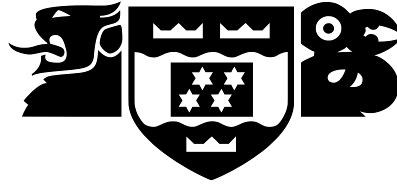
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Applying Formal Modelling to the Specification and Testing of SDN Network Functionality

Matt Stevens
School of Engineering and Computer Science
Victoria University of Wellington
Email: matt@stevens.net.nz

Supervisors: Bryan Ng, David Streader, Ian Welch

Submitted in partial fulfilment of the requirements for
Master of Engineering.

Abstract

Software Defined Networks offers a new paradigm to manage networks, one that favors centralised control over the distributed control used in legacy networks. This brings network operators potential efficiencies in capital investment, operating costs and wider choice in network appliance providers. We explore in this research whether these efficiencies apply to all network functionality by applying formal modelling to create a mathematically rigourous model of a service, a *firewall*, and using that model to derive tests that are ultimately applied to two SDN firewalls and a legacy stateful firewall. In the process we discover the only publicly available examples of SDN firewalls are not equivalent to legacy stateful firewalls and in fact create a security flaw that may be exploited by an attacker.

Acknowledgements

This work would not have been possible without the help and support of a large number of people.

My long suffering partner Megan McEwan, who went along with this mad idea and offered support for four years of study.

The lecturers and indeed the institution of Victoria University of Wellington, who provided guidance and support as well as cautionary advice on work loads; but ultimately facilitated achieving a four year Software Engineering degree in three years, with first class honours. To date I believe I am the only student of Victoria University to have achieved this with an engineering qualification.

The ability to start this research was enabled by an anonymous donor and the management team at the Faculty of Engineering. A special thank you to Suzan Hall for facilitating and huge thank you to my anonymous donor. I sincerely hope you see value in this research.

My three supervisors have been fantastic. Dr. Bryan Ng's background knowledge of networking and SDN is deep and was invaluable, as was his deft touch with offering constructive criticism. Dr. Ian Welch's security focus inspired examining stateful firewall behaviour where SDN decouples the algorithm and state from the forwarding plane; an architecture that offers some benefits but may create significant security problems. While Dr. David Streader bought a level of analytical thinking that I had not observed before, both exceedingly frustrating and yet ultimately highly satisfying. We spent many hours discussing the finer points of network algorithms and railing against loose terminology.

I broke the Universities internet. For that I apologise and I thank Mark Davies for the restraint he showed when he visited to fix it and the help supplied afterwards. His sharp question "Have you been playing with the network?" and my hesitant response "Probably, yes" is etched into my memory.

Contents

1 Introduction **1**
1.1 Research goals . . . 3
1.2 Contributions . . . 3

2 Background **5**
2.1 Legacy Networks . . . 5
2.1.1 Networking overview . . . 6
2.1.2 Middleboxes . . . 7
2.1.3 Criticisms . . . 9
2.2 A new paradigm — viewing the network as a system . . . 9
2.3 Software Defined Networks . . . 11
2.3.1 Control plane . . . 13
2.3.2 Forwarding plane . . . 14
2.3.3 OpenFlow compliant switches . . . 14
2.3.4 SDN Dogma . . . 14
2.3.5 Formal Properties of Networks . . . 15
2.4 Sourcing and managing in-line functionality . . . 16
2.4.1 SDN control of in-line functionality . . . 17
2.4.2 Proprietary Hardware . . . 18
2.4.3 SDN controller applications . . . 19
2.4.4 Switch waypoints and cloud services . . . 19
2.4.5 Language approaches . . . 20
2.4.6 Managing state . . . 20
2.4.7 NFV management . . . 21
2.5 Network Functions Virtualisation . . . 22
2.5.1 Properties of NFV . . . 22
2.5.2 Virtualisation options . . . 23
2.5.3 Speed of virtual in-line services . . . 23
2.5.4 Testing NFV equivalence . . . 23
2.5.5 Chains of network functionality . . . 23
2.5.6 Aggregating functionality on hardware . . . 24
2.6 Problems with comparing implementations . . . 24
2.7 Problems with SDN applications . . . 25
2.7.1 Placing the controller on the attack path . . . 26
2.7.2 Increasing control channel bandwidth . . . 26
2.7.3 Increasing SDN controller workload . . . 26
2.7.4 Many flow rules slow the switch . . . 27
2.7.5 Pushing local state across multiple SDN controllers . . . 27
2.7.6 Inconsistent SDN Controller state . . . 27

2.7.7 Pushing local state versus pulling switch statistics . . . 27
2.7.8 SDN Controller stress — may manifest slow network behaviour . . . 28
2.7.9 Convergence of the forwarding plane with the control plane . . . 28
2.7.10 Convergence of the network function with end hosts . . . 28
2.8 Firewalls — an example network function in SDN . . . 29
2.8.1 The Firewall Algorithm . . . 30
2.9 SDN research into network functionality . . . 32
2.10 Summary . . . 32

3 Formal Methods **33**
3.1 Overview . . . 34
3.2 Three common network functions . . . 35
3.2.1 NAT . . . 36
3.2.2 Load Balancer . . . 37
3.2.3 Firewall . . . 39
3.3 Formally describing a generic network function . . . 41
3.3.1 Packet flows . . . 42
3.3.2 A stateless network function . . . 43
3.3.3 A stateful network function . . . 45
3.3.4 A firewall example — stateless . . . 46
3.3.5 A firewall example — stateful . . . 48
3.3.6 Formally describing a chain of network functions . . . 49
3.4 Modelling Tools . . . 50
3.4.1 Rodin . . . 50
3.4.2 Event-B . . . 50
3.4.3 An example model built in Rodin, using Event-B . . . 50
3.4.4 Model refinement . . . 53
3.5 Model-Based Testing . . . 54
3.5.1 Existing research . . . 55
3.5.2 Other testing methods . . . 55
3.5.3 Modelling the software-under-test and its environment . . . 56
3.5.4 The MBT test process . . . 56
3.5.5 MBT test strategies . . . 57
3.5.6 State explosion . . . 59
3.5.7 Test metrics . . . 59
3.5.8 Defensible testing in industry . . . 59
3.6 Applying MBT to networking . . . 61
3.6.1 Industry Experience . . . 61
3.6.2 Hurdles to adopting MBT for SDN . . . 64

4 Research Direction **69**
4.1 SDN's third layer of state divergence . . . 69
4.2 The hypothesis . . . 70
4.3 Implementing MBT to test the hypothesis . . . 71

5 Applying Model Based Testing **73**
5.1 Generating a formal model of a stateful firewall . . . 73
5.1.1 The network environment . . . 73
5.1.2 The firewall model . . . 76
5.1.3 Creating test cases . . . 78

5.2 Creating the test harness . . . 79
5.2.1 Recording state . . . 79
5.2.2 Aggregating state . . . 80
5.2.3 Test-harness work flow . . . 81
5.2.4 Test and response servers . . . 82
5.2.5 TCP's unexpected behaviours . . . 82
5.2.6 Analysis Server and Results Presentation . . . 83
5.2.7 Discussion . . . 83
5.3 Networking the test harness . . . 84
5.3.1 Virtual machines . . . 84
5.3.2 Incorporating the Firewall . . . 86
5.3.3 Performing the tests . . . 87
5.4 Revisiting the firewall model . . . 87
5.4.1 State explosion . . . 88
5.4.2 Refactoring the test harness . . . 88
5.5 Revisiting the hypothesis . . . 90
5.6 Testing Multiple Firewalls . . . 90
5.6.1 Adjusting the test harness for SDN applications . . . 91
5.6.2 The Ryu firewall . . . 91
5.6.3 The Floodlight firewall . . . 92
5.6.4 Testing other firewalls . . . 93

6 Results and Discussion **95**
6.1 Key areas of interest . . . 95
6.1.1 Testing the SDN dogma — firewalls . . . 96
6.1.2 SDN's third layer of state divergence . . . 96
6.1.3 MBT in Networking . . . 97
6.2 Contributions . . . 97
6.3 Future Work . . . 98

Appendices **101**

A 15 Surveys of SDN research **103**

B Firewall in Event-B **105**
B.1 Firewall Context . . . 105
B.2 Firewall Model . . . 106

C MBT generated tests **119**
C.1 Initial tests . . . 119
C.2 Additional tests . . . 122

Figures

2.1 Legacy networks have the control plane bound to the forwarding hardware. . 5
2.2 The OSI model. 6
2.3 Network functionality (middleboxes) are controller agnostic, isolated from other functions and operate on traversing packets. 8
2.4 Automaton for a stateful network function 8
2.5 The Network Operating System — using abstraction to treat complexity . . . 10
2.6 Splitting the control and forwarding planes. 12
2.7 The predominate OpenFlow view of SDN, clever/slow control plane, dumb-/fast data plane. 15
2.8 SDN network functions are now incorporated into management applications. 17
2.9 Vendors of proprietary functions, advocate SDN with proprietary solutions. . 18
2.10 The NFV view of SDN, provide network functions that are agnostic to control plane solutions. 22
2.11 A service chain of middlebox algorithms 24
2.12 Implementations (i_1, i_2) are equivalent where they share the same set of modelled behaviours. 25
2.13 Domains of interest to a firewall 31

3.1 Finite State Machine for a stateful NAT (K = key, V = value) 36
3.2 Finite State Machine for a Load Balancer (K = key, V = value) 38
3.3 Finite State Machine for a switch as a stateless firewall 39
3.4 Finite State Machine for a stateful firewall — first model 40
3.5 Automaton for a stateless network function 43
3.6 Automaton for a stateful network function 45
3.7 Passing a flow through a chain of network functions and receiving the reply. . 49
3.8 Labelled transition system, packet events as transitions between states 51
3.9 Rodin and Event-B, model context 51
3.10 Rodin and Event-B, model machine — invariants and initialisation 52
3.11 Rodin and Event-B, model machine — packet_send 52
3.12 Rodin and Event-B, model machine — packet_arrived, packet_dropped 53
3.13 The Model Based Testing process 56
3.14 Incorporating modelling into the software development cycle 57
3.15 The Ariane 5 explosion, caused by a software bug, 4th June 1996. 60

4.1 The first layer of SDN state divergence. Distributed controllers holding different views of the forwarding plane 69
4.2 The second layer of SDN state divergence. Forwarding plane switches not in the same state as the controller view . 70
4.3 The third layer of SDN state divergence. Network functions not holding the same connection state as end hosts . 70

5.1 An event machine for sending and receiving network packets. 74
5.2 An event machine for sending and receiving network packets. 75
5.3 An event machine for a stateful firewall (occupies a chokepoint). 77
5.4 An application racing the kernel to establish a TCP session, will fail. 80
5.5 Sketch of MBT test server algorithm 81
5.6 The test harness framework 84
5.7 The test harness, including response servers and the firewall 86
5.8 The external host should not be able to open a TCP session. 87
5.9 Test and response servers swap active and listening roles, mid test. 90

Chapter 1

Introduction

> "The use of explicit models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, not reproducible, not documented, lacking detailed rationales for the test design, and dependent on the ingenuity of single engineers." —Utting *et al.* (2012) [1]

It can be anticipated that investigations into software disasters will find that a catastrophic bug was preventable. For example, the Mars lander software bug was found to be caused by a miscommunication between two coding teams [2]. The academic literature has many fascinating examples of disasters, used to impress upon young computer scientists the importance of testing and that risk can and should be managed. This helps reinforce the need for testing to be done at the individual, team and system level by developers. However the examples of real life failures are also replete with developer assumptions, incidental code complexity, miscommunicated requirements, workarounds and commercial pressures leading to rigour being sacrificed for convenience or profits.

Software consumers, both technical and non-technical, find it difficult and time-consuming to ascertain whether software does the job required and is error free. Even where code is available to review, this is time consuming and requires technical expertise. Where unit tests are available, these are likely incomplete and will incorporate undocumented developer assumptions.

This research sits at the intersection of multiple technologies and seeks new ways they can leverage each other. The parts are the decoupling of the networking control plane from the forwarding plane, the decoupling of network algorithms from network hardware, new software virtualisation technologies — virtual machines and containerisation, two new classes of generic hardware — hosts and switches, formal methods, model-driven development which may lead to compiling code from mathematically proven models and model-based testing which uses mathematically proven models to validate hand crafted software artefacts.

Networking as a technology coalesced thirty years ago and has largely satisfied the networking community since. There have been evolutionary developments in networking, but nothing architecturally revolutionary for decades. The current rapid growth in networking, propelled by mobile devices, streaming media and the upcoming *Internet of Things* is pushing current network technologies to their limits, so much so that networking today is perceived as labour intensive and error prone, difficult to develop for and adopting new technologies is high risk. This typically means products must be proven before use and have big names behind them to accept liability when these products fail. We explore this existing network infrastructure in more detail in Section 2.1.

Two new technologies have captured researchers and industries interest. Software De-

fined Networks (SDN) and Network Functions Virtualisation (NFV). Both technologies are being used today by entrepreneurs, cloud service providers and telecommunication providers; including big names such as Google and Microsoft. The first enabling technology is NFV, which is the separation of network software from network hardware, this allows highly portable network functions running on virtual machines (VMs). It de-couples the software innovation cycle from the hardware innovation cycle, in the process opening up network functions to greater innovation while avoiding vendor lock-in. The architectural revolution is often labelled SDN, it is the separation of the network's forwarding and control planes, allowing the control plane to be centralised and automated, offering the potential for better resource optimisation and the automation of many labour intensive networking tasks which in turn will reduce costs and human error. We explore these topics further in Section 2.3 and 2.5.

The relationship between SDN and NFV has proved contentious. This was found in a survey of 15 recent SDN survey papers (see Appendix A) which overwhelmingly stated the SDN preference of consolidating algorithms and state in the control plane (as opposed to using NFV in the forwarding plane). Partly in response, this author published a paper *Global and Local Knowledge in SDN* [3] to promote a more nuanced approach, highlighting the advantages of state being treated in each plane. The Open Network Foundation has since published TR-518 *Relationship of SDN and NFV* to address similar issues and assist the two fields to capitalise on each others strengths instead of "reinventing the wheel" [4].

As an example of this conflict skewing perceptions, the author was surprised to find quite late in this research that the SDN dogma driving network functions out of the forwarding plane, has not yet resulted in either a commercial or open source stateful SDN firewall.

Network Engineers are risk averse. This is for very good reason, they control large complicated and fragile communication systems and there are often economic penalties for mistakes. The training and focus for most Network Engineers is tools based rather than on fundamental underlying principles in topics such as architecture, testing and formal methods. Typically Network Engineers do limited programming. They write scripts in, for example, Bash and Python to facilitate common tasks such as network configuration, but do not write programs or implement applications that provide network functionality. They rely instead on innovation from big name vendors which is necessarily slow and thorough due to the responsibility they adopt. New technologies from new vendors are consequently perceived as high risk by Network Engineers and to be avoided, contributing to network ossification. To make inroads into this space, new tools and techniques for testing networks and network functionality need to be developed in order to allow validation of network functionality implemented using novel mechanisms such as SDN. These tools and techniques need to be tailored for use by existing Network Engineers; ideally push button solutions to lower the barrier for entry.

Model-Based Testing (MBT) [5, 6] offers a potential solution. It is a risk management methodology that is independent of the developers creating the software. It can occur in parallel to developing the software artefact, does not delay the project and can be started early in the software development life cycle, finding architectural problems when they are trivial to fix [7]. The act of creating a formal model not only provides valuable insights into the software and its operating environment, but also may act as a benchmark (providing black-box tests) to developers. The model's logic and insights can potentially also be explained to senior non-technical stakeholders and may be peer reviewed by other engineers, a practice common in more mature engineering disciplines.

Formal methods and black-box model-based testing are a potential solution with a huge adoption hurdle, it is not push button technology. In addition it is not sufficient in isolation.

However, in concert with other methods such as code inspection and performance tests it can form part of a package that can ensure an algorithm behaves as expected, is reliable, secure and fast. We explore formal methods and model-based testing in chapter 3.

This research pulls together these interesting developments and seeks to formally define an example network algorithm describing stateful network firewall functionality and prototype a tool that may be used to facilitate the work of Network Engineers in validating implementations of network functionality. In the process we expose and demonstrate SDN's third layer of state divergence.

