f:63:93:27:73,
  nw_src=10.0.0.1,nw_dst=10.0.0.2,nw_tos=0,icmp_type=8,icmp_code=0
  actions=output:2
```

In the above listing we have underlined the matches, while the action for each flow specification is boldface. The idle age (`idle_age`) of the first two flows that match on the ARP packets shows that the ping command had been running for approximately 15 seconds. All the flows are configured with an idle timeout (`idle_timeout`) of sixty seconds. The idle ages of the two ICMP flow specifications are zero because they had been reset by an ICMP echo-reply pair within the same second that the `ovs-ofctl` command queried the switch. Stopping the ping command and waiting sixty seconds would cause all flow specifications to time out, and would leave the flow table empty.

OVS is pre-configured by Mininet according to the given topology. For the purpose of our research, manually creating an OVS bridge, adding ports to it and configuring a controller for the bridge are the most important uses of the OVS command line utilities. We use the `ovs-vsctl` command for all three tasks. In the example below, we create a new bridge named *s2*, and add the port named *eth0* to this bridge:

```
# ovs-vsctl add-br s2
# ovs-vsctl add-port s2 eth0
# ovs-vsctl set-controller s2 tcp:192.0.2.1:6633
```

These commands accept additional parameters that may be required with non-Linux kernel data-path implementations and to specify extra port properties. Finally, our experiments require access to port frame counters to detect and locate packet loss. We also obtain these counters using a `ovs-vsctl` command:

```
# ovs-vsctl get Interface eth0 statistics
{collisions=0, rx_bytes=150, rx_crc_err=0, rx_dropped=0, rx_errors=0,
    rx_frame_err=0, rx_over_err=0, rx_packets=2, tx_bytes=0, tx_dropped=0,
    tx_errors=0, tx_packets=0}
```

Appendix B

Source code listing

Listing B.1: NFShunt POX controller

```
# Copyright 2014 CSIR
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
import sys
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpidToStr
import pox.lib.packet as pkt # POX convention
from threading import Thread
from xml.etree import ElementTree
from io import BytesIO
from subprocess import Popen, PIPE
import json
import re

log = core.getLogger()

class NFShunt(object):
        def __init__(self, configfilename):
                self.connection = None
                self.config = None
                self.read_config(configfilename)
                core.openflow.addListeners(self)
                #core.addListenerByName("DownEvent", self._handle_DownEvent) # WHY?!
                core.addListenerByName("GoingDownEvent", self._handle_GoingDownEvent) # WHY?!
                log.info("Launch complete, waiting for OF connection...")

        def read_config(self, configfilename):
                text = open(configfilename).read()
                try:
                        self.config = json.loads(text)
                        self.config['port_slow'] = {}
```

```
45                          self.config['port_fast'] = {}
46                          self.config['port_physdevin'] = {}
47                          for port in self.config['ports']:
48                              self.config['port_slow'][port['slow']] = port
49                              self.config['port_fast'][port['fast']] = port
50                              self.config['port_physdevin'][port['physdevin']] = port
51                  except Exception as e:
52                          raise type(e), type(e)(e.message + '_happens_with_[%s]_' % text), sys.
                                exc_info()[2]
53
54          def conntrack_read_events(self, stdout, dummy):
55                  for line in iter(stdout.readline, b''):
56                          if line.find("flow")!=-1: # conntrack sometimes output non-XML lines
57                                  try:
58                                          etree = ElementTree.parse(BytesIO(line))
59                                          ev = next(etree.iter())
60                                  except:
61                                          log.error('Failed_to_parse_event_data:_{}'.format(
                                                ev_xml))
62                                          continue
63                                  if ev is None:
64                                          continue
65                                  self.try_shunting(ev)
66
67          def try_shunting(self, flow):
68                  try:
69                          if not flow.findall(".//mark"): return
70                          eventtype = None
71                          if 'type' in flow.attrib:
72                                  eventtype = flow.attrib['type']
73                          if eventtype in ["destroy"]: return
74                          mark = int(flow.find('.//meta[@direction="independent"]/mark').text)
75                          connid = int(flow.find('.//meta[@direction="independent"]/id').text)
76                          timeout = None
77                          timeouttag = flow.find('.//meta[@direction="independent"]/timeout')
78                          if timeouttag is not None: timeouttag.text
79                          state = None
80                          statetag = flow.find('.//meta[@direction="independent"]/state')
81                          if statetag is not None: state = statetag.text
82                          client_ip_tag = flow.find('.//meta[@direction="original"]/layer3/src')
83                          server_ip_tag = flow.find('.//meta[@direction="original"]/layer3/dst')
84                          client_port_tag = flow.find('.//meta[@direction="original"]/layer4/
                                sport')
85                          server_port_tag = flow.find('.//meta[@direction="original"]/layer4/
                                dport')
86                          if client_ip_tag is None or server_ip_tag is None or client_port_tag is
                                 None or server_port_tag is None:
87                                  log.debug("Flow_doesn't_have_all_L3_and_L4_info_we_need,_
                                        ignoring.")