1.1 Research goals

This research explores applying MBT to well understood network functionality. It seeks to demonstrate that not all firewalls behave as expected and that SDN firewalls that rely on state in the control plane, are likely to fail some tests as a consequence of the SDN architecture.

1.2 Contributions

This thesis makes multiple contributions to SDN research, including using formal methods, modelling and MBT as a framework to explore the implementation of network functionality in both legacy networks and SDN. The case for SDN to purge network functions from the forwarding plane is examined and found to be a widely held yet unproven dogma. Several examples of common middleboxes are examined and contrasted with their SDN implementations to find the architecture used in SDN appears to limit the ability of SDN switches to perform as network functions. This means there is a capability trade-off that is not expressed in the SDN literature, a pre-requisite for discussing how novel SDN management functions may overcome the resulting problems. To illustrate the difference a stateful firewall is formally modelled and utilising model-based testing, an in-line implementation of a firewall is compared with two SDN implementations in order to find if the SDN firewalls have traded away features in order to be stateless and managed by the controller. Specifically the main contributions of this thesis are:

1. An analysis of SDN literature to find ten problems that may be exacerbated by moving network function algorithm and state from the forwarding plane to the controller.
2. The creation of several prototypes, including a formal model of a generic network function, two formal models of a firewall network function and a MBT test harness.
3. The first application of MBT to network functions. The creation of tests from a formal model of a network function and applying them to a test harness that tests an implementation.
4. Possibly the first use of MBT as a tool for determining behavioural equivalence between black-box implementations, to the extent that behaviours have been modelled. Useful in this domain because of the volume of vendors offering network functionality that provide similar behavioural properties.
5. The creation of abstractions of middlebox functionality which may assist in proving fundamental network properties such as no loops, no black holes and reachability in the presence of dynamic network functionality. These are the subject of ongoing research.

6. A third layer of state divergence in SDN. The first two layers are discussed (state divergence in the logically centralised control plane and state divergence in the control to forwarding plane) and the third (state divergence between network function and end hosts) is described and demonstrated.

Chapter 2

Background

There has been a range of research applying formal methods and model checking to networks. For example, the NICE tool which has applied model checking to the new network technologies of Software Defined Network (SDN) controllers and switches [8]. Other recent research explores network architecture, including alternative ways to abstract middlebox functionality [9–11].

Current networks have been described as difficult to manage because of the focus on low-level constructs and the lack of high-level abstractions, for example installing and wiring individual switches and middleboxes, writing rules for individual packet flows and assigning IP addresses for users [12]. Utilising a logically centralised controller can abstract away and automate this detail using management applications while utilising virtual machines (VMs) means network functionality need no longer be tied to a physical device.

The field of computer networking has had little exposure to Model Based Testing (MBT) — so far the only example found of MBT (as opposed to model checking) is testing Service Oriented Architecture (SOA) integration from choreography models [13]. This work which examines network functionality using formal methods and MBT, will add to this research.

From here we discuss legacy networks before discussing the idea of a network as a system. SDN is described, followed by outlining what in-line functionality is. Network Functions Virtualisation (NFV) is described before the chapter finishes with reviewing problems in SDN and discussing the firewall algorithm as an example network function.

2.1 Legacy Networks

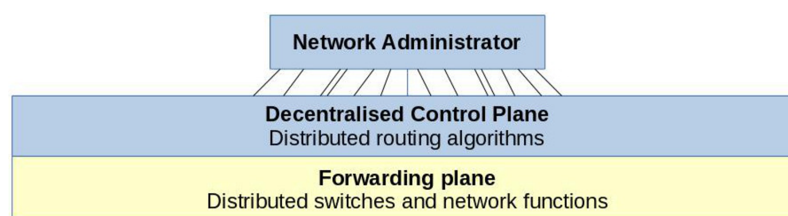


Figure 2.1: Legacy networks have the control plane bound to the forwarding hardware.

The Internet is a network of networks. It might be described as a federation of networks with agreed communication protocols that allow hosts within and between networks to communicate with each other. In legacy networks these protocols utilise distributed algorithms bound to switches and routers (see Figure 2.1) that discover each other and manage

routing decisions. Within switches and routers, these distributed algorithms form the control plane while rule tables and network interface cards (NICs) connected to physical cables form the forwarding plane.

Legacy networks are primarily comprised of black-box hardware. This is hardware provided by big brand names like Cisco, Juniper and Huawei to provide important network functionality such as routers and firewalls. Manufacturers invest heavily into research seeking to optimise proprietary hardware which has brought many benefits including greatly increased reliability and very high throughput.

Network hardware falls into two broad categories. Hardware concerned with routing (the switches and routers) and hardware concerned with improving network properties — the definition of properties is left deliberately wide ranging. For example, firewalls improve security, intrusion detection systems (IDS) improve intruder detection, caching improves latency (response time) to the user and reduces network load for the provider, load balancing allows the use of parallelism to meet increased user demand. There are many examples of network properties that can be improved with another network function and there is no doubt more being researched.

2.1.1 Networking overview

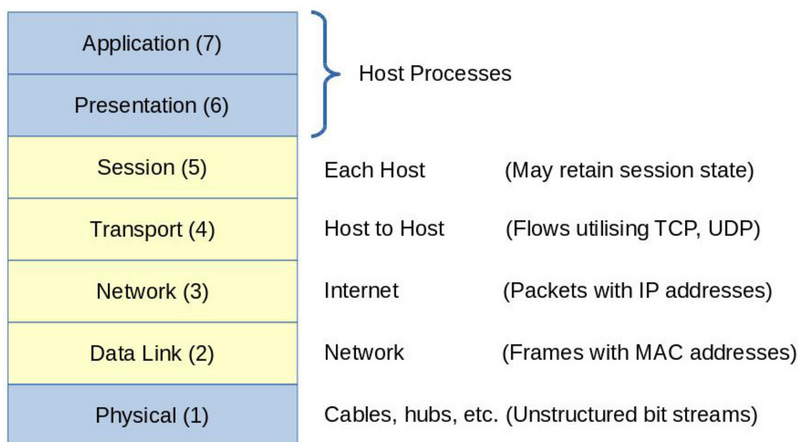


Figure 2.2: The OSI model.

Many discussions describing existing networks rely on the Open Systems Interconnection (OSI) model to provide a framework for the concepts. In brief the OSI model (Figure 2.2) recognises 7 layers within networking, from the application layer down to the physical layer comprised of cables and hardware. This discussion primarily refers to layers 2-5, the data link, network, transport and session layers.

Switches provide layer 2 (data link) services, that is they are concerned with local area networks (LANs) consisting of switches and hosts, that are typically physically wired together (layer 1). All network devices have a unique media access control (MAC) address which are used within a LAN for packet addressing and each switch utilises a table matching MAC addresses with the correct outgoing port and wire. The table is populated by listening for broadcasts from new arrivals and noting which wire (or port) each new arrival occupies. If a packet is received with an unknown MAC address, the switch will broadcast that fact in the hope another switch will provide a new table entry for future packets. Switches have several other protocols to resolve collisions on the wire, detect liveness, etc.

Routers provide layer 3 (network) services using Internet Protocol (IP) addresses to facilitate message passing between networks, potentially via intervening networks. For ex-

ample, a gateway router will sit on the border of a LAN and will listen for broadcasts from other routers about IP addresses they know. It utilises a table matching IP addresses with an outgoing physical port. If the IP address is unknown, or is perhaps known by several routers, the router uses best guess algorithms to pick the next hop destination. Within a LAN, routers may match a host's IP address with MAC addresses and resolve routing using switch functionality.

End hosts communicate using layer 4 (transport) protocols like TCP or UDP. The TCP/UDP packet includes a port number (not related to physical ports) that identifies the application to the two end hosts and also includes a packet sequence number. While layers 2 and 3 represent best effort communications, layer 4 protocols may (TCP) or may not (UDP) make the communication reliable, checking if packets are missing and reassembling them into the correct order. All Internet browsers use port 80, common email clients use ports 25 and 110.

Hosts may have need to remember data during a session which is layer 5 (sessions). For example an online shopping basket. Should a service be provided by several hosts in parallel it becomes important to consistently use the same host.

An individual packet forms part of a packet flow which can be identified through common packet header fields; typically the five tuple of message protocol, source IP address and port plus destination IP address and port. An application, (for example, Netflix) will partition its service (a streaming movie) into many packets of typically 1400 bytes[1]

A packet's 1400 bytes of data is encapsulated by the layer 4 transport header which includes the application source and destination ports and the sequence number to allow reordering at the destination. Next the layer 3 network layer encapsulates the packet adding the source and destination IP addresses. The packet is then handed to the layer 2 data link layer which encapsulates the packet with the MAC address of the host as the source and the gateway router's MAC addresses as the destination. Finally the gateway on receipt of the packet will decapsulate it (removing the layer 2 wrapper) and forward it to the layer 3 network layer destination, using best guess if the destination is unknown.

Elephant flows are large data flows such as movie streaming or big data that can be identified from the volume of packets [14]. Identifying these flows and redirecting them to dedicated paths is an example of network functionality. The goal in this case is to segregate large flows to prevent performance degradation within a network while potentially allowing other functionality, like traffic accelerators, to process the large flows on dedicated paths. Network functions like these, aim to improve properties of the Internet and are typically facilitated by middleboxes providing the network functionality.

2.1.2 Middleboxes

Middleboxes are defined by RFC3234 as intermediary boxes performing non-router functions on the path between two hosts [15]. These are, for example, firewalls, load balancers and NATs, performing operations largely at line speed on packets as they pass. Figure 2.3 shows a middlebox positioned between switches **A** and **B**. Typically these are proprietary middleboxes which are autonomous algorithms (finite state machines) optimised to run on fast, but expensive, Application Specific Integrated Circuits (ASICs).

Middleboxes typically sit between switches, hence 'middle' and they are typically *stateful* meaning they use and manage dynamic state in order to provide functionality. To call a middlebox or network functionality *stateless* is not claiming a lack of state, rather that its state is not dynamic, any state changes are provided by a third party. According to surveys

[1] A byte is 8 bits and a bit is a 0 or 1, 1400 bytes is therefore 11200 0's and 1's. A movie of 8MB breaks down into many packets. The small packet size allows networks to interlace data streams more efficiently.

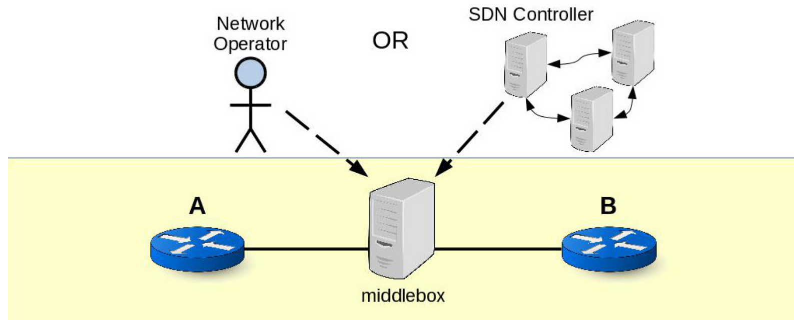


Figure 2.3: Network functionality (middleboxes) are controller agnostic, isolated from other functions and operate on traversing packets.

conducted by Sherry *et al.* (2012) and Sekar *et al.* (2012) [10, 11] just under half the hardware in current networks are middleboxes while the other half are switches and routers.

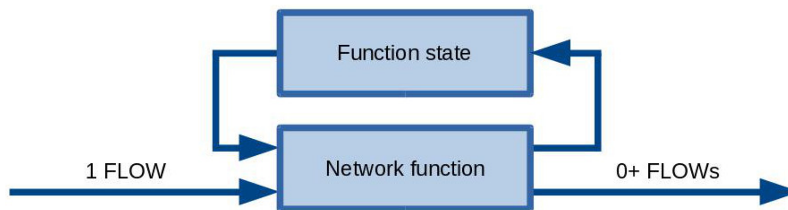


Figure 2.4: Automaton for a stateful network function

Middleboxes have a common feature that makes them ideal for abstraction. They operate in isolation and act only on the network packets they see — redirecting, dropping, modifying and/or retaining state information. Put another way, a middlebox blindly applies a set of rules to the packets passing through it and may keep state information should it need to act on other related packets or a reply. As a function that takes packets as inputs, uses state and generates packets as outputs, they can be treated as a finite state machine (FSM). Figure 2.4 shows the automata for a stateful network function.

Middleboxes implemented as FSM in the forwarding plane offer several attractive features. FSM functionality is encapsulated and highly cohesive, they are modular, may be replicated for scaling up or down and may be replaced with upgraded versions. Typically FSM supports pipelining and may be linked in service chains. FSM may also be proven using model checkers. The placement of FSM in the forwarding plane and decisions on when to update or replace them, uses a centralised view of the network provided by the network controller which may be human, SDN or proprietary.

Middleboxes will typically be placed between two hosts with the intention of processing all traffic passing between them. If the middlebox provides a service for a domain, one host will be in the domain with the other outside, the path the middlebox is on will transition between the two. Alternative architectures exist, such as placing the same functionality on every host instead which avoids traffic choke-points. Typically the choice of placement is based on resource efficiency but other properties may also have an impact, for example, a firewall placed at the border can stop domain mapping or denial of service (DoS) attacks while a distributed firewall implicitly allows domain mapping and in the event of a DoS attack, the entire domain will be affected, potentially hampering efforts to respond on the network.

There are several highly desirable properties that middleboxes should possess; correctness, flexibility, efficiency and fault tolerance. In addition middleboxes operate in a dy-

namic environment with potentially high rates of network, policy and potentially middlebox churn [16] which are concerns for the management of middleboxes.

2.1.3 Criticisms

While the distributed algorithms used in network routing are very effective, they have come under criticism by a variety of authors for their shortcomings, for example, the **network convergence time** when recovering from a failure, the **lack of a centralised view** which may allow network resources to be optimised or the need for distributed algorithms hosted by different parties to **maintain pre-existing protocols**.

To create a network comprised of switches, routers and middleboxes is **labour intensive and error prone** when hardware is physically wired into the network and individually configured. In addition distributed algorithms can be **challenging to debug**. To assist, vendors have developed **proprietary hardware** and certification courses for technicians. Vendors also provide equipment to tertiary colleges at discount prices, meaning many graduates qualify with a knowledge of and predisposition towards a vendors products. This creates problems with **vendor lock-in**, facilitated by existing investment in proprietary hardware and staff training on proprietary equipment.

The difficulty of getting agreement to change or risk conducting live experiments with new protocols and network functions has contributed to what has been described as the **ossification of the Internet** [17]. This is a consequence of several factors; service agreements specify reliability metrics, leading many Network Engineers to **prefer others conduct experiments** while they adopt tried and true technologies; **proprietary vendors dominate** the hardware space, they represent 'tried and true', leaving **little scope for innovative startups**; and technician product certification which while necessary due to the complexity of running a network also means **the workforce is largely trained on one vendors line of products**, contributing to vendor lock in.

The hardware and service agreements for **proprietary solutions are expensive**. As is using **hardware duplication** to provide fault redundancy or capacity for peak flows. The power consumption of **under-utilised hardware** adds to running costs and the **capital invested** in under-utilised hardware is no longer available for other purposes. **Expensive operations** include; transitioning from one vendor to another, retraining the work force, managing the current stock of hardware and being responsive to network needs by moving hardware from one part of the country to another. These issues drive the research desire to provide efficiency and reliability at much reduced costs.

There is a common theme underlying approaches to improving on legacy systems and mitigating these criticisms. It is to adopt an integrated approach seeking to treat the network as a system.

2.2 A new paradigm — viewing the network as a system

In the computing space outside of networking, abstractions provided by operating systems and programming languages have enabled systems of greater complexity which has in turn enabled further efforts to adopt greater abstractions. As an example, the machine language code to enable read and write access to the hard drive occupies multiple pages. In comparison, the high level languages Python and Ruby may use the Active Record pattern described by Martin Fowler (2002), which can be implemented using only a few lines of code [18]. There have been several calls to bring the benefits of high level abstractions to networking.