88                                  return
89                          client_ip = client_ip_tag.text
90                          server_ip = server_ip_tag.text
91                          client_port = int(client_port_tag.text)
92                          server_port = int(server_port_tag.text)
93                  except Exception as e:
94                          raise type(e), type(e)(e.message + '_happens_with_[%s]_' % ElementTree.
                                tostring(flow)), sys.exc_info()[2]
95                  flags = mark >> 28
96                  flags_physdevin = (flags & 0x4) >> 2
97                  flags_physdevout = (flags & 0x2) >> 1
98                  if not (flags_physdevin and flags_physdevout):
99                          log.debug("Flow_is_probably_not_via_one_of_the_slow_path_ports,_
                                ignoring.")
100                         return
101                 flags_flowmark = flags & 0x1
102                 physdevin = (mark & 0x0f000000) >> 24
103                 physdevout = (mark & 0x00f00000) >> 20
104                 flowmark = (mark & 0x000f0000) >> 16
105                 of_ports_in = self.config['port_physdevin'][physdevin]['fast']
106                 of_ports_out = self.config['port_physdevin'][physdevout]['fast']
```

```
107                 of_ports = [of_ports_in, of_ports_out]
108                 log.info("Conntrack event: type=%s mark=%s [flags=(pdin=%s,pdout=%s,flow=%s)] pdin=%s, pdout=%s, flowmark=%s, connid=%s, timeout=%s, state=%s, client=%s:%s, server=%s:%s" %
109                     tuple(map(str, [eventtype, hex(mark), flags_physdevin, flags_physdevout, flags_flowmark, physdevin, physdevout, flowmark,
110                         connid, timeout, state, client_ip, client_port, server_ip, server_port])))
111                 if flags_flowmark:
112                     if state in ["FIN_WAIT", "LAST_ACK", "TIME_WAIT"]:
113                         log.info("Not installing shunt because connection state is %s" % state)
114                     else:
115                         action = self.config['mark_actions'][str(flowmark)]
116                         log.info("User policy flowmark of %d detected in conntrack entry, action is: %s" % (flowmark, action))
117                         if action == "ignore":
118                             log.info("Doing nothing, because user policy asked us to ignore this flow.")
119                             # if default_no_shunting=true, this is equivalent to forcing via the slow path
120                         else:
121                             if action == "shunt":
122                                 # For shunting we add flows to match, which send packets via fast path
123                                 self.connection.send(of.ofp_flow_mod(action=of.ofp_action_output(port=of_ports[1]), # if packet came from of_ports[0], send to of_ports[1]
124                                     match=of.ofp_match(in_port=of_ports[0], dl_type=0x800, nw_dst=server_ip, nw_src=client_ip,
125                                     nw_proto=pkt.ipv4.TCP_PROTOCOL, tp_src=client_port, tp_dst=server_port),
126                                     priority=33000, idle_timeout=self.config['default_shunt_timeout'],
127                                     flags=of.OFPFF_SEND_FLOW_REM, cookie=connid))
128                                 self.connection.send(of.ofp_flow_mod(action=of.ofp_action_output(port=of_ports[0]), # if packet came from of_ports[1], send to of_ports[0]
129                                     match=of.ofp_match(in_port=of_ports[1], dl_type=0x800, nw_dst=client_ip, nw_src=server_ip,
130                                     nw_proto=pkt.ipv4.TCP_PROTOCOL, tp_src=server_port, tp_dst=client_port),
131                                     priority=33000, idle_timeout=self.config['default_shunt_timeout'],
132                                     flags=of.OFPFF_SEND_FLOW_REM, cookie=connid))
133                                 log.info("Shunt installed for server %s:%d [via port %d] -> client %s:%d [via port %d] - conntrack id %d"
134                                     % (server_ip, server_port, of_ports[1], client_ip, client_port, of_ports[0], connid))
135                                 log.info("Shunt installed for client %s:%d [via port %d] -> server %s:%d [via port %d] - conntrack id %d"
136                                     % (client_ip, client_port, of_ports[0], server_ip, server_port, of_ports[1], connid))
137                                 self.connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
138                             elif action == "block":
139                                 # For blocking we add flows to match, which send packets to dev null
140                                 self.connection.send(of.ofp_flow_mod(action=[], # empty action list == drop
```

```
141                                                     match=of.ofp_match(in_port=of_ports[0],
                                                            dl_type=0x800,nw_dst=server_ip,
                                                            nw_src=client_ip,
142                                                     nw_proto=pkt.ipv4.TCP_PROTOCOL,tp_src=
                                                            client_port,tp_dst=server_port),
143                                                     priority=33000,idle_timeout=self.config
                                                            ['default_block_timeout'],
144                                                     flags=of.OFPFF_SEND_FLOW_REM,cookie=
                                                            connid))
145                                             self.connection.send(of.ofp_flow_mod(action=[],
                                                        # empty action list == drop
146                                                     match=of.ofp_match(in_port=of_ports[1],
                                                            dl_type=0x800,nw_dst=client_ip,
                                                            nw_src=server_ip,
147                                                     nw_proto=pkt.ipv4.TCP_PROTOCOL,tp_src=
                                                            server_port,tp_dst=client_port),
148                                                     priority=33000,idle_timeout=self.config
                                                            ['default_block_timeout'],
149                                                     flags=of.OFPFF_SEND_FLOW_REM,cookie=
                                                            connid))
150                                             log.info("Block installed for server %s:%d [via
                                                        port %d] -> client %s:%d [via port %d] -
                                                        conntrack id %d"
151                                                         % (server_ip, server_port, of_ports[1],
                                                                client_ip, client_port, of_ports
                                                                [0], connid))
152                                             log.info("Block installed for client %s:%d [via
                                                        port %d] -> server %s:%d [via port %d] -
                                                        conntrack id %d"
153                                                         % (client_ip, client_port, of_ports[0],
                                                                server_ip, server_port, of_ports
                                                                [1], connid))
154                                             self.connection.send(of.ofp_stats_request(body=
                                                        of.ofp_flow_stats_request()))
155                                     # Now that we've installed flows, we must nuke the
                                                conntrack entry
156                                     self.delete_conntrack(connid, client_ip, client_port,
                                                server_ip, server_port)
157
158
159         def delete_conntrack(self, connid, client_ip, client_port, server_ip, server_port):
160                 log.info("Running command to delete conntrack entry %d" % connid)
161                 os.system("conntrack -D -p tcp -s %s --sport %d -d %s --dport %d" % (client_ip,
                        client_port, server_ip, server_port))
162                 log.info("Done deleting.")
163
164         def _handle_ConnectionUp(self, event):
165                 log.info("Switch %s is up.", dpidToStr(event.dpid))
166                 self.connection = event.connection
167                 if self.config['delete_flows_on_startup'] is True:
168                         log.info("Deleting existing flow entries.")
169                         self.connection.send(of.ofp_flow_mod(command=of.OFPFC_DELETE))
170                 if self.config['default_no_shunting'] is True:
171                         log.info("Adding flow entries for default slow-path switching.")
172                         for portgroup in self.config['ports']:
173                                 self.connection.send(of.ofp_flow_mod(action=of.
                                        ofp_action_output(port=portgroup['fast']),match=of.
                                        ofp_match(in_port=portgroup['slow']))))
174                                 self.connection.send(of.ofp_flow_mod(action=of.
                                        ofp_action_output(port=portgroup['slow']),match=of.
                                        ofp_match(in_port=portgroup['fast']))))
175                 log.info("Done with setup, now ready for conntrack events...")
176                 self.connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
177                 log.info("Controller running, checking for existing conntrack objects...")