Casado *et al.* (2007) with Ethane, asks the question "How could we change the enterprise network architecture to make it more manageable?" [19]. He proposed a central controller

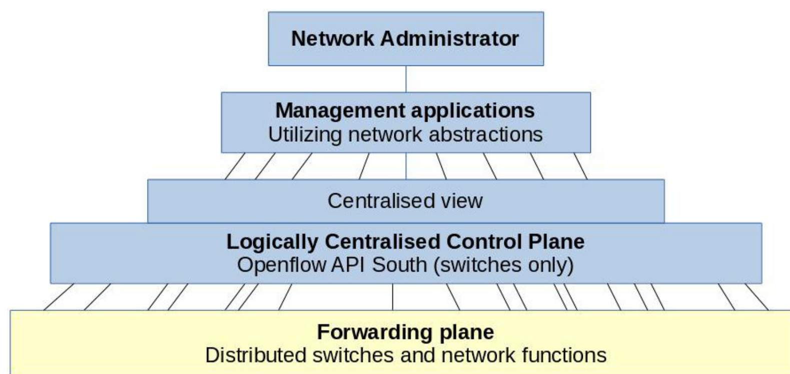


Figure 2.5: The Network Operating System — using abstraction to treat complexity

managing simple network switches. This separates the control plane from the forwarding hardware as illustrated in Figure 2.5. A network operator tells the controller about, for example, network security policies which are translated to switch rules for the forwarding plane. When a switch element asks the controller about an unknown flow, a suitable flow table rule is sent back to the switch. This was successfully prototyped in a university with 300 registered hosts and several hundred users. The devices hosted ranged from wireless devices to printers and workstations.

From this research came two influential papers, the first by McKeown *et al.* (March 2008) describes OpenFlow, an API for simple switches [17]. OpenFlow compliant switches contain flow tables with three fields; a match field for a packet header that defines the flow, an action to be taken with the flow and statistics counting packets and bytes for each flow plus a time-stamp for the last time this rule was matched. The OpenFlow API between the controller and basic switches allows reading switch statistics (which contribute to the controllers centralised view) plus requesting and returning flow rules of the form {match, action}.

The second influential paper was by Gude *et al.* (July 2008) and describes NOX, an implementation of a network controller that creates a centralised view of the network and offers high level abstractions [12]. It uses OpenFlow compliant switches, slaved to a central decision maker that overseas the entire network. NOX is provided as an example of a Network Operating System that can host network management applications.

While these researchers laid the foundation for what is now known as Software Defined Networking (SDN), the movement towards virtualisation of applications was already getting stronger. In brief a virtual machine (VM) runs in isolation on a host system with its own operating system supporting private applications. In this way an operating system and application can be packaged with a VM and will run on any host that supports that VM. Similar to Java which revolutionised programming with its language portability, VMs provide application portability.

Research into Xen by Barham *et al.* (2003) [20], prototyped a high performance VM with strong isolation properties and low overheads on the host machine. This demonstrated the potential for virtualisation on generic hardware, with a design goal of 100 VMs on a modern server[2]. With Xen and VM technology, a range of benefits became possible such as server consolidation and application mobility.

A few years later and Wang & Ng (2010) in their examination of virtualisation within the Amazon cloud describe how Xen VMs are used by a major service provider [21]. The

[2]An x86 Dell 2650 dual processor 2.4GHz Xeon server with 2GB RAM — memory limitations proved to be the constraint, the 100 VM goal was achieved with each VM hosting a minimal OS of 4.2MB.

technology has enabled services like Amazon to sell compute instances easily, on demand and for short durations. Sherry *et al.* (2012) describes how these might be used to provide middlebox functionality on demand and avoid the need to have overcapacity in proprietary hardware to handle peak loads [11].

Open vSwitch is an example of a virtualised network function. Developed by Pfaff *et al.* (2009) Open vSwitch allows OpenFlow switch functionality to be provided by generic hosts [22, 23]. This led to the potential for virtualising the networking layer and also to co-hosting an Open vSwitch with other virtualised network functions, potentially decoupling networking functionality from its switching capabilities.

ClickOS by Martins *et al.* (2014) builds on the interest in NFV and capitalises on inexpensive commodity hardware, for example x86 servers with 10GB NICs. ClickOS is a Xen based software platform optimised for packet processing [24]. It is small at 5MB, boots quickly in about 30 milliseconds and adds little to latency, about 45 microseconds. For simple processing of 1500B packets it can achieve 9.68GB/s throughput.

Taken together these technologies are allowing networking to abstract away from first the hardware, using virtualisation technologies, second from the networking details by using centralised controllers as a platform to build management tools on. This abstraction brings a number of benefits. It makes programs easier to write, reason about and debug; an abstract implementation can be copied and re-used, and if popular and open-source, many eyes ensure it is debugged quickly and is robust. Together these are starting to address the concerns outlined earlier and pave the way to creating higher level abstractions.

While a network operating system is the goal, the research is still falling short. There is no consensus on an API for the applications that might use such an operating system and no consensus on how network functionality should be managed in the forwarding plane, aside from OpenFlow compliant switches. Despite these concerns, research and industry are coalescing around SDN as the preferred technology.

2.3 Software Defined Networks

SDN as a network operating system is characterised by the separation of the network packet forwarding plane from the control plane. This enables the move away from a hardware plane characterised by switches, routers and middleboxes running proprietary systems, nurtured by talented technicians and network administrators. Instead moving toward homogeneous hardware capable of supporting a forwarding plane and a control plane that is managed solely by network administrators. The intended result is better use of resources, both human and capital with the control plane providing a foundation for management applications.

SDN's scope is not global (as in planet wide), it is a solution for a single entity's network (referred to from here as a network administrator, however it could well be a group of network administrators working for a large enterprise managing networks of networks). While theoretically SDN may have a super controller to rule them all, just as it may contain hierarchies of controllers [25], it is unlikely that the network administrators that control each of the Internet's federated networks will give up control or security of their networks to a third party. This means there is still a requirement for protocols to communicate between networks, for example BGP (Border Gateway Protocol).

The separation of control and forwarding planes is shown in Figure 2.6 where a hardware bus is replaced by a network protocol, for example, OpenFlow. This allows the controller to be located remotely and for switch and controller software and hardware to develop independently of each other. It may be noticed that the split planes do not illustrate

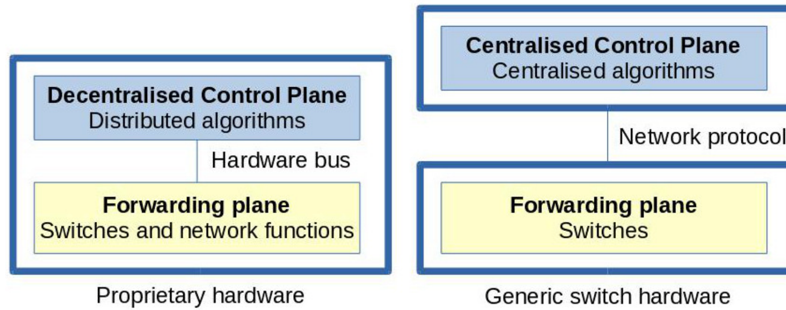


Figure 2.6: Splitting the control and forwarding planes.

network functions such as middleboxes. Network functions do not yet have a commonly accepted network protocol.

Research and industry are largely working with the OpenFlow API which offers useful protocols to pull flow statistics from OpenFlow switches, push switch requests to the controller and for the controller to push new rules back. This OpenFlow facilitated separation brings several benefits including opening up both the simple switch hardware market and the software controller market to competition.

However the OpenFlow API does not offer a complete solution to controlling the forwarding plane. The intuition is that switches and routers have proved easy to categorise and reason about and achieving a broad consensus on a switch API has proved possible. However in response to middlebox functionality which is characterised by complexity and diversity, adherents to OpenFlow have largely adopted a switch centric view of SDN and an aversion to middleboxes, the resulting research push has been to remove all network functions from the forwarding plane (see Appendix A).

SDN as an architecture, scales from controlling many switches to one. Ryu-faucet, is an example of a layer 2 SDN controller[3] that might control a single OpenFlow compliant switch and host a management application. Deployed as a single unit onto a generic host, the black-box behaviour may be indistinguishable from a middlebox.

At the other extreme in 2011 Google deployed SDN on its B4 inter-datacenter WAN, the largest production network at Google[4,5]. The underlying goal was to manage the WAN as a fabric rather than as a collection of hardware. It required custom building their own OpenFlow switches and they achieved a single network operating system controlling three forwarding planes; optical, MPLS and IP.

The benefits of SDN include the potential for resource optimisation (both hardware and personal) by utilising the centralised view, better security and efficiency through flow analysis, greater fault tolerance through automating fail-overs and the creation of alternative fail-over paths[6], and increased research rate on live networks where new algorithms can be incorporated incrementally into a production network and rolled back in a controlled manner.

[3]See ryu-faucet repository; https://pypi.python.org/pypi/ryu-faucet

[4]See presentation slides; unknown author (2012),
http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf

[5]Also B Koley (2014),
http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42948.pdf

[6]Fail-over refers to the act of detecting failed hardware or software and responding to restore connectivity between end-hosts. This needs to be done swiftly, under 50ms, to prevent end hosts from assuming the service has been lost.

2.3.1 Control plane

The SDN control plane with its awareness of topology, flows and state enables a powerful centralised network view that can drive decisions on traffic routing. It can also direct the network to respond to equipment failure, intruders, attacks, adapt to changing network load, enhance network-wide security and allows more efficient resource management.

Network complexity can be managed through the control plane. For example, 'Intents' allow the network operator to manage a network using abstractions [26]. Implementing the abstraction is typically performed in the controller which defines specific instructions for individual switches.

To resolve early concerns about the controller becoming a single point of failure (in the event of, for example, hardware failures, planned outages or network partitioning) multiple controllers are used to ensure high availability. The controller is therefore considered a logically centralised controller to reflect that the controller instances and controller state may be geographically distributed.

Ros & Ruiz (2014) explores SDN controller reliability, demonstrating that to achieve fine nines reliability (99.999% up time) in a network requires a significant number of controllers and redundant failover paths. The networks studied ranged from 6 to 55 nodes (ignoring one outlier at 197 nodes) and required a median 6 controllers. 75% of the networks studied required 10 controllers or less [27]. This might be described as controller explosion which may potentially complicate managing and replicating shared controller state.

The controller redundancy obtained comes at a cost, a consequence of the controllers acting as a distributed database (holding shared controller state) [28]. CAP theorem (*Consistency, Availability, Partition tolerance*) is a database concept that suggests the architect of a distributed database may pick two of the three properties and must manage weakness in the third, albeit Brewer (2012) suggests in "CAP twelve years later" that the trade off today may be less binary, more of a continuum [29,30].

From the perspective of an SDN distributed controller; *Availability* is ensuring controller availability at all times. This is often achieved through controller replication. *Partition Tolerance* is ensuring that link failures do not prevent controller services. This is often achieved through controller distribution, which may also improve local controller latency. *Consistency* is ensuring multiple requests return the same answer — that multiple controllers present the same view of the network. This can be achieved with locking databases until they are consistent, at the cost of slowing down controller reaction time which can dramatically impact on *availability*. Canini *et al.* (2013) presents a novel example of this by using a locking mechanism in middleware between the controller and switches [31].

Choosing high availability and partition tolerance, means managing consistency which is referred to as adopting the *eventually consistent* model where over time the controllers will converge towards consistency. Should global state change stop, the controllers will achieve consistency, but in normal operations the controllers are expected to be in a state of perpetual convergence.

We refer to this trait later as the first layer of state divergence which may lead to poor network properties such as loops, black holes and unexpected reachability problems.

One way to manage *consistency* is to reduce the amount of state in the controller in order to reduce the effort required to manage that state across all instances. Another strategy is to ensure management applications react gracefully when inconsistencies are found. However it might be observed this passes the problem to third party management application providers whom may fall short.

Other convergence problems exist between the control plane and the forwarding plane, and between network functions and end hosts (the second and third layers of SDN state

divergence). These are explored further in the following sections.

2.3.2 Forwarding plane

The SDN ideal is for the forwarding plane to be populated by simple switches capable of high speed packet processing. These switches should comply with an API and should be commodity products. In operation they have tables holding a packet header tuple as a key, an action such as 'drop' or 'forward' and collect flow data using counters and a time-stamp for the last time the flow was used.

The most popular API to date is OpenFlow. It defines protocols for accessing switch data and for updating flow rules. OpenFlow v1.5 is the current specification, however most available switches offer v1.3. It is expected this will change rapidly as more switches are made v1.4 compliant.

2.3.3 OpenFlow compliant switches

OpenFlow's acceptance has resulted in a range of commercially available OpenFlow compliant commodity switches. These cover a range of capabilities from industrial use to research and home hobbyist. In addition to generic hardware switches, Open vSwitch is a virtualised software switch that is OpenFlow compliant for use in VMs.

OpenFlow is still developing as an API. For example, over time the specification has expanded from the initial 12 (v1.0) header match fields to 42 (v1.4) and 45 (v1.5). In the process switch complexity is increasing and there are concerns that hard coding a finite number of commonly accepted header fields may potentially stifle future innovation looking to develop new protocols. Researchers are looking to address this, for example, P4 suggests that switches should be programmable [32]. Rather than a switch being capable of matching all fields, a switch could have a smaller set of generic fields that can be programmed to hold meaning for this part of the network.

Stressed switches are also of concern, Wood *et al.* (2015) found that Open vSwitch's 10GB/s throughput drops to 4GB/s when 10% of the packets are sent to the controller [33]. Kuzniar *et al.* (2015) found a number of issues; that switches with more than two batches of rule up-dates in-flight overload the switches capacity to process those updates; that a switches flow table update rate decreases as the number of entries increase; that prioritising updates decreases the update rate and that rule modifications are slower than additions and deletions [34].

Switches hold finite rule space. Utilising this space efficiently in the presence of network dynamism is challenging. For example, performing a consistent commit of rules that does not lose flows is a multi-step process that includes tagging flows, installing rules into the network, waiting for packets using the old rules to exit, then removing the old rules. This requires twice the rules space on a finite resource. Katta *et al.* (2013) propose a method that reduces the rule space needed but reduces network responsiveness (trading time for rule space) [35].

2.3.4 SDN Dogma

After reviewing fifteen recent surveys of SDN research (see Appendix A), it is apparent there is a switch centric view within the research community that has created the dogma that in-line functionality should be purged from the forwarding plane and placed in the controller. Figure 2.7 shows this view as a clever, programmable control plane and a dumb, fast forwarding plane. The assumption is that in-line functionality slows and adds complexity to

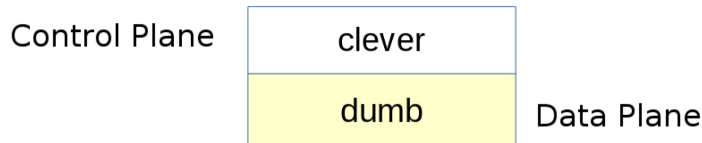


Figure 2.7: The predominate OpenFlow view of SDN, clever/slow control plane, dumb/fast data plane.

the forwarding plane. The preferred solution is to use only stateless (no dynamic state) network functionality and place the network function algorithm in the controller. However the new found ability to program the control plane does not imply this is the only or best place for network functionality.

In effect this dogma takes a de-coupled application that can be developed and tested in isolation, is highly portable and can be easily replicated across a network; and replaces it with a remote application coupled to a controller implementation (no widely accepted application API exists yet) resulting in replacing proprietary hardware vendor lock-in with SDN Controller lock-in.

There are other consequences of adopting this preferred architecture including increasing the controller work-load, increasing the volume of centralised state, increasing switch rule updates, increasing the volume of rules held by switches and increased use of switch buffers. We will revisit these topics shortly and discuss existing research.

This author's paper *Global and Local Knowledge in SDN* [3], discusses the SDN view that local knowledge should become part of the centralised state. It points out that state held in the logically centralised controller is replicated several times for redundancy and requires algorithms to manage the increased state convergence problems between distributed databases and also between the control and forwarding planes.

For example caching is provided by a middlebox. Its function is to store and respond with information commonly requested by end users in order to improve response times and conserve upstream computational and bandwidth resources. This cannot be achieved with an OpenFlow switch and while there are outspoken views that such activities represent computation that must be purged from the forwarding plane, there is presently no research backing up these claims, it is a philosophical stance (see Appendix A).

SDN researchers are largely ignoring research into NFV, which easily and efficiently provides stateful applications in the forwarding plane. NFV treats packets in-line and fast decisions can be made without reference to a third party application. Recent NFV research papers have demonstrated them as highly portable, fast to install and performing at line speeds. We discuss NFV in greater depth in Section 2.5.

2.3.5 Formal Properties of Networks

Establishing network properties such as no loops, no black holes and reachability is difficult in a legacy network given the dynamism of distributed algorithms. Any property proved is transient and may be almost immediately invalidated by the network.

Proving properties in SDN is attractive given the availability of the centralised view, typically represented as a connected graph. Ideally this capability combined with controller directed rule updates allows the controller to not only spot problems but also pre-empt them. However research into proving properties of SDN networks has so far proved difficult, with researchers adopting simplified models of dynamic network functionality to make the problem more tractable [36].

However, complicating the SDN approach, are concerns expressed over whether the

forwarding and control plane views converge fast enough [27, 30, 34, 35, 37–40] (the second layer of SDN state divergence). This can result in black holes, loops and reachability issues, as a consequence of switches not reacting to controller instructions fast enough. For example if a chain of switches are under stress due to recovering from failed hardware or a DoS attack, the stressed switches may not be able to process controller instructions in a timely manner.

Switch implementations have been found to be inconsistent. For example, they often allow temporary divergence, up to 400ms and may implement switch updates out-of-order. The implementation of the barrier command across switch implementations is specified in the OpenFlow specifications and intended to provide update guarantees. However the implementation in switches has been demonstrated as inconsistent, and "cannot be trusted" [34].

> "Ignoring the problem leads to an incorrect network state that may drop packets, or even worse, send them to an undesired destination!"
>
> Kuzniar, Peresini & Kostic (2015)

Forwarding and control plane convergence, is explored further in Section 2.7.

2.4 Sourcing and managing in-line functionality

In legacy networks middleboxes are physically inserted between switches and allow a network to transparently perform operations on packets which are passed from one end host to another. Middleboxes include, for example, NATs, firewalls, caches and load balancers. The set of middleboxes and their possible configurations is large.

When adding a middlebox the network operator will customise the configuration and test it. If set up correctly the middlebox is left alone for an extended period as the labour intensive process does not facilitate easy changes. NFV are treated similarly but with the benefit of automation reducing configuration time and errors, plus facilitating network changes at fast time scales — minutes as opposed to months. SDN applications are another approach where the functionality is added to the controller and utilises switches in the forwarding plane.

To provide a sense of scale Sekar *et al.* (2011) found in one enterprise (with ten's of sites across several geographic regions and serving ~80k users) that it used over 1500 network appliances comprised of ~900 switches and routers and 636 middleboxes [9]. Sherry *et al.* (2012) surveyed 57 enterprise network managers to find that large networks averaged 2850 routers and 1946 middleboxes while small networks averaged 7.3 routers and 10.2 middleboxes [11].

The purpose of network functionality is to improve one or more properties of the network such as latency, resource use or load handling. These services are often stateful in that they manage state required to facilitate the operation of the function. Some implementations manage this state off-line with a third party application.

Middleboxes fall into several categories (not an exhaustive list);

1. Security, such as firewalls and intruder detection systems
2. Network efficiency, such as caches and TCP acceleration
3. Network partitioning, such as network address translators
4. Privacy, such as creating private WANs using encryption.

This list is not intended to be comprehensive, in fact researchers are actively seeking novel ways of providing services to Network Engineers. For example the start-up FastSoft (2006-2012) commercialised Fast TCP technology [41] initially using proprietary middlebox hardware. After acquisition by Akamai in 2012[7] technology improvements subsequently allowed the FastSoft algorithm to be incorporated into the edge routers of Akamai's network[8].

Software algorithms lie at the heart of network functionality. For example, F5 is a proprietary, industrial grade firewall. Its specifications reveal functionality that includes a proxy algorithm, a stateful firewall algorithm, a NAT, potentially a load balancer and deep packet inspection (DPI) algorithms to name a few. We discuss combining network functionality when describing chains of network functions in Section 3.3.6.

2.4.1 SDN control of in-line functionality

As opposed to the manual set-up and on-going servicing of middleboxes in legacy networks, SDN offers the potential for automating these tasks. In the process allowing the network operator to be more efficient, make fewer errors, use resources more efficiently, update software easier and conduct experiments with new network functions in a controlled manner.

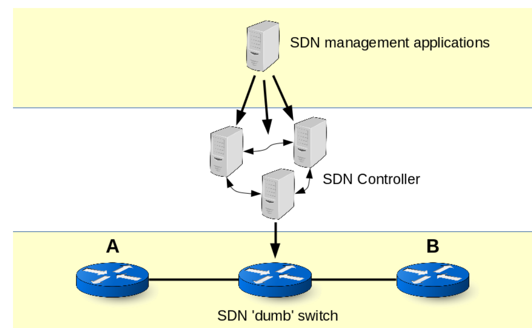


Figure 2.8: SDN network functions are now incorporated into management applications.

The preferred SDN architecture is for network function algorithms to be sitting on the controller as applications as Figure 2.8 shows. The controller in this case does everything except forward packets. Messages are passed to the controller from simple switches and rule updates are returned. The logic being that the control plane is smart and the forwarding plane is dumb and fast. The advantages are many; modularity, clean separation, use of abstraction, removes computation from the forwarding plane, increases control at the control plane, data replication allows swift network recovery, the centralised view allows network properties to be proven, switch rules can be checked for policy violations and more.

There are some challenges to this architecture. For example, while the OpenFlow API allows controller to switch communication, there is no similar API facilitating communication between the controller and network functionality — whether above or below the controller. Other concerns include increased controller workload, controller explosion, controller state convergence and that controller stress slows the networks responsiveness. These are discussed in section 2.7.

ONOS is an example of an industry ready SDN network controller that recognises that controllers are worked hard and offers reliable controller scalability. [42]. The ONOS developers targeted four challenges; high throughput, up to 1M requests per second; low latency,

[7] https://www.akamai.com/us/en/about/news/press/2012-press/akamai-acquires-fastsoft.jsp
[8] https://www.akamai.com/us/en/about/news/press/2013-press/akamai-speeds-downloads-and-online-video-quality.jsp

10-100ms event processing; centralised network state size, **up to 1TB of data**; and high availability, four 9's of reliability (99.99% up-time).

ONOS borrows the concept of Policy Domains from Cohen *et al.* (2013) as a high level abstraction that directs network flows and functionality [26]. Both the abstraction and the implementation of the abstraction are kept in the controller.

Other work that examines the management of in-line functionality includes Jamjoom *et al.* (2014) whom uses a middlepipes paradigm to discuss how this functionality can sit anywhere on the forwarding plane including within the hosts at either end [43]. Gember discusses OpenNF, Stratos and OpenMB [44–46]. OpenNF is a network functions controller for the control plane that requires modifying middleboxes — which increases coupling. Stratos seeks to allow users to use control plane abstractions to specify scaling and composition of middlebox policies which Stratos then uses to create individual middlebox instructions and OpenMB is a control framework for middleboxes that operates alongside the SDN controller and generates individual middlebox instructions. Similar efforts include Merlin which uses constraint solvers in the controller to create middlebox instructions [47]. Slick and Split/Merge are control plane frameworks to enable middlebox scalability [48,49].

In its treatment of in-line functionality, or middleboxes, there exist at least six alternative SDN paradigms. The use of proprietary hardware where SDN is used to route flows to the correct black-box services; SDN controller applications which direct simple switches to enforce in-line functionality; switch waypoints and cloud services where SDN routes flows to third party service providers; language approaches offering abstractions to programmers; choice of state location either within packets, within the forwarding plane or in the control plane; and NFV management. We discuss each of these in turn.

2.4.2 Proprietary Hardware

SDN is proving to be a disruptive technology for vendors of proprietary hardware. Consequently they are trying a variety of strategies to maintain market share. These include some embracing open source and open standards while others have separated their hardware and software products to allow more agile technology improvements. The ideal perhaps is to retain market control by promoting and supplying proprietary solutions and APIs for the controller, switches and in-line services (Figure 2.9). This would allow retention of IP and the ability to create a unique value proposition.

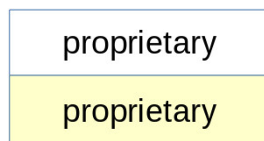


Figure 2.9: Vendors of proprietary functions, advocate SDN with proprietary solutions.

Proprietary hardware features high speeds, often is excess of 10GB/sec; high capacity buffers in RAM that may be in the GB range; high performance ASICs; and high reliability which is a consequence of high quality parts, excellent hardware design and extensive testing. Vendors pursue tough reliability measures, for example, five 9's reliability which gives 99.999% uptime and less than six minutes downtime per year [27].

The negatives include vendor lock-in, lack of ability to customise the product, requiring technicians with vendor specific certification, the difficulty of relocating existing hardware to where it is needed, the cost of provisioning to allow for peak loads and the lead-time required to design a new proprietary system to support a new service [50].

Existing network operators that wish to transition to SDN will often have large existing stocks of proprietary hardware — largely because proprietary hardware has in many ways been a good investment. These stocks of proprietary hardware will need to be utilised to minimise the transitions financial impact and operational risk. Levin *et al.* (2014) discusses ways to transition networks to SDN, managing both cost and risk by initially utilising a minimal number of switches and expanding SDN capability incrementally [51].

2.4.3 SDN controller applications

Placing network functions in the SDN control plane has great appeal. It is tidy and places all the programming in a centralised location which can be easily maintained. However it is also an architectural decision which hard bakes into the network properties that may be better considered in advance than found in hindsight. For example, in Section 2.7.1 we outline how an SDN firewall places the controller on the attack path. In Section 2.7.8 we discuss controller stress and highlight research that shows overloading the controller slows the entire network. We also show that network functionality in the controller requires consistently high levels of controller traffic which contrasts with switches that over time trends to minimal control traffic.

There is no support in existing literature (see Appendix A) that network functionality in the control plane is faster than network functionality in the forwarding plane. In large part this is because they both run on generic hardware. Latency between the switch and controller may be an issue, particularly in wide area networks (WANs), however this is less so when considering data centres where controllers as virtual machines may be located in or near the same host that holds an in-line service. An NFV architecture utilising a single SDN controller co-located with and controlling a single switch is also unlikely to suffer latency issues.

An alternative to placing network functionality on the controller is to make all network functionality in the forwarding plane, stateless. While undoubtedly this would have advantages, many examples seen to date achieve stateless forwarding plane operations by moving the state and algorithm to the control plane, again contributing to the workload of the controller and the control channel.

When the implications of a *logically centralised* controller are considered, controller state is replicated across multiple controllers (ONOS allows for up to 1TB of centralised state). Now state which used to reside in one place on the forwarding plane is now stored and maintained in multiple places in the control plane for the purpose of providing controller redundancy and reliability in the event of controller failure. This also holds benefits for recovery from switch failure as switch state is replicated in or can be recreated by the controller, but the cost seems high and to date we have no research comparing this architecture with alternatives, such as utilising a recovery service (for example, see Sherry *et al.* (2015) [52]).

2.4.4 Switch waypoints and cloud services

By attaching middleboxes to switches, flows can be routed to the switch which proxies for the middlebox's functionality. Gibb *et al.* (2011) describes this as the *waypoint model*, a means of allowing middlebox services to be hosted anywhere in the network [53]. This goes some way towards abstracting the use of middleboxes and allows easier scalability, easy use of the switch liveness protocol, fail-over policies if a middlebox fails and allows traffic to easily bypass middlebox services that are not required.

The waypoint approach also has the flexibility to utilise existing stocks of proprietary

hardware and easily transitions to in-house generic hardware or cloud services when the proprietary equipment is retired.

Sherry *et al.* (2012) adopts this approach suggesting chaining switches connected to cloud based middleboxes [11]. Gibb *et al.* (2012) develops this further and demonstrates outsourcing such services is viable [54]. Simple-fying adopts this waypoint approach, suggesting middleboxes may be connected to a switch to form a policy, these switches may then represent abstract policies that are implemented and used by the controller [55]. Kul Cloud Inc.[9], an SDN vendor based in South Korea, offers an example in production, using this approach in their i-Chain product[10].

2.4.5 Language approaches

There are several advantages for network operators in using domain languages, however, uptake is slowed by the need to learn and gain proficiency. In addition the new language must compete against the inherent investment in existing systems such as Juniper and Cisco.

Most language based approaches to middlebox functionality adopt the strategy of creating abstractions for the programmer. These are then implemented by the controller which maintains state and directs individual switches. Benson *et al.* (2011) discusses CloudNaas, a language framework for programming at the controller level that treats middleboxes and templates of multiple middleboxes as primitives [56]. Other language based approaches include Frenetic, P4 and Pyretic which again seek to create high level abstractions which the controller implements using simple switches [32,57,58]. Song (2013) provides an example of an alternative API that would form a superset of the OpenFlow API and includes protocols to dynamically control programmable network elements in the forwarding plane [40].

2.4.6 Managing state

There are at least three places state may be kept within the forwarding plane; within packet headers by overloading header fields, within switches by extending the OpenFlow specification and switch capability, and within network elements acting as VMs on commodity hardware. A forth option is to place network state in the control plane.

Retaining state within packets

Several researchers suggest overloading packet header fields, utilising the capability to retain meta-data or state. Flowtags is an example seeking to enable identification of flows after they have been modified by middleboxes, potentially useful for traffic management, forensics and diagnostics [59]. Cascone *et al.* (2015) looks to use overloaded header fields to pass state among switches, creating a network of switches forming a mealy machine [60].

Retaining state on switches

Several implementations of generic OpenFlow compliant switches use multiple tables. These may act as a single table or by using goto instructions they may branch from the first table into others. In these implementations table traversal is only permitted one way as cycles may cause problems including acting as a source of hard to debug, loops and black holes for the network.

Bianchi *et al.* (2014) argues that switches should be able to hold and manage internal state. He suggests multiple flow tables may be used for this, by allowing cycles. This permits earlier rules to be rewritten on-the-fly and for the packet to be re-processed based on the rewritten rules [61]. However this adds to switch capability at the cost of greater complexity

[9] See the Kul Kloud website http://www.kulcloud.com/
[10] https://www.youtube.com/watch?v=dRX8ooUAFF8

and does not allay concerns that loops and black holes within switches may be difficult to diagnose.

Retaining state on network functions

This is the approach taken by legacy networks utilising proprietary hardware and by many NFV implementations. State required by a network function to perform its role is stored and managed by the function.

Retaining state on the controller

This is the approach adopted by SDN. However most SDN researchers do not appear to distinguish between local state that is useful to in-line functionality and aggregated switch state used to inform a centralised view that may in turn inform management applications.

2.4.7 NFV management

For a discussion on NFV, see the next section (Section 2.5), however in the interests of continuing the theme of discussing how various network functions are sourced and managed, this section discusses the management of NFV. Some readers may prefer to skip ahead.

NFV uses generic hardware which is slower than specialised proprietary hardware. Generic hardware however is versatile. It can host both firewalls and load balancers at the same time and when it is quiet it can be shut down to save power or have its compute capacity contracted out to process big data for other industries. In contrast proprietary systems offer no flexibility or revenue generating opportunities. Moving algorithms from one generic host to another is simple and fast, as is shutting down old versions of applications and replacing them with new versions in a controlled manner. Hardware fail-over strategies benefit from having cheap and quickly instantiated instances available on other resources. Where peak load exceeds in house capacity, generic hardware or network services are available from cloud service providers at costs charged out over short time frames, this is referred to as *cloud bursting* [62].

Fault tolerance in proprietary systems is based on careful engineering and deploying backup hardware. NFV does not typically have control over its environment and it may be expected that commodity hardware will fail more frequently than proprietary hardware. Sherry *et al.* (2015) observes that resorting to backup devices or NFV instances in any case provides only a weak form of recovery for stateful network functions [52]. Simple master-slave data stores provide stronger recovery but may still exhibit problems when the states diverge. Proposed is an interesting variant comprised of a backup network function with a recent snapshot of the data and a log of transactions since the snapshot. Implementing this in an NFV function may be considerably cheaper and more effective than existing proprietary solutions.

To spawn a virtual network function, the controller requires only knowledge of its type, any configuration required and its location as defined by the IP addresses of the two hosts it is connected to. This is similar to utilising proprietary middleboxes which in addition to a type inherent in its hardware, a location dictated by its wiring and a set of rules, may also have a proprietary language.