178                 conntrack_existing = Popen(['conntrack','-L','-o','xml,id'], stdout=PIPE,
                        bufsize=1)
179                 self.conntrack_read_events(conntrack_existing.stdout, True)
180                 log.info("Done checking existing objects, starting new conntrack events process
                        ...")
181                 conntrack_process = Popen(['conntrack','-E','-o','xml,id'], stdout=PIPE,
```

```
                bufsize=1)
182             conntrack_thread = Thread(target=self.conntrack_read_events, args=(
                    conntrack_process.stdout, True))
183             conntrack_thread.daemon = True
184             log.info("Starting conntrack event consumer thread...")
185             conntrack_thread.start()
186
187     def _handle_ConnectionDown(self, event):
188             log.info("Switch %s is down.", dpidToStr(event.dpid))
189             self.connection = None
190
191     def _handle_FlowRemoved(self, event):
192             log.info("Switch removed flow: reason=%d cookie=%d duration=%d/%d bytes=%d
                    packets=%d" %
193                     (event.ofp.reason, event.ofp.cookie, event.ofp.duration_sec, event.ofp.
                            duration_nsec, event.ofp.byte_count, event.ofp.packet_count))
194             self.connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
195
196     def _handle_FlowStatsReceived(self, event):
197             log.debug("Flow stats follow:")
198             for stat in event.stats:
199                     log.debug(self.format_stats(stat))
200
201     def format_stats(self, stat):
202             def safehex(n):
203                     if n is None:
204                             return "(None)"
205                     else:
206                             return hex(n)
207             def append (obj, f, formatter=str, prefix=' '):
208                     try:
209                             v = getattr(obj, f)
210                             if v is None: return ''
211                             return prefix + f + "=" + formatter(v)
212                     except AttributeError:
213                             return ''
214             outstr = 'match:['
215             outstr += append(stat.match,'in_port', prefix='')
216             outstr += append(stat.match,'dl_src')
217             outstr += append(stat.match,'dl_dst')
218             outstr += append(stat.match,'dl_vlan')
219             outstr += append(stat.match,'dl_vlan_pcp')
220             outstr += append(stat.match,'dl_type', safehex)
221             outstr += append(stat.match,'nw_tos')
222             outstr += append(stat.match,'nw_proto')
223             outstr += append(stat.match,'nw_src')
224             outstr += append(stat.match,'nw_dst')
225             outstr += append(stat.match,'tp_src')
226             outstr += append(stat.match,'tp_dst')
227             outstr += '] actions:['
228             first = True
229             for action in stat.actions:
230                     if first:
231                             outstr += '['
232                             first = False
233                     else:
234                             outstr += ' ['
235                     outstr += append(action, 'type', prefix='')
236                     outstr += append(action, 'port')
237                     outstr += append(action, 'queue_id')
238                     outstr += append(action, 'vlan_vid')
239                     outstr += append(action, 'vlan_pcp')
240                     outstr += append(action, 'dl_addr')
241                     outstr += append(action, 'nw_addr')
242                     outstr += append(action, 'nw_tos')
243                     outstr += append(action, 'tp_port')
244                     outstr += append(action, 'vendor')
245                     outstr += ']'
246             outstr += ']'
247             outstr += ' duration_sec=' + str(stat.duration_sec)
```

```
248                 outstr += ' duration_nsec=' + str(stat.duration_nsec)
249                 outstr += ' priority=' + str(stat.priority)
250                 outstr += ' idle_timeout=' + str(stat.idle_timeout)
251                 outstr += ' hard_timeout=' + str(stat.hard_timeout)
252                 outstr += ' cookie=' + str(stat.cookie)
253                 outstr += ' packet_count=' + str(stat.packet_count)
254                 outstr += ' byte_count=' + str(stat.byte_count)
255                 return outstr
256
257         def _handle_DownEvent(self, event):
258                 log.debug("Running Down event")
259
260         def _handle_GoingDownEvent(self, event):
261                 log.debug("Running GoingDown event")
262                 if self.config['delete_flows_on_shutdown'] is True:
263                         log.info("Deleting flows before shutting down")
264                         self.connection.send(of.ofp_flow_mod(command=of.OFPFC_DELETE))
265
266 def launch (configfilename="nfshunt.json"):
267         core.register("nfshunt", NFShunt(configfilename))
```

Listing B.2: Sample configuration file

```
1  {
2          "delete_flows_on_startup": true,
3          "default_shunt_timeout":10,
4          "default_block_timeout":10,
5          "default_no_shunting":true,
6          "ports":[
7                          {
8                                  "fast":49,
9                                  "slow":10,
10                                 "physdevin":1
11                         },
12                         {
13                                 "fast":50,
14                                 "slow":11,
15                                 "physdevin":2
16                         }
17         ],
18         "mark_actions":{
19                         "0":"ignore",
20                         "1":"shunt",
21                         "2":"block"
22         }
23 }
```

Listing B.3: Sample iptables configuration script

```
1  #!/bin/bash
2  # Copyright 2014 CSIR
3  #
4  # Licensed under the Apache License, Version 2.0 (the "License");
5  # you may not use this file except in compliance with the License.