The controller also manages hardware resources and determines which hosts the NFVs are spawned on (or it may utilise existing proprietary hardware). The controller may also use strategies, for example, to minimise latency by keeping NFVs within the same datacentre.

2.5 Network Functions Virtualisation

NFV is an architecture that is characterised by the separation of software from hardware. This allows middleboxes to become reusable software elements on generic hardware [10].

It should be apparent that NFV technologies are independent of SDN technologies. The benefit of NFV to both legacy and SDN networks includes its inherent plug 'n play nature which allows code re-use, use of commodity hardware, greater peak load flexibility and the potential for automated management of network functions, an example of which might be provided by an SDN management application.

The separation of software from hardware allows both technologies to innovate at different rates. Typically software will innovate faster than hardware with its longer production cycles and slower pay-back period. Proprietary solutions, for example, typically package hardware and software as an indivisible product, slowing innovation to the pace of hardware innovation.

Commoditising services using NFV encourages vendors to compete on cost rather than added value features couched in marketing speak directed at non-technical decision makers. The expectation is that over time this will drive down the cost of both hardware and software, reducing capital investment.

2.5.1 Properties of NFV

NFV research is largely focused on replacing existing middlebox functionality. As such NFV display interesting properties that exist in middleboxes. They are modular in that code is organised into a single self-sufficient element, they provide a layer of abstraction - knowledge of the internal code is not needed to gain the service provided, they are encapsulated in that other code cannot directly access internal data and the functionality is decoupled from other network functions and the controller.

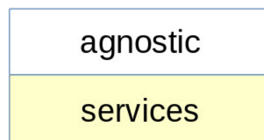


Figure 2.10: The NFV view of SDN, provide network functions that are agnostic to control plane solutions.

NFV brings further advantages, unavailable to middlebox hardware. The code is reusable and upgrades are easily incorporated into the network in a controlled manner. The service provided may be rapidly scaled up or down by instantiating or decommissioning instances, coping with peak demands and enabling power savings by aggregating flows in order to run the minimum number of instances for the load. Hardware scaling capacity may be provided at peak times by cloud providers offering IaaS. Software scaling capacity may be provided by cloud providers offering SaaS. NFV are controller agnostic, meaning they also decoupled from the controller whether human, SDN or proprietary (Figure 2.10).

Making decisions locally, provides fast responses as the algorithm has immediate access to both the packets in a flow and local state created from observations of previous flows. Being on the forwarding path also facilitates other processing related to protocols, such as recognising the end of a TCP session.

2.5.2 Virtualisation options

Virtualisation technologies are gaining in popularity and usability. Products such as VMware and VirtualBox are widely used to create virtual machines (VMs) and may be booted with an operating system (OS) that is different to the one on the host computer [63].

Upcoming technologies include Docker which uses containerisation, similar to VMs but using the host OS whilst still maintaining secure separation between applications in containers. Generic network hosts, with multiple cores and large RAM, can host tens of VMs (for example, Xen and ClickOS discussed earlier). The same hardware can potentially be host to hundreds of containers [64].

2.5.3 Speed of virtual in-line services

The ClickOS VM in testing nine implementations of middleboxes[11], achieved line rate processing near 10GB/s [24]. This was using a low end server with four cores at 3.1GHz and 16GB of DDR-3-ECC RAM.

Proprietary hardware solutions are optimised for speed, capacity and reliability. They consequently, box for box, will perform significantly better than solutions running on generic software. However greater speeds are possible in generic hardware using parallelism, a technique used widely in networking research, for example, to allow speeds near 2TB/s on fibre, utilising twenty one 100GB/s fibre cables [65]. Of course this is possible with both proprietary hardware and NFV on generic hardware, the point being that NFV holds large cost advantages — the financial cost of running parrallel algorithms on generic hardware is less than the cost of purchasing the equivalent capacity in proprietary hardware. Over time it is reasonable to expect that improved generic hardware and software will allow virtual machines to achieve greater speeds with less parallelism.

2.5.4 Testing NFV equivalence

Adopting NFV from smaller companies involves risks such as; incorrect application behaviour, poor reliability, back-doors, security concerns and liability insurance when services go wrong. It is surprising that for well understood concepts like common network functions that there has been no research into a tool that might answer questions around behaviour equivalence between implementations. Proving A is equivalent to B may lead to better purchasing decisions. We explore this further in section 2.6.

2.5.5 Chains of network functionality

Typically a network operator will arrange a sequence of middleboxes to perform tasks at a given point in the network. These are referred to as Service Chains. For example the entry to a domain may have a NAT to ensure internal IP addresses are logically separated from external IP addresses. This also creates a single entry-point occupied by the NAT that may also be monitored by other network functions such as a firewall algorithm.

An example service chain is provided in Figure 2.11. For a user to set up a session with Host B or C, it traverses three services. The firewall ensures the external user is not known to be dangerous. The NAT ensures the user can only access the host via a service chain that includes this NAT. The load balancer ensures the user is serviced by the same end host so that any session state created by the connection is always available.

[11] These were; Wire (for baseline performance), Ether mirror, IP Router, firewall, NAT, BRAS, IDS, load balancer and flow monitor.

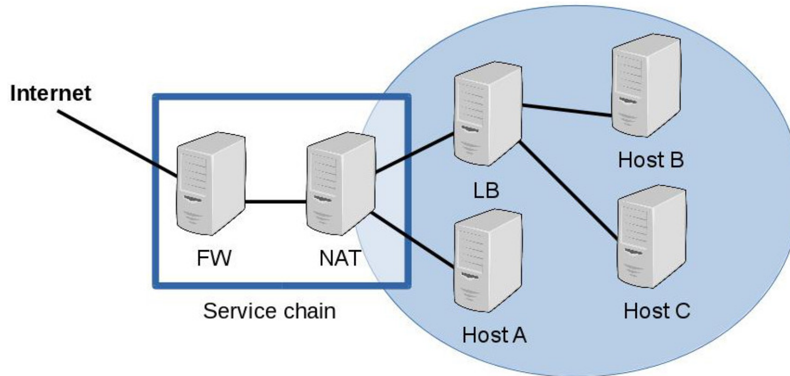


Figure 2.11: A service chain of middlebox algorithms

Monsanto *et al.* (2013) discusses several issues relating to composition of components in a service chain [58]. These may be mitigated by formalising what is required of the service and using a heuristic to determine the best place to locate the service within a network [66]. We discuss this shortly in Section 3.3.6

There are several implementation options, for example, chaining middleboxes to a single switch which cycles packets through remote services [55, 67], a variation on waypoints discussed earlier. Another uses meta-data like Network Service Headers to encapsulate the original packet [68] or packet MAC addresses for in-chain addressing [69]. OpenSCaaS uses both a controller and MAC addresses. [70].

2.5.6 Aggregating functionality on hardware

One advantage of cohabiting virtual network functions on the same hardware is the potential to improve performance. For example, reducing packet input buffers from one per middlebox to one per chain at the cost of compositional flexibility (the middleboxes are now dependent on the input buffer) [10].

One disadvantage is the increased management complexity and another is that chains of virtual network functions form an indivisible unit, a larger building block which contributes to the bin-packing problem, resulting in under-utilised resources [71].

Lee *et al.* (2011) initially proposed stacking VMs in commodity hardware in his paper *No More Middlebox*, describing removing dedicated hardware middleboxes and replacing them with VMs [72]. Anderson *et al.* (2012) brings xOMB which ties middlebox VMs to servers, so the servers become the primary abstraction [73]. Mekky *et al.* (2014) has a similar approach, tying VMs to Open vSwitches set up by the controller [74].

2.6 Problems with comparing implementations

Concerns around intellectual property has led many vendors to ship their products as black-box implementations. In most cases customers accept this, provided the implementation *behaves* as expected [12]. When faced with a choice of implementations comparing their behaviours through the thick veil of marketing material and non-technical sales advisers is difficult and not a robust process.

[12]Behaviours captures the concept of observable behaviours as seen by the environment, however it is accepted that not all behaviours can be tested using formal methods, such as algorithm speed or processing capacity. We leave these parameters to other tools.

1	B	the domain of all network function behaviours
2	$m \in pow(B)$	a model is a set of behaviours
3	$i \in pow(B)$	an implementation is a set of behaviours

Table 2.1: Algebra definitions

This leads to a potentially novel use of MBT as a tool for comparison between implementations. Many existing middleboxes have been well studied and behaviours detailed, MBT allows modelling these behaviours and may then use this model to test and compare black-box implementation behaviours. We can express this formally.

$$equivalent : (pow(B), pow(B), pow(B)) \rightarrow \{true, false\}$$

$$equivalent(m, i_1, i_2) = (m \subseteq i_1 \wedge \\ m \subseteq i_2) \vee \\ m / i_1 = m / i_2)$$

The intuition (see Figure 2.12) is that two implementations are equivalent if they share the same subset of behaviours captured in the model. This definition is limited to the extent the model captures the behaviours of interest.

In practice we expect to partition network functions into known algorithms, for example, firewalls, load balancers, NATs, that each have a finite set of behaviours. Capturing these function behaviours, while not a trivial exercise, is achievable. More difficult is capturing the wider set of behaviours in the environment, however once a new behaviour is identified (for example, an attack attempt), it can also be modelled and used to test implementations against each other and the new model.

2.7 Problems with SDN applications

Earlier the range of network applications was described as divided into stateless (no dynamic state) and stateful (manages own dynamic state).

SDN explores a third type of network function where the state is managed by a third party. This is driven by the desire to utilise the OpenFlow architecture where a remote SDN controller communicates with and instructs the dumb OpenFlow switch. This requires that local knowledge which is present in the switch, be moved to the controller to recreate the local knowledge in the centralised view. This allows a centralised middlebox algorithm to form a decision. This is cumbersome and introduces problems and limitations not addressed in current SDN research. For example, a legacy in-line firewall operating on passing packets

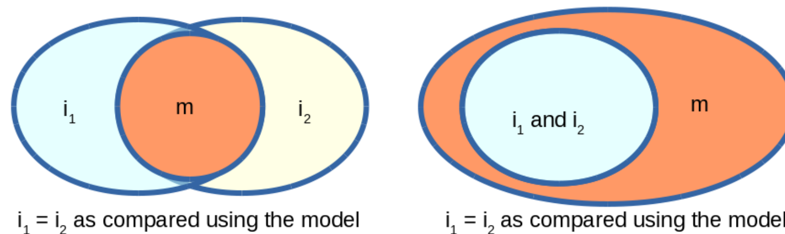


Figure 2.12: Implementations (i_1, i_2) are equivalent where they share the same set of modelled behaviours.

may hold state to recognise returning packets. Such a firewall gains no benefit from a centralised view. Moving the firewall's algorithm and state to a third party creates a range of problems which we outline shortly. A counter example that may be considered is a security application that identifies abnormal flows in the network — such a service may use existing centralised knowledge and does not require further local knowledge from the forwarding plane.

This section outlines a range of interesting problems caused or aggravated by SDN's remote controller architecture being applied to stateful network functions.

2.7.1 Placing the controller on the attack path

A basic tenet of network security is defence in depth — utilising perimeter security and demilitarised zones (DMZ) between the Internet and the secure domain, facilitated by firewall appliances that are designed to resist attacks. In comparison a commodity OpenFlow switch is designed to cost effectively forward packets, including sending requests to the controller for every new flow. OpenFlow switches may therefore quickly collapse under DDoS (distributed denial of service) attack, but not before flooding the controller (also not designed to withstand attacks) and potentially causing it to crash as well. Regaining control of the switch or the controller in the face of a DDoS attack may be difficult while the attack is in progress. Should this switch be a gateway, the network will now be isolated. Should the controller collapse, control of the network may be lost for the duration of the attack.

2.7.2 Increasing control channel bandwidth

A stateful network function in legacy systems, once set up, requires no further communication with the SDN Controller or network operator in order to perform its tasks. In contrast, a network function as a controller application holds local knowledge as state in the control plane. This means switch-to-controller messages for every flow to set and compare against controller state. In normal operation this creates consistent, significant, two-way control traffic, that scales up linearly with the number of flows. This contrasts with SDN switches forming the other half of the network, which trend toward minimal control traffic.

Adding to the bandwidth of switch to controller traffic is the use of TCP protocol [75] and potentially encryption to ensure controller communication security.

2.7.3 Increasing SDN controller workload

SDN controller workload and state is minimised where a network has stateful network functions being managed in the forwarding plane by the controller as place and forget functionality. The SDN alternative is hosting the functionality on the controller with its finite CPU and RAM capacity. Or as a remote third party control application, with control traffic routed via the controller.

A variety of strategies may be adopted to preserve controller responsiveness, often adding to controller complexity, such as partitioning a network each with its own controller or parallelism which allows multiple controllers to share the workload, however controller parallelism is a more complex strategy than network function parallelism in the forwarding plane. Another alternative is to increase controller hardware specifications, but this may add to infrastructure procurement complexity and reduce the portability of controller applications.

2.7.4 Many flow rules slow the switch

Network functions form nearly half the appliances in legacy networks and replacing them with commodity switches will nearly double the number of switches managed by controllers. The alternative is to increase (double?) the number of rules in existing switches, however Kuzniar *et al.* (2015) found that switches are slowed by large rule sets [34]. This may be mitigated by flow rule optimisation across switches, but again at the cost of increasing controller workload and complexity.

2.7.5 Pushing local state across multiple SDN controllers

A centralised data store, in order to be reliable, must replicate data across instances. This will typically require at least three SDN Controllers to have copies of the state. If one fails, the remaining two may continue and will still offer resilience in the face of a second failure. Local knowledge for an SDN controller-network-function therefore involves the computational and storage resources of at least three SDN Controllers.

A valid observation is the centralised state allows recovery from network function failure, however this might be better handled by a distinct service as described by Sherry *et al.* (2015) in her paper *Rollback Recovery for Middleboxes* [52]. This provides network function failure recovery without burdening the controller.

2.7.6 Inconsistent SDN Controller state

Typically SDN Controllers adopt *always available, eventually consistent* as a distributed data store model [29, 38]. *Eventually consistent* means that the global view shared amongst SDN Controllers is in a perpetual state of nearly consistent. Decisions based on an inconsistent centralised view may lead to problems such as routing loops, partitioning the network or poor security decisions. We have termed this **the first layer of state divergence**.

Of course this is not unique to SDN controllers and other data stores have found ways of addressing the inconsistent view problem. Typically two strategies are involved; First, reduce the convergence time of the views by minimising the size of the data stored and control the rate at which updates are applied. Second is to add more complexity to applications that read and write to the data store, to create the appearance of a consistent view.

Current SDN already uses the first strategy, SDN minimises the state held by selecting a subset of local knowledge from each switch — specifically the flow and packet byte counts for each table entry and the time the last packet matched the entry [17,76]. SDN also controls the rate at which this switch data is gathered to form the global view (switch data is pulled by the controller). For the second strategy SDN relies on controller applications to be complex enough to gracefully handle inconsistent views, however, how or if this is done may not be easy for an operator to assess. When SDN middlebox dogma is considered where all network functions must be controller applications, SDN introduces state bloat and uncontrolled state updates (local knowledge is pushed to the controller). Consequently SDN casts aside middleboxes for controller applications that both increase the prevalence of inconsistent views and it is hoped will gracefully handle those inconsistent controller views.

2.7.7 Pushing local state versus pulling switch statistics

Legacy networks have operations occurring in two time frames, human operator speeds which apply to configuration changes and packet flow speeds in the forwarding plane which are orders of magnitude faster.

SDN introduces a third time frame, centralised view updating which is completed by the SDN controller periodically updating its centralised view by requesting (pulling) switch statistics from commodity switches. This is completed at time frames convenient to the controller and sits between human operator speed and packet flow speed.

OpenFlow switches interrupt controller activities at packet flow rates, requiring fast response to new flows on-the-fly. This involves sending (pushing) local information to the controller for processing and typically trends to minimal controller contact as many flows are long lived and satisfied by a single controller request.

A switch performing stateful network functionality also pushes local knowledge in order to create controller state of benefit to the switch (the switch will access this state regularly). The resulting decisions are then returned at packet flow rates. This pushed centralised local state is distinct from the centralised view created from switch statistics in that switch statistics are pulled at the controllers discretion whereas switch requests must be treated immediately. The risk is, that the controller may be obliged to prioritise network functionality and saving local state at packet flow rates, over updating its centralised view.

2.7.8 SDN Controller stress — may manifest slow network behaviour

Stateful middlebox applications on the controller require sending the controller significantly more traffic than is the case with stateful middleboxes in the forwarding plane, which after configuration require no communication in normal operations. Increasing the workload of controllers and the controller channel, would be counter to concerns expressed regarding controller stress. For example; Phemius and Bouet (2013) examine the case when the switch-controller link is stressed and conclude stressing the controller may have a significant negative impact on network performance [39]; Devoflow identifies this problem and offers an example solution which seeks to minimise communication, by only consulting the controller when elephant flows are identified [77].

For example, the failure of a SDN Controller causes stress in a logically centralised controller. This requires the remaining SDN Controllers to redistribute state to ensure copies are maintained across at least three active SDN Controllers. State redistribution is not a trivial exercise and it may impact on network performance obliging ordinary SDN Controller traffic to be queued. Note that in addition to controller failure, scaling the number of controllers up or down to handle changes in controller workload incurs the same activity.

2.7.9 Convergence of the forwarding plane with the control plane

A similar concern to inconsistent state across controllers is the inconsistent view between controller(s) and the forwarding plane. We have termed this **the second layer of state divergence**.

Ignoring latency between controller and switch which is mostly a concern for WANs, there are additional delays between the controller determining its preferred network graph, communicating flow table updates to switches and the switches implementing those instructions. Kuzniar *et al.* (2015) found that the forwarding plane may fall behind the control plane by up to 400ms and rule updates may be applied in a different order to that assumed by the control plane despite use of the barrier command intended to enforce ordering [34].

2.7.10 Convergence of the network function with end hosts

Network functions are intended to be transparent to end hosts. This typically requires that the network function and end hosts both hold the same view of the communication state

between end hosts. Where this is important and fails to occur we have termed this **the third layer of SDN state divergence**.

For example a firewall should converge rapidly with communicating end hosts. In particular if both hosts believe a communication has ended, the firewall should reflect this. Where a firewall uses flow rule timeouts to achieve convergence, by design this means network function state will lag behind end host state, leaving vulnerabilities. Where a firewall fails to act on a TCP sessions FIN handshake between end hosts signalling the session is closed, again it leaves vulnerabilities.

2.8 Firewalls — an example network function in SDN

The firewall was chosen for this research for a variety of reasons. Time constraints would preclude a thorough analysis of multiple network functions; there appears to be a lack of in depth knowledge of firewall types within SDN commentators; and it was felt interesting properties may be uncovered by comparing inline stateful firewalls with SDN controller based stateful or stateless firewalls. Potentially by analysing this one network function we could prove the method used to find an example of the third layer of state divergence, sufficient to warrent further research on other network functions at a later date.

The phrase *Firewall* has many meanings in networking leading to confusion amongst commentators. A stateless firewall (a switch packet filter) is not equivalent to a stateful firewall (with dynamic state) and neither is equivalent to an IDS (Intrusion Detection System) using switches to isolate intruders. These three examples exhibit different behaviours and solve different security problems. All three are confusingly referred to as Firewalls.

Firewalls may also be one or more service chains of network functionality designed to offer *Unified Threat Management*. One of the functions will be the firewall algorithm. Others will include; NATS, proxies, load balancer, caches, DPI, encryption and more.

As service chains and indeed network paths are pipelines, the firewall algorithm can be considered in isolation, reacting solely to the packets it observes, according the algorithm and any configuration rules it has. The firewall algorithm is the network function of interest in this research. In addition it has been rigorously studied and detailed in RFCs, by NIST and in numerous textbooks [15, 78–83]. The remaining network functions in a *Firewall* are left for future work.

The firewall algorithm operates at line speed, this is typically defined as 10GB/s[13], however there also exist a range of commercially available firewall appliances providing speeds from 100MB/s. Determining a packet's status {allow, drop} is achieved by analysing the packet's header fields and comparing them to an ACL (access control list) and any dynamic firewall state. Outside the scope of a firewall algorithm is inspection of the data. This is deep packet inspection (DPI) which is another algorithm that incurs latency delays from decryption and message assembly, which may be considered an acceptable trade-off in certain use-cases.

Placing firewall functionality in the control plane and using switches in the forwarding plane as the first line of defence, intuitively does not seem wise given that switches are not designed to withstand attack. They instead efficiently pass attack traffic to the controller for processing. None-the-less there are increasing numbers of research papers proposing or using such an architecture [84–91]. The Floodlight controller firewall module is an implemented example of a stateless firewall that passes rules to switches based on an ACL list.

[13]This definition appears to be a convention repeated in academic and industry articles. To place it in context, Victoria University of Wellington with a student population of ~17,000, has a 10GB/s pipe to the Internet.

The Ryu controller firewall is also an example of a stateless firewall[14]. Neither controller appears to have a stateful firewall application openly available.

There are other concerns, for example, keeping dynamic state with a remote third party requires referring to that third party to make flow decisions based on that state. Yoon *et al.* (2015) experimented with an SDN stateful firewall built on the Floodlight controller, finding it added 31.595ms latency to a legitimate FTP connection [92]. They also expressed concern over the volume of control messaging required. Shah (2015) experiments with a stateful firewall application for Ryu and found it gave only 10-100MB/s speeds. This was attributed in part to the use of TCP protocols between the switch and the controller [75].

Complex protocols are another limitation of remote algorithms. For example, the end of a TCP session cannot be pre-determined in advance nor accurately identified by the switch. OpenFlow 1.5[15] allows matching against TCP flags such as FIN, SYN and ACK, however the final handshake does not end on a FIN packet, it ends on the responding ACK packet. Sending every ACK packet to the controller is not viable as there is a near one to one match between data packets and $ACKs$. Reacting to a FIN via the controller may also be difficult, there are two in the handshake and the responding $ACKs$ may have passed before the controller response can be implemented. For example, Kuzniar *et al.* (2015) reports measuring controller to switch delays of up to 400ms [34]. Delaying the FIN until the switch is ready for the ACK may be viable but increases use of buffer resources, contributing to the switches vulnerability to attack and adds to latency — however adding to latency may be mute given this is the end of the flow. Rule time outs are another alternative but may lead to a situation where both hosts believe the connection is finished but the firewall remains open.

The SDN Floodlight Controller project has a Firewall module as a controller application[16]. It is a stateless firewall with the ACL rules held by the control module. On a switch request the ACL is referenced and an appropriate rule is pushed to the switch. The project discusses as a limitation that the module does not delete flow rules on a switch and that flow rules are expected to time out. Other limitations include a lack of information from the switch to facilitate a delete decision, lack of dynamic firewall behaviour, threat to the controller from a distributed DoS attack (one that uses many IP addresses) and state divergence between the controller and forwarding plane where the controller firewall algorithm believes the connection is closed while the firewall switch is doing otherwise.

At present there are few companies overtly offering virtualised firewalls. When cloud service offerings are examined for references to security, they tend to be vague.

Cisco is one exception, they reference their Virtual Security Gateway for VMs that appears to be modelled from FlowGuard [85], although without more information the comparison is tentative. They also offered the *ASA 1000v Cloud Firewall*[17] which is a NFV utilising VMware with a 1.2GB/s throughput. Unfortunately support for it has been discontinued and it appears to have not been replaced with all links to virtual security appliances pointing to the discontinued offering.

2.8.1 The Firewall Algorithm

A firewall algorithm offers edge security between two domains, occupying a choke-point controlling traffic between the two and treating one side in a different manner to the other

[14]Floodlight firewall url: http://www.projectfloodlight.org/
Ryu firewall url: https://osrg.github.io/ryu/

[15]Note that OpenFlow 1.3 and 1.4 is currently being implemented by switch providers. Switches implementing OpenFlow 1.5 are not yet available.

[16]https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Firewall

[17]http://www.cisco.com/c/en/us/products/collateral/security/asa-1000v-cloud-firewall/data_sheet_c78-687960.html

by dynamically collecting state from outgoing flows, for use on incoming flows. This state is used to allow two-way communication across the border only if it is initiated by the internal host. This describes a stateful layer 4 firewall holding dynamic state, as opposed to a stateless firewall which is trivially implemented using a switch as a packet filter applying drop actions to specific (or ranges of) flows.

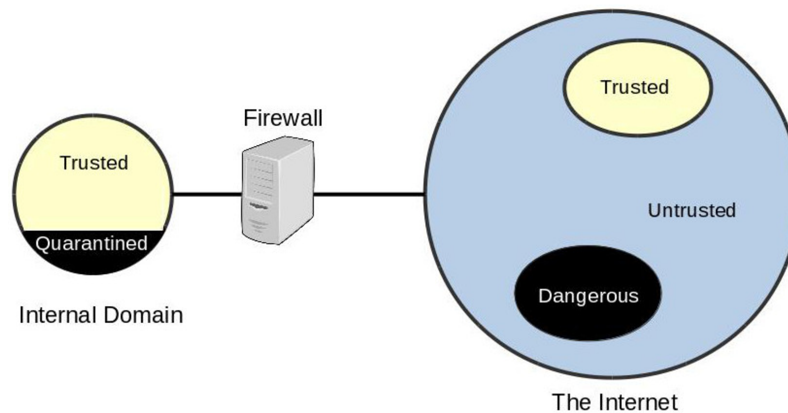


Figure 2.13: Domains of interest to a firewall

Firewalls use an ACL (access control list) to categorise hosts into a proscribed list of dangerous hosts and a permitted list of trusted hosts. ACLs are portrayed in Figure 2.13, ACLs cannot detail every host in the Internet due to both hardware constraints and the underlying mutability of the Internet. In a stateless firewall packets that are neither trusted or dangerous are typically treated as dangerous and dropped. In contrast a stateful firewall may treat unspecified hosts as a third category labelled untrusted and allow two-way communication with them if an internal host initiates the exchange.

Interaction	State and Action
$A \parallel \leftarrow B$	no relevant state held, packets are dropped by default
$A \rightarrow B$	is allowed, the first packet adds FW state
$A \leftarrow B$	is allowed, the flow finishing removes FW state
$A \parallel \leftarrow B$	no relevant state held, packets are dropped by default

Table 2.2: Firewall in the forwarding plane

To describe the behaviour of a stateful firewall we refer to table 2.2 in conjunction with (the earlier) Figure 2.3. The external host *www.catlovers.com* which is neither proscribed nor permitted by network policies finds the firewall will only recognise its incoming flow if communication has been instigated by an internal host. Figure 2.3 shows an internal switch labelled **A** and an external switch leading to *www.catlovers.com* labelled **B**. In-between is a middlebox firewall that was placed and has ACL rules set by either a human or automata. Table 2.2 first shows the results of an attempt by *www.catlovers.com* to contact an internal host ($\mathbf{A} \parallel \leftarrow \mathbf{B}$), $\parallel$ indicates the packet is dropped. If the internal host initiates a communication, the firewall on seeing the outgoing packets ($\mathbf{A} \rightarrow \mathbf{B}$) retains state in order to recognise a reply. The reply ($\mathbf{A} \leftarrow \mathbf{B}$) is then allowed to pass and state is removed once the flow has completed — indicated by observing the hosts exchanging and acknowledging FIN packets. Any future attempt by *www.catlovers.com* to contact an internal host ($\mathbf{A} \parallel \leftarrow \mathbf{B}$) is once again dropped.

A fundamental assumption of the stateful firewall is that it maintains the same connection state as the end hosts [15, 78–83]. In particular, if the end hosts believe the session is

finished, so should the firewall. The use of timeout values are common in firewalls, but are insufficient and may result in creating a security vulnerability seen in cellular networks [93]. Not closing the firewall after seeing the finishing TCP handshake leaves an open door in the firewall after the flow is finished that an attacker can maintain in a perpetually open state — a prerequisite, for example, to explore and steal data or conduct a battery draining attack [93].

2.9 SDN research into network functionality

SDN research papers typically endeavour to rely on commodity switches in the forwarding plane. For example, FlowGuard uses ACLs which can be trivially implemented in flow rules on switches [85]. The benefit is that local knowledge is completely ignored which greatly simplifies the controller application and communication needs with the forwarding plane. However such an approach limits its usability. In contrast, FlowGuard's use of global knowledge to ensure new switch rules do not circumvent global security policies is a good example of creating value from the centralised view.

OpenState suggests local state be stored in packet headers and switches be used as nodes in a mealy machine [60]. Another suggestion is that commodity switches be extended to allow flow tables to act as state machines [61]. Hassas & Ganjali (2012) also recognises local state and proposes partitioning the network to move controllers closer to the forwarding plane [94]. Mekky *et al.* (2014) uses flow tables to store local state which require vendor extensions to allow over-writing existing flow rules, but allows the potential for flow tables to become finite state machines [74].

These sample papers attempt to work around the SDN ideal that all logic must be purged from the forwarding plane, by either ignoring local knowledge, moving the remote decision maker closer or making the SDN switch more than a simple forwarding element. This is in contrast to other views (such as NFV) that support a forwarding plane of simple and fast forwarding elements and network functionality on high throughput, highly portable and scalable virtual machines running on commodity hardware [33].

2.10 Summary

This chapter has covered a lot of ground. It has described existing legacy networks and perceived problems, Software Defined Networks as a potential solution, middleboxes and their virtualised network function replacements and the preferred SDN approach to network functions that causes a number of unresolved issues for SDN networks. Finally SDN firewalls have been discussed, including a brief discussion on existing research.

The next chapter describes formal methods and their application to network functions.

Chapter 3

Formal Methods

Software Defined Networking (SDN) looks to move in-line network functionality from the forwarding plane to remote applications that utilise a controller to communicate instructions to switches. By applying formal methods and model-based testing (MBT) it is intended that interesting problems seen in this architecture (for examples see section 2.7) can be explored.

This exploration involves a novel use of MBT to enable the comparison of black-box implementations (in-line network functions and SDN network functions), limited only by the detail included in the formal model (see Section 2.6).

> "[Coupling is] the measure of the strength of association established by a connection of one module to another. Strong coupling complicates a system, since a module is harder to understand, change or correct by itself if it is highly interrelated by other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules."
>
> Stevens, Wayne, Myers, Glenford & Constantine (1974) Structured Design [95]

> "Modules with strong cohesion, in particular with functional cohesion, are easier to maintain, and furthermore, they greatly improve the possibility for reuse. A module has strong cohesion if it represents exactly one task of the problem domain, and all its elements contribute to this single task."
>
> Eder, Kappel & Schrefl (1994) Coupling and Cohesion in Object-Oriented Systems [96]

Modular code may lend itself to formal methods as it can typically be analysed as a discrete system with well defined behaviours and interfaces. Of interest to networking is that network functions in legacy networks are modular (albeit often proprietary systems utilising software and hardware) with low coupling and high cohesion. This might be compared with the stronger coupling required when network functions are incorporated with and work through a controller remote from the forwarding plane. In practical terms an in-line firewall may be tested against a model in isolation, it is functionally complete. An SDN firewall requires testing three components; the firewall application, the SDN controller that directs and responds to the switch and the in-line SDN switch.

From here three network functions are discussed to highlight common properties leading to creating a formal description of a generic network function. The modelling tool Rodin and the Event-B modelling language are both briefly introduced and are followed by a discussion on model-based testing. This chapter finishes with a review examining experiences in industry with model-based testing and the unresolved problems holding back its widespread adoption.

3.1 Overview

To borrow from Baier & Katoen (2008) formal methods can be considered as "the applied mathematics for modelling and analysing ICT systems" [97]. The aim is to establish system correctness with mathematical rigour while a further benefit is that it allows automated model checking which is the exploration of state space to check if properties are violated.

Hand-crafted code is also a mathematical model, albeit one created with less mathematical rigour. It is typically built from the bottom up and tested using methods such as unit-tests, code reviews and integration tests. These are human driven test strategies that suffer from human and commercial frailties, for example, maintaining a library of tests in the face of code change can be expensive in developer time with limited benefit to profits. It may also be commercially prudent to release code then triage the inevitable bugs reported by users according to available developer time and the bugs impact to business values (profit, customer responsiveness, etc.)[1].

Model-Driven Development (MDD) goes further, it refines a model and its proofs until it is capable of being compiled into code [98–100]. Because of the mathematical rigour involved in creating the model, the resulting code will be faultless, with the caveat that the model is correct.