6  # You may obtain a copy of the License at
7  #
8  # http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 echo "Creating chains"
17 iptables -t mangle -N NFSHUNT_PRE
18 iptables -t mangle -N NFSHUNT_POST
```

```
19 iptables -t mangle -N NFSHUNT_POLICY
20 iptables -t mangle -N NFSHUNT_PRE_PD_IN
21 iptables -t mangle -N NFSHUNT_POST_PD_OUT
22
23 echo "Adding test user mark to FORWARD"
24 iptables -t filter -A FORWARD -j MARK --set-xmark 0x1234/0xffff
25
26 echo "Populating NFSHUNT_PRE_PD_IN"
27 iptables -t mangle -A NFSHUNT_PRE_PD_IN -m physdev --physdev-in p1p1 -j MARK --set-xmark 0
       x41000000/0x4f000000
28 iptables -t mangle -A NFSHUNT_PRE_PD_IN -m physdev --physdev-in p1p2 -j MARK --set-xmark 0
       x42000000/0x4f000000
29 iptables -t mangle -A NFSHUNT_PRE_PD_IN -j RETURN
30
31 echo "Populating NFSHUNT_PRE"
32 iptables -t mangle -A NFSHUNT_PRE -m physdev --physdev-in '!p1+' -j RETURN # ignore packets not
        coming from the interfaces on the slow path
33 iptables -t mangle -A NFSHUNT_PRE -j CONNMARK --restore-mark # copy mark from connection state
       to packet
34 iptables -t mangle -A NFSHUNT_PRE -m mark ! --mark 0x40000000/0x40000000 -j NFSHUNT_PRE_PD_IN #
        if physdev_in is not marked, send to chain where we do this
35 iptables -t mangle -A NFSHUNT_PRE -j RETURN
36
37 echo "Populating NFSHUNT_POST_PD_OUT"
38 iptables -t mangle -A NFSHUNT_POST_PD_OUT -m physdev --physdev-out p1p1 -j MARK --set-xmark 0
       x20100000/0x20f00000
39 iptables -t mangle -A NFSHUNT_POST_PD_OUT -m physdev --physdev-out p1p2 -j MARK --set-xmark 0
       x20200000/0x20f00000
40 iptables -t mangle -A NFSHUNT_POST_PD_OUT -j RETURN
41
42 echo "Populating NFSHUNT_POLICY"
43 iptables -t mangle -A NFSHUNT_POLICY -p tcp --dport 4999 -m conntrack --ctstate RELATED,
       ESTABLISHED -j MARK --set-xmark 0x10000000/0x100f0000 # ignore
44 iptables -t mangle -A NFSHUNT_POLICY -p tcp --dport 5000 -m conntrack --ctstate RELATED,
       ESTABLISHED -j MARK --set-xmark 0x10010000/0x100f0000 # shunt
45 iptables -t mangle -A NFSHUNT_POLICY -p tcp --dport 5001 -m conntrack --ctstate RELATED,
       ESTABLISHED -j MARK --set-xmark 0x10010000/0x100f0000 # shunt
46 iptables -t mangle -A NFSHUNT_POLICY -p tcp --dport 5666 -m conntrack --ctstate RELATED,
       ESTABLISHED -j MARK --set-xmark 0x10020000/0x100f0000 # block
47 iptables -t mangle -A NFSHUNT_POLICY -j RETURN
48
49 echo "Populating NFSHUNT_POST"
50 iptables -t mangle -A NFSHUNT_POST -m physdev ! --physdev-is-bridged -j RETURN # don't bother
        with non-bridged packets
51 iptables -t mangle -A NFSHUNT_POST -m mark ! --mark 0x40000000/0x40000000 -j RETURN # if we
        didn't mark physdev_in, then it's another bridge
52 iptables -t mangle -A NFSHUNT_POST -m mark ! --mark 0x20000000/0x20000000 -j
       NFSHUNT_POST_PD_OUT # if physdev_out is not marked, send to chain where we do this
53 iptables -t mangle -A NFSHUNT_POST -m mark ! --mark 0x10000000/0x10000000 -j NFSHUNT_POLICY #
        if flow flag is not set, we need to jump to the shunt policy table
54 iptables -t mangle -A NFSHUNT_POST -j CONNMARK --save-mark
55 iptables -t mangle -A NFSHUNT_POST -j RETURN
56
57 echo "Adding rule to PREROUTING to go to NFSHUNT_PRE"
58 iptables -t mangle -A PREROUTING -j NFSHUNT_PRE
59 echo "Adding rule to POSTROUTING to go to NFSHUNT_POST"
60 iptables -t mangle -A POSTROUTING -j NFSHUNT_POST
```

Bibliography

[1] K. Accardi, T. Bock, F. Hady, and J. Krueger. Network processor acceleration for a linux* netfilter firewall. In *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 115–123. ACM, 2005.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke. GridFTP: Protocol extensions to FTP for the Grid. *Global Grid ForumGFD-RP*, 20:1–21, 2003.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 54. IEEE Computer Society, 2005.

[4] J. Aweya. IP router architectures: an overview. In *Journal of Systems Architecture*, 1999.

[5] E. Balas and A. Ragusa. Scipass: a 100gbps capable secure science dmz using openflow and bro. In *Supercomputing 2014 conference (SC14)*, 2014.

[6] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 151–160. ACM, 1998.

[7] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[8] E.-J. Bos, E. Martelli, P. Moroni, and D. Foster. LHC tier-0 to tier-1 high-level network architecture. Technical report, CERN, Tech. Rep, 2005.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent

packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[10] S. Bradner. Benchmarking terminology for network interconnection devices. RFC 1242, RFC Editor, 1991.

[11] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, RFC Editor, 1999.

[12] P. Calyam, A. Berryman, E. Saule, H. Subramoni, P. Schopis, G. Springer, U. Catalyurek, and D. K. Panda. Wide-area overlay networking to manage science DMZ accelerated flows. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 269–275. IEEE, 2014.

[13] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *HotNets*, pages 1–6. Citeseer, 2008.

[14] CERN, the European Organization for Nuclear Research. Large hadron collider. http://lhc.web.cern.ch/lhc/. Accessed: 26/02/2013.

[15] M.-S. Chen, M.-Y. Liao, P.-W. Tsai, M.-Y. Luo, C.-S. Yang, and C. E. Yeh. Using NetFPGA to offload Linux Netfilter firewall. In *2nd North American NetFPGA Developers Workshop*, 2010.

[16] J. Collings and J. Liu. An OpenFlow-Based Prototype of SDN-Oriented Stateful Hardware Firewalls. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 525–528. IEEE, 2014.

[17] E. Dart. The Science DMZ. https://fasterdata.es.net/assets/Uploads/20130717-dart-sciencedmz.pdf. Accessed: 6/10/2015.

[18] E. Dart. Science DMZ security. In *Joint Techs, Winter 2013, Honolulu, Hawaii*, 2013.

[19] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski. The Science DMZ: A Network Design Pattern for Data-intensive Science. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 85:1–85:10, New York, NY, USA, 2013. ACM.

[20] Data plane development kit. http://dpdk.org. Accessed: 29/07/2015.

[21] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. Lazio. The square kilometre array. *Proceedings of the IEEE*, 97(8):1482–1496, 2009.

[22] Energy Sciences Network. Fasterdata: Firewall architecture. https://fasterdata.es.net/network-tuning/firewall-performance-issues/firewall-architecture-exercise/. Accessed: 13/08/2015.

[23] Energy Sciences Network. Fasterdata: Firewall performance issues. https://fasterdata.es.net/network-tuning/firewall-performance-issues/. Accessed: 13/08/2015.