An example of modelling and MDD is the control software for the driverless trains on the Paris Metro line 14. The first driver-less trains to run on a major metropolitan line [101]. Passenger safety was considered a key concern and properties such as no train collisions and passenger doors closing safely before the train moves off formed part of the invariants. The B-method, a formal modelling language, was used to create the model representing the train control system and the environment it was to operate in, the Paris Metro line 14. A top down, iterative process was used which identified high level abstractions and proved properties about the interaction of those abstractions. Once these properties were satisfied, the abstractions were progressively refined, a process called stepwise refinement [102], and new properties proven in a process that ultimately results in a mathematical model that was then compiled into production code. The Paris Metro Line 14 went live in 1998 and carried 3.5 million passengers in that year.

That modelling has not become the de-facto standard for creating software speaks to both the difficulty of training engineers to create rigorous mathematical models and the relative ease of training developers in traditional coding. Consequently human developers are cheaper and easier to source for projects. The fact that they produce flawed code is then managed.

Using formal methods has pitfalls. It brings up-front costs to projects in order to mitigate the expense of fixing flawed architecture and code later. In addition creating a formal model involves modelling the environment the software will interact with, raising at least two concerns; first that the environment is understood well enough to create a reliable model, second that any environmental event outside those modelled will result in non-deterministic behaviour. For example, an interesting modelling exercise is *the cars on the bridge* exercise which produces a model for controlling traffic lights on a bridge safely [103]. However should a driver choose to ignore the traffic lights, the model can no longer guarantee safety.

A criticism of formal modelling is that the modelling approach explores a relatively static domain. It contrasts with the Agile methodology used in the programming industry that

[1]The Novopay payroll system for the New Zealand Government's Ministry of Education (2002-ongoing) is a recent example of commercial and political pressures influencing a NZ$182 million contract to supply teacher payroll software. Rectifying software failures has cost an additional NZ$45 million to date. Many articles to choose from, this one discusses the political decision to go live, ignoring the poor results of a trial. http://www.nzherald.co.nz/nz/news/article.cfm?c_id=1&objectid=10855399

regularly presents interim products to the client who is learning and developing their understanding of what is possible. In the process potentially changing the projects requirements. The process of modelling to derive production code may require a domain and product that are already well understood as the client will receive the end result in a single release and client learning opportunities during the modelling process may be difficult to capture. In contrast, using modelling to explore architecture and black-box behaviour is significantly less time consuming than creating models that compile to code and the process may be fast and flexible enough to be used alongside an Agile environment.

Model-based testing (MBT) is a lesser known field which utilises the idea that modelling black-box behaviour may be sufficient for some purposes, for example, to test proprietary black-box implementations. In essence a formal model is created that captures the observable behaviours of a black-box implementation. This formal model can then be utilised to generate a brute force state space exploration — that is, it can generate paths that test for example, every node, vertex, variable or branching decision involved in creating the observable behaviours given an input. Thousands of paths may be generated to cover all possibilities, with each path specifying both the input and expected output which can be compared with the output from a black-box implementation given the exact same input. Discrepancies between the two may then be investigated further. Given the range of potential inputs, for example the range between 0.0 and 1.0 is infinite, strategies such as checking input boundaries may be adopted in order to limit the state space explored to a manageable size.

Industry and researchers report a range of benefits from using formal methods and MBT. In 2001 a tool called the Test Model Toolkit was being used by over twenty teams at Microsoft. It took a week to generate a set of test cases that would take eight weeks by hand and code coverage increased by 50% [104]. Pretschner *et al.* (2005) found that both hand crafted and MBT tests found similar numbers of programming bugs while MBT found more requirements errors [105]. Programming bugs equate to *building the thing right* while requirements errors question whether you are *building the right thing*. Despite the benefits however it is concerning that MBT is not 'sticking' to industry. It is an open question whether this technology can be generally applied in industry to validate implementations — a problem for which it appears neither industry nor researchers yet have an answer.

3.2 Three common network functions

The phrase 'network functions' describes any implementation of network functionality, including middleboxes, switches and routers. However the focus of this research is on network functionality other than routing functionality, meaning the use of the phrase 'network function' should be assumed to exclude switches and routers.

Network functions cover a wide range of tasks each aiming to improve one or more properties of the network, such as traffic throughput, latency, service capacity, security and more. For example, a Network Address Translator (NAT) is typically used to partition and isolate a domain within a network and creates a single ingress path to that domain, in the process increasing a networks available address space. A load balancer allows increasing service capacity through parallelism. A firewall increases security by preventing domain mapping and resisting DDoS attacks — whilst still allowing internal hosts to have access to Internet services. Network functions provide network functionality beyond that required for routing decisions.

Two important terms arise when discussing network functions, *stateful* and *stateless* which confusingly do not refer to the presence of state. All network functions have state important

to their functionality, these phrases refer to who manages it;

- A stateless network function relies on a third party to manage its state. Examples of a third party include a network administrator or controllers.
- A stateful network function dynamically self-manages state at or near line speed.

We next explore three common examples of stateless and stateful network functionality to find common properties with the goal of developing a formal model of a generic network function in Section 3.3.

3.2.1 NAT

Network Address Translators (NAT) were developed in part to allow reuse of limited numbers of public IP addresses[2]. Today they are valued also for their ability to partition a network into isolated domains, each with only one ingress path. An important property of a NAT is that it provides its service in a manner transparent to Internet traffic, in that end hosts on either side of the NAT are oblivious to its presence.

Network partitioning is achieved by assigning a set of hosts IP addresses from the private address spaces specified by RFC1918[3] [106]. These private addresses provide domain isolation by virtue of their reuse within the network — network routing protocols require addresses to be unique, otherwise routing to the correct destination cannot be guaranteed.

To function correctly the NAT must be placed on the boundary of the internal and external networks where its translating algorithm will ensure the NAT is the sole authority capable of correctly mutating packets when traversing the network boundary. This ensures the path the NAT is on, becomes the sole path traversing the two domains. This single ingress path may also be useful for other network functions such as firewalls and traffic logging which both require that they see all traffic entering or leaving a domain of interest.

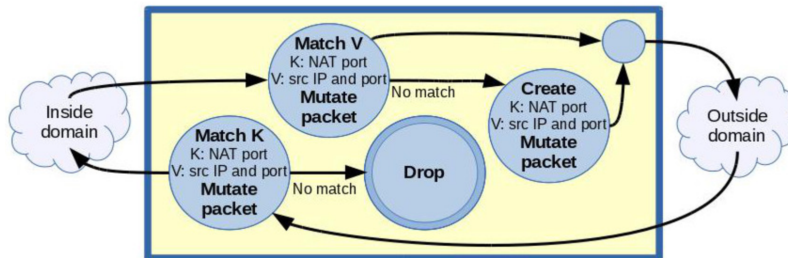


Figure 3.1: Finite State Machine for a stateful NAT (K = key, V = value)

In operation a stateful NAT receives a packet or flow from an internal host (see Figure 3.1), if it recognises the internal host it then mutates the packet by over-writing the source address and port number with its own address and a port number representing this internal host before sending it on. In effect the NAT proxies for the inside domain. If it does not recognise the internal host it allocates a port number[4]. The NAT will subsequently use this port number as a lookup key for a data tuple consisting of the packets original source

[2] At the time the limitations of the IPv4 address space were of concern and the IPv6 address space was still only a concept.

[3] Three IP address ranges are reserved for private address spaces. 10.0.0.0 to 10.255.255.255, 192.16.0.0 to 192.32.255.255 and 192.168.0.0 to 192.168.255.255. Note that other addresses may be used, however the isolation property is strongest when many use the same IP addresses.

[4] A port number is a 16 bit unsigned integer ranging from 0 to 65,535 which allows for sufficient internal hosts for most private domains.

address and port number. The packet is then mutated, over-writing the source IP and port numbers as above. The destination host will then return the reply to the NAT and the NAT uses its port number to retrieve the correct internal address.

A NAT prevents initiation of connections from the external network to the internal network. However it is sometimes useful to expose services to the outside network, this can be achieved using *port-forwarding*. This directs the NAT to use the port number permanently as a key for referencing the internal host. A NAT that is solely concerned with port-forwarding uses static mapping and does not use dynamic state.

Early NAT implementations had static mapping, however later implementations utilised dynamic state to overcome the limitations imposed by the limited number of ports available. To better utilise the ports or flow rule capacity[5] available to a NAT (or to improve performance by minimising the rule space, see Section 2.7.4), it is recognised that only a portion of the hosts in a network require Internet access at any one time. A NAT may use this property by placing unused ports into a pool, allocating port numbers to hosts as needed and de-allocating port numbers when hosts have finished or are idle for a period. This requires the NAT to hold dynamic state recording port number allocation, but allows a single NAT to either; service domains significantly greater than the port number or the NAT's flow rule limit would suggest; or maintain a much smaller rule space compared to static mapping.

Table 3.1 summarises properties of stateless and stateful NATs.

	Stateless NAT — without dynamic state
1	Low coupling and high cohesion
2	Proxies for the domain
3	Represents a domain with a single globally unique addresses
4	Expands the address space available to a network
5	Transparent to hosts on either side
6	Domain isolation due to re-use of public addresses
7	Partitions a network and facilitates nesting isolated domains
8	Single ingress path (without relying on hardware)
9	Maximum rules are limited to the lesser of 65,535 or the functions flow rule limit
10	Drops packets if a rule does not exist
	Stateful NAT — with dynamic state
11	Autonomous, requires no controller input unless the centralised view changes
12	May service internal hosts in excess of port or flow rule limits
13	Only drops incoming packets if a rule does not exist
14	Minimises the rule space held
15	Improves performance by minimising the rule space

Table 3.1: Properties of an in-line NAT

3.2.2 Load Balancer

Traffic loads can potentially overwhelm the compute resources of a single service host which is a problem that can be resolved through parallelism — running multiple instances of a host provider in a manner that is transparent to end hosts.

[5]The ZNYX OpenArchitect LineRateOF switch management software package for their B1 top-of-rack switch, provides capacity for 16,000 L3 flow entries, significantly less than 65,535 port numbers available. See http://www.znyx.com/products/software/lrof-overview/

There are a variety of options for dynamic load balancing and we refer to Cardellinni, Colajanni & Yu (1999) whom describe four categories and their pros and cons [107]: Client-based, DNS-based, Dispatcher-based and Server-based. Our focus here is in-line functionality or Dispatcher-based dynamic load balancing as this provides both fine grained load balancing and better performance compared to other options.

The dynamic load balancer splits the network load amongst the multiple host providers in a way that is considered fair, typically by creating state and over-writing packet headers with its own address to ensure the load balancer proxies for the service (see Figure 3.2). The definition of fair is subjective, implemented by an algorithm and will result in a selected service host for this packet or flow. The selected host may then collect state over the course of a session, which in turn means the load balancer must ensure each flow in the session reliably reaches the correct service host.

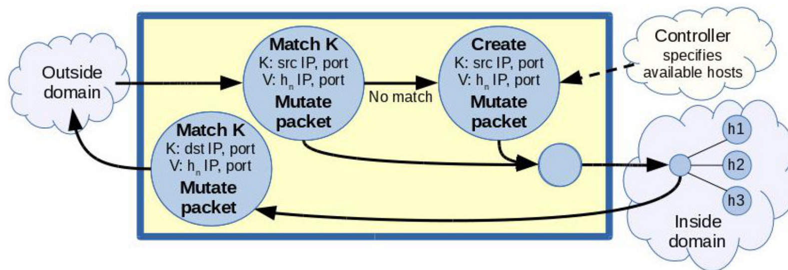


Figure 3.2: Finite State Machine for a Load Balancer (K = key, V = value)

A stateful load balancer dynamically allocate flows to host service providers, relying on locally held state and using its position as a proxy to utilise local knowledge of how busy each host provider is.

A stateless load balancer relies on a third party, typically utilising coarse load allocation (fine grained may be impractical due to the increased switch to third party workload). This is the allocation of blocks of flows, perhaps using hashes or IP address ranges to determine how to allocate flows. For example, a hash from the five tuple of addresses and protocol may form a hash ring[6] that can be split as the third party desires amongst the available hosts. Alternatively the IP address space may be partitioned with suitable rules pushed to an OpenFlow switch, ensuring flows within each partition are forwarded to a dedicated host [108]. These solutions result in coarse grained load distribution which brings management problems when the coarse load distribution needs to change as network loads change.

Changes in network loads may be caused by, for example, the daily cycles of major markets starting and finishing their working days. Being able to consolidate work load onto fewer hosts and ramp up capacity quickly can result in improved resource use. For example, in-line load balancers use fine grained load allocation, meaning new flows are able to utilise new host providers immediately. Existing hosts will not be sent new tasks until the new hosts workload matches theirs.

While the long term load distribution using coarse partitioning is fair, in the short term there are at least two problems that must be managed. First is how to treat the existing flows. The naive approach of changing the partitioning will result in many existing sessions being redirected to new hosts — which will reject them as unrecognised. If this is resolved the second problem is that the load distribution remains unbalanced for longer than a fine

[6]A hash ring utilises a property of hashes in that they are evenly spread across the range of numbers the hash utilises. A fair distribution may be achieved by dividing the number space evenly across hosts. If the hash is created from a packets 5 tuple, which stays constant throughout a session, this number is also guaranteed to result in the same service provider through the session.

grained approach — the existing hosts continue to receive work while the new host remains under-utilised. For example, if four existing hosts have a fifth host added, the fifth host from that point will receive 20% of the traffic. There are strategies to mitigate these problems, for example, an approach may use partitioning to send all traffic to the new host for a period before re-partitioning for long term fair distribution. However it seems the solutions to these problems, an artefact of using an architecture that requires a remote third party, add layers of complexity compared to an in-line stateful function offering fine grained, flow-by-flow, load balancing.

	Stateless Load Balancer — without dynamic state
1	Low coupling and high cohesion
2	Proxies for the service
3	Transparent to hosts either side
4	Consistent, always directs session flows to the same service host
5	Session state may be kept by the service host
6	Fair distribution is chosen by a third party
7	Capacity can be expanded or contracted
8	The distribution of flows is coarse, which complicates increasing capacity
	Stateful Load Balancer — with dynamic state
9	Autonomous, requires no controller input unless the centralised view changes
10	Fine-grained distribution of flows, can utilise new resources immediately
11	Local knowledge of existing flows informs fair load distribution

Table 3.2: Properties of an in-line Load Balancer

3.2.3 Firewall

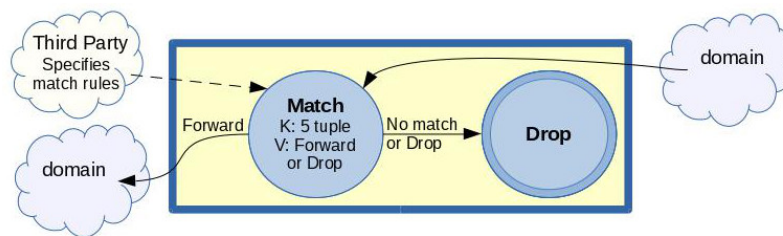


Figure 3.3: Finite State Machine for a switch as a stateless firewall

The phrase 'firewall' has many different meanings, for example, firewalls might be distinguished based on what level of the OSI model they service, such as application gateways (level 7), circuit gateways (layer 4) and MAC layer firewalls (layer 2). Here we abstract away from that detail and discuss common functionality and the management of its state. We start with defining four types of firewall.

A **stateless firewall** is an algorithm that filters traffic based on an access control list (ACL) describing addresses (IP or MAC), applications (ports) and protocols as either trusted or dangerous[7]. Such a firewall is shown as a simple finite state machine in Figure 3.3). It cannot list every host on the Internet and for safety will often default to treating all unrecognised packets as dangerous. It might be observed that it is trivial to direct a switch with flow rules

[7]Also referred to as white and black lists.

to manifest this behaviour and the standard SDN firewall modules provided with Floodlight and Ryu are of this type.

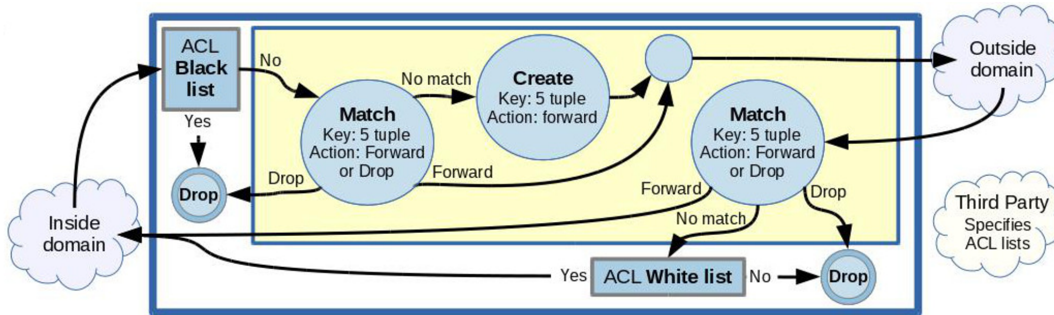


Figure 3.4: Finite State Machine for a stateful firewall — first model

A **stateful firewall** is a firewall algorithm with dynamic state — it manages its own rule space as shown in Figure 3.4. This finite state machine is able to treat unknown packets usefully, preventing incoming packets from untrusted sources from passing unless an internal host has initiated the exchange. This requires the firewall to provide different treatment for packets arriving on the ports representing each domain and helps to prevent, for example, attempts by external attackers to map the domain and directly attack key internal hosts.

An **industrial firewall** is a hardware appliance designed for unified threat management. It is made up of a variety of algorithms, including a stateful firewall, working in concert to defeat many threats. These may include, for example, a load balancer to provide capacity in the face of an attack and a NAT to isolate the internal network being protected.

A **distributed firewall** describes the firewall algorithm as distributed to hosts and switches within the domain, rather than at the domain's perimeter. Typically the goal is to mitigate the possibility of attacks originating within the domains perimeter. It might be observed that this may be resource intensive, for example, a military camp may post guards inside every building, requiring many guards and those passing between buildings to undergo many checks. It also allows outsiders unimpeded access to the camp to observe the buildings and traffic. Buildings may then be attacked in isolation having researched the construction plans, the buildings occupants and any existing, documented vulnerabilities. If the firewall (guard) is hosted on the system being attacked, the attackers start with a foot already inside the door.

In operation a firewall algorithm will make two key assumptions; the first is assuming that protected hosts can be treated as trustworthy — an unsafe assumption if a network administrator does not utilise other tools and processes, including the human resources department. The second is assuming the domain's perimeter is closed and all transiting traffic is via the path the firewall occupies, again this assumption is unsafe if the network operator has not prevented other exit points.

A stateful firewall algorithm is often paired with a NAT because of the useful properties the NAT provides; it defines a point where connections move from the internal domain to the external. A firewall placed at this point is guaranteed to see all transiting traffic. In addition, because the internal domain is relatively static, the ACL may specify every internal host in the internal domain as trusted or dangerous (quarantined), typically there will be no unknown hosts internally.

Firewall behaviour is described in RFC2979 [78], RFC3234 [15], by NIST [79] and in articles and text books that repeat the following three widely accepted properties of a firewall [80–83];

1. All communication must pass through the firewall

2. The firewall permits only traffic that is authorised

3. The firewall can withstand attacks on itself

	Stateless Firewall — without dynamic state
1	Low coupling and high cohesion
2	Transparent to hosts either side (unless flows are dropped)
3	All entry points to the domain must be identified and controlled
4	Partitions universe of hosts into good versus bad
5	Must see all communication
6	Must only permit authorised traffic, typically specified on an ACL
7	Must be resistant to attacks
	Stateful Firewall — with dynamic state
8	Autonomous, requires no controller input unless the centralised view changes
9	Utilises local knowledge of outgoing flows to allow incoming flows
10	Responds to end of session packets to recover resources

Table 3.3: Properties of an in-line Firewall

3.3 Formally describing a generic network function

This formalism is a starting point based on a study of three common network functions. As more network functions are analysed the expectation is this will inform future iterations of this generic model. The goal is to create a generic network function that describes a wide range of behaviours such as packet forwarding, dropping packets, mutating packets, creating multiple outputs, operating in isolation, dynamic behaviours and more.

For convenience we ignore the influence of the network administrator who, possibly via a controller, will deploy network functions to fulfil higher goals, provide configuration and may periodically update the initial configuration. This is in order to focus on packet processing which occurs orders of magnitude faster than network administrator activities.

For convenience the packet data is largely ignored as most network functionality is based on packet header information. Interesting things can be done with data such as deep packet inspection (DPI), however this is considered a special case (not considered further here) as it may involve a mix of encryption services, proxy services and message reassembly before the DPI can perform its task.

All network functions observed so far operate in isolation. This isolation property means the function is decoupled from other functions, responding only to the packets it observes. This also holds when considering network functions that are deployed in pairs, such as encryption and decryption services — each device still operates in isolation. This isolation property allows network functions to be analysed as discrete finite state machines.

This work acknowledges and builds on the work of Joseph & Stoica (2008) whom describe an algebra for specifying middleboxes [109]. The approach used here is Typed Lambda Calculus which forms a basis for typed functional programming languages.

1	T	the set of all possible network function types
2	$FW \in T$	a firewall (FW) is a network function
3	$\{SW, LB, NAT\} \subset T$	as are, for example, switch (SW), load balancer (LB) and (NAT)
4	P	the domain of all packets that may be on a network
5	P^*	all packet sequences that may be on a network
6	P^{**}	all sequences of packet sequences that may be on a network
7	P^+	P^* but not the empty packet sequence
8	$p \in P$	one packet
9	$p^+ \in P^*$	one or more packets forming a packet sequence, note that the order of the packets is relative to the observing function
10	H	the domain of all possible packet headers
10	$h \in H$	one header
12	$address \in h$	a header contains a set of fields used for determining routing. For example, source IP:port, destination IP:port and protocol type which form a 5 tuple. *address* is deliberately abstract to capture a variety of protocols including MAC addressing.
13	$header(p) \mapsto h$	a packet has a header
14	$addr(p) \mapsto address$	a packet has address fields
15	$FLOW$	The domain of all possible flows that may be on a network such that $\{p_1...p_n \mid p \in P,\ (\forall\, i \cdot 1 \leq i \leq n,\ addr(p_i) = addr(p_1))\}$
16	$flow \in FLOW$	a sequence of packets sharing common address fields
17	$head(flow) \mapsto p$	the first packet in a flow
18	$tail(flow) \mapsto p^*$	the flow after the first packet
19	$flow_addr(flow) \mapsto$	$address$ equivalent to $addr(head(flow)) \mapsto address$
20	$action$	an instruction for the network function, for example $'drop'$ or $'forward'$
21	$R = address\ x\ action$	domain of all possible rules
22	$r \in R$	a rule is an *address* and an *action*
23	$R_T \subset R$	the set of all possible rules for this network function type
24	$domain(R_T) = address$	the set of all addresses in R_T
25	$R_{T(current)} \in pow(R_T)$	the current set of rules held by this network function
26	$rule(address, R_{T(current)})$	$\mapsto \{action,\ \theta\}$ finds $r \in R_{T(current)}$ and returns its *action* or null if no rule is found
27	$f_rule(flow, R_{T(current)})$	$\mapsto \{action,\ \theta\}$ equivalent to $rule(flow_addr(flow),\ R_{T(current)}) \mapsto \{action, \theta\}$
28	θ	null
29	ϵ	the empty packet sequence
30	x'	x prime is a (possibly) changed x after a calculation

Table 3.4: Algebra definitions

3.3.1 Packet flows

A flow is the transmission of a sequence of packets across the network from one host to another. Because the ordering of the packets will mutate across the network and only be

rectified at the end host, the packet sequence observed by a network function, is the order in which packets arrive.

The flow's packets are identified by a common packet *address*, often the five tuple consisting of the message protocol and the source and destination addresses (IP address and port number). Table 3.4 introduces the definitions.

3.3.2 A stateless network function

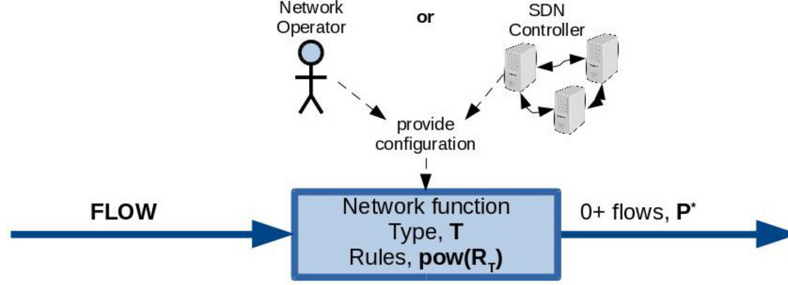


Figure 3.5: Automaton for a stateless network function

NF = Network Function

$$NF : (T,\ FLOW,\ pow(R_T)) \rightarrow P^* \tag{3.1}$$

We first define the Network Function (NF) (3.1), which takes as a tuple a function type $t \in T$ which describes the algorithm to be applied to a packet flow $flow \in FLOW$ and the functions current set of rules $R_{T(current)} \in pow(R_T)$. With these inputs NF will output a sequence of packets $p^* \in P^*$. This sequence may be the empty sequence or it may consist of one or more flows. For example, a firewall dropping a flow will output the empty packet sequence ϵ while a logger may output the packet sequence observed plus a packet sequence destined for remote storage. Figure 3.5 shows a simple automata for a stateless (no dynamic state) network function (NF).

A network function will treat each packet in a flow, in turn. To facilitate this we define sNF.

sNF = stateless Network Function

$$sNF : (T,\ P,\ pow(R_T)) \rightarrow P^* \tag{3.2}$$

The stateless network function sNF (3.2) takes a type $t \in T$, the first packet $head(flow) \in P$ in the packet sequence and the network functions current set of rules $R_{T(current)} \in pow(R_T)$. The function sNF outputs a packet sequence P^*, possibly containing multiple flows.

$$sNF(t,\ head(flow),\ R_{T(current)}) \rightarrow P^*$$

The function NF can now be defined recursively using sNF. Note that the definition of NF accepting an empty packet sequence, terminates the recursion.

$$NF(t,\ flow,\ R_{T(current)}) = sNF(t,\ head(flow),\ R_{T(current)}) :: NF(t,\ tail(flow),\ R_{T(current)})$$
$$NF(t,\ \epsilon,\ R_{T(current)}) = \epsilon$$

Operate in isolation with low coupling, high cohesion

The ability to define a network function as NF (3.1) is possible because network functions operate in isolation on packet flows, they feature low coupling and high cohesion, acting only on the network packets they see — redirecting, dropping, modifying and/or retaining state information. Put another way, a network function blindly applies an algorithm and a set of rules to the packets passing through it and may keep state information should it need to act on other related packets. It performs at packet flow rates, which may be orders of magnitude faster than configuration updates provided by human or SDN controllers.

Packet forwarding and packet mutation

Packet forwarding is a feature of all network functions and packet mutation is a feature of many.

Packet forwarding is captured in the output of NF (3.1) as a packet sequence that may be empty, may not be the same as the packets received and may contain multiple flows.

Packet mutation is the rewriting of header information in order to achieve a goal, typically involving addressing. For example, NATs and load balancers have both been described as mutating packet headers in order to achieve their functionality. The NAT to act as a proxy to a domain and the load balancer to act as a proxy to a set of service hosts. Mutation of packet data is a special case and not explored here. However mutating a packet by treating it as data (encapsulating it inside more IP headers) is briefly explored next in services that transparently encrypt packets (including header information) and send them to a remote service separated by time and/or distance for decryption.

Transparent operation

The presence of a network function should not be noticed by the end hosts in a successful communication. This is trivial when there is no packet mutation, but when packet mutation forms part of the functionality There will typically also be a need to undo the mutation.

This property is formally described next. Three examples are shown of mutating yet transparent functionality, a single network function that acts as a proxy for others, paired network functions that are characterised by separation over time and/or distance and a cache which intercepts traffic and returns replies on behalf of a remote host.

This example of a single packet mutating network function acts as the translator between domains. Hosts on either side access the proxy which in turn allows access to what may otherwise be an inaccessible service. This describes, for example, a NAT and Load Balancer.

$$NF_{instance_1}(t,\ flow,\ R_{T(current)}) \rightarrow flow' \\ \implies NF_{instance_1}(t,\ flow',\ R_{T(current)}) \rightarrow flow$$

Note that the function takes a $flow$, mutates it to $flow'$ and on the flows return, will mutate $flow'$ back to $flow$.

Paired functions offer complementary services, for example, an encryption function may encrypt a message, requiring a remote function to decrypt the message. This might be network tunnels where the entire packet, including the header information, is encrypted and sent as data to the remote proxy for decryption.

$$NF_{encrypt}(t,\ flow,\ R_{T(current)}) \rightarrow flow' \\ \implies NF_{decrypt}(t,\ flow',\ R_{T(current)}) \rightarrow flow$$

Caches provide transparent operation while seeking to increase response speed and reduce the network load between the cache and the source host. Here NF_{host} is the source, NF_{cache} is the cache. What is portrayed next is that both return the same reply $flow'$. That the cache is reached first and replies on behalf of the host is transparent to the requester.

$$
\begin{aligned}
&NF_{cache}(t,\ flow,\ R_{T(current)}) \mapsto flow' \lor \\
&(\\
&\qquad NF_{cache}(t,\ flow,\ R_{T(current)}) \mapsto \epsilon \land \\
&\qquad NF_{host}(t,\ flow,\ R_{T(current)}) \mapsto flow' \\
&)
\end{aligned}
$$

Packet mutation and function transparency both have implications for function chaining (see Section 3.3.6).

3.3.3 A stateful network function

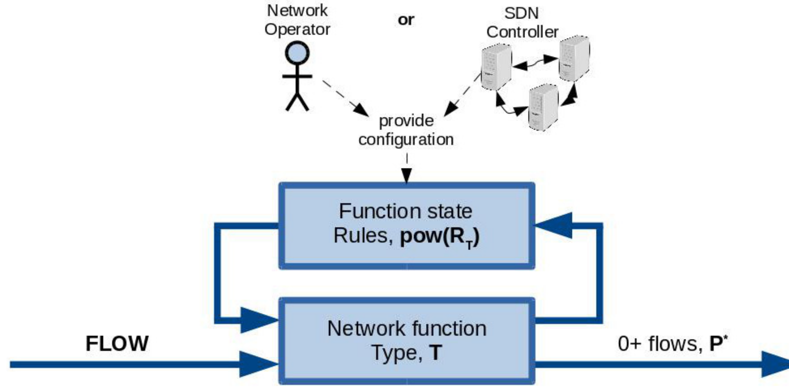


Figure 3.6: Automaton for a stateful network function

Adding state to a network function automata allows it to retain information on observed packet sequences and apply it to future packet sequences. This can be seen in Figure 3.6 showing a finite state automata where the network function accepts a packet sequence and the current state, then outputs zero or more packet sequences and the new state. We use this model from here as it captures both stateful and stateless behaviour — stateless behaviour merely accepts and outputs an unchanged set of states.

Next we describe the stateful function $sfNF$ used by NF (3.1). It takes the type of function $t \in T$, the head packet from a sequence of packets between two hosts $head(flow) \in P$ and the network functions current set of rules $R_{T(current)} \in pow(R_T)$. The function $sfNF$ outputs a packet sequence P^*, possibly containing multiple flows and a (possibly modified) set of rules $R'_{T(current)} \in pow(R_T)$.

$sfNF$ = stateful Network Function

$$
sfNF : (T,\ P, pow(R_T)) \rightarrow (P^*,\ pow(R_T)) \tag{3.3}
$$

The $sfNF$ (3.3) is customised as needed for the network function. For example, a load balancer, NAT and firewall each have unique definitions. A firewall example is provided shortly.

Two helper functions follow. These accept the tuple output of $sfNF$, that is $(P^*, pow(R_T))$ and returns the packet sequence P^* (which may contain many flows) or the current set of rules $pow(R_T)$.

$$flows(p^*, R_{T(current)}) \mapsto p^*$$
$$rules(p^*, R_{T(current)}) \mapsto R_{T(current)}$$

These helper functions are utilised in the NF to create a recursive function. The intuition is that the first packet in a flow checks existing state and may cause the state to be modified, the following packets check against the state left by earlier packets, possibly modifying it in turn.

For convenience, we repeat equation (3.1) here.

$$NF : (T, FLOW