[24] Energy Sciences Network. Iperf3. http://software.es.net/iperf/. Accessed: 11/03/2015.

[25] Energy Sciences Network. Science DMZ Security - Firewalls vs. Router ACLs. http://fasterdata.es.net/science-dmz/science-dmz-security/, 2013. Accessed: 19/11/2012.

[26] J. Engelhardt. Relation of the different Netfilter components to another. http://inai.de/images/nf-components.svg, 2008.

[27] J. Engelhardt. Packet flow in Netfilter and General Networking. http://inai.de/images/nf-packet-flow.svg, 2011.

[28] W.-c. Feng, A. Goel, A. Bezzaz, W.-c. Feng, and J. Walpole. TCPivo: a high-performance packet replay engine. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 57–64. ACM, 2003.

[29] S. Floyd. HighSpeed TCP for large congestion windows. RFC 3649, RFC Editor, 2003.

[30] O. N. Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[31] J. M. Gonzalez, V. Paxson, and N. Weaver. Shunting: a hardware/software architecture for flexible, high-performance network intrusion prevention. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 139–149. ACM, 2007.

[32] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41(4):30–32, 2008.

[33] T. Halfhill. Intel network processor targets routers. *Microprocessor Report*, 13(12):1, 1999.

[34] B. Harris and R. Hunt. TCP/IP security threats and attack methods. *Computer Communications*, 22(10):885–897, 1999.

[35] S. Hemminger et al. Network emulation with NetEm. In *LinuxConf AU*, pages 18–23, 2005.

[36] A. Hey, S. Tansley, and K. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.

[37] B. Hickman, D. Newman, S. Tadjudin, and T. Martin. Benchmarking methodology for firewall performance. RFC 3511, RFC Editor, 2003.

[38] ITU-T. Recommendation m.3400, tmn management functions, 2000.

[39] W. Johnston and R. McCool. The Square Kilometer Array, A next generation scientific instrument and its implications for networks. *TERENA Networking Conference*, 2012.

[40] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[41] I. Kennedy. Lost call theory. *Lecture Notes, ELEN5007–Teletraffic Engineering, School of Electrical and Information Engineering, University of the Witwatersrand*, 2005.

[42] M. Kühlewind, S. Neuner, and B. Trammell. On the state of ECN and TCP options on the internet. In *Passive and active measurement*, pages 135–144. Springer, 2013.

[43] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[44] W. Liu, R. Kettimuthu, B. Tieman, R. Madduri, B. Li, and I. Foster. Gridftp gui: an easy and efficient way to transfer data in grid. http://www.mcs.anl.gov/~kettimut/talks/gridnets09.pdf. Accessed: 6/10/2015.

[45] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 160–161. IEEE, 2007.

[46] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI*, volume 11, pages 2–2, 2011.

[47] G. Lu, R. Miao, Y. Xiong, and C. Guo. Using CPU as a traffic co-processing unit in commodity switches. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 31–36. ACM, 2012.

[48] M. Mathis, J. Heffner, and R. Reddy. Web100: extended TCP instrumentation for research, education and diagnosis. *ACM SIGCOMM Computer Communication Review*, 33(3):69–79, 2003.

[49] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[50] N. McKeown. How to tell your plumbing what to do: Protocol Independent Forwarding. http://yuba.stanford.edu/~nickm/talks/ONF%20Talk%20Sept%2014%20v3.pptx. Accessed: 02/09/2015.

[51] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[52] P. McMahon and A. Hutchison. A security architecture for high performance computing facilities. In *Information Security South Africa, Balalaika Hotel, Sandton, South Africa*, 2006.

[53] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware data plane processing in SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2014.

[54] T. Meyer, F. Wohlfart, D. Raumer, B. E. Wolfinger, and G. Carle. Measurement and simulation of high-performance packet processing in software routers. *Leistungs-, Zuverlassigkeits-und Verlasslichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, 7, 2013.

[55] S. Miteff and S. HazelHurst. NFShunt: a Linux firewall with OpenFlow-enabled hardware bypass. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN) (NFV-SDN'15)*, pages 102–108, San Francisco, USA, Nov. 2015.

[56] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 61–66. ACM, 2014.

[57] R. Narayanan, S. Kotha, G. Lin, A. Khan, S. Rizvi, W. Javed, H. Khan, and S. A. Khayam. Macroflows and microflows: Enabling rapid network innovation through a split SDN data plane. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 79–84. IEEE, 2012.

[58] R. R. Narisetty. How long does it take to offload traffic from firewall? Master's thesis, University of Houston, 2013.

[59] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. Corrêa, S. C. de Lucena, and M. F. Magalhães. Virtual routers as a service: the routeflow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, pages 34–37. ACM, 2011.

[60] D. Newman. Benchmarking terminology for firewall performance. RFC 2647, RFC Editor, 1999.

[61] P. Newman, G. Minshall, T. Lyon, and L. Huston. IP switching and gigabit routers. *IEEE Communications magazine*, 35(1):64–69, 1997.

[62] NSS Labs Inc. TEST METHODOLOGY: Next Generation Firewall (NGFW). https://www.nsslabs.com/sites/default/files/public-report/files/Next%20Generation%20Firewall%20Test%20Methodology%20v5_4.pdf. Accessed: 02/09/2015.

[63] A. Nygren, B. Pfaff, B. Lantz, et al. OpenFlow Switch Specification Version 1.5.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, 2014.

[64] A. Ossipov. Maximising Firewall Performance. http://www.alcatron.net/Cisco%20Live%202013%20Melbourne/Cisco%20Live%

20Content/Security/BRKSEC-3021%20%20Maximising%20Firewall%20Performance.pdf. Accessed: 27/11/2013.

[65] J. Pettit. Open vSwitch and the Intelligent Edge. http://openvswitch.org/slides/OpenStack-140513.pdf. Accessed: 5/10/2015.

[66] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[67] J. Pirko and S. Feldman. Ethernet switch device driver model (switchdev). https://www.kernel.org/doc/Documentation/networking/switchdev.txt. Accessed: 5/10/2015.

[68] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi. Cheap silicon: a myth or reality? picking the right data plane hardware for software defined networking. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 103–108. ACM, 2013.

[69] Project Floodlight. Floodlight firewall. http://docs.projectfloodlight.org/display/floodlightcontroller/Firewall+(Dev). Accessed: 22/05/2013.

[70] K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. RFC 2481, RFC Editor, 1999.

[71] C. E. Rothenberg, R. Chua, J. Bailey, M. Winter, C. N. A. Corrêa, S. C. de Lucena, M. R. Salvador, and T. D. Nadeau. When open source meets network control planes. *IEEE Computer*, 47(11):46–54, 2014.

[72] S. Russell. Thimble. In *Joint Techs, Winter 2013, Honolulu, Hawaii*, 2013.

[73] Sergiodc2, M. Pauley, and Scil100. TCP state diagram. https://upload.wikimedia.org/wikipedia/commons/f/f6/Tcp_state_diagram_fixed_new.svg, 2010.

[74] A. Shieha. Application Layer Firewall Using OpenFlow. Master's thesis, University of Colorado Boulder, 2014.

[75] J. Shiers. The Worldwide LHC Computing Grid (worldwide LCG). *Computer Physics Communications*, 177(1):219 – 223, 2007.

[76] J. Sommers, H. Kim, and P. Barford. Harpoon: a flow-level traffic generator for router and network tests. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 392–392. ACM, 2004.

[77] A. S. Tanenbaum. *Computer Networks, 3-rd edition*, chapter 5, pages 410–412. Prentice Hall, 1996.

[78] O. Titz. Why TCP over TCP is a bad idea. http://sites.inka.de/bigred/devel/tcp-tcp.html. Accessed: 22/07/2015.

[79] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger. Enabling Internet-wide deployment of explicit congestion notification. In *Passive and Active Measurement*, pages 193–205. Springer, 2015.

[80] R. Wang, D. Butnariu, J. Rexford, et al. Openflow-based server load balancing gone wild, 2011.

[81] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[82] Y. Yang and W. Yonggang. A Software Implementation for a hybrid Firewall Using Linux Netfilter. In *Software Engineering (WCSE), 2010 Second World Congress on*, volume 1, pages 18–21. IEEE, 2010.

[83] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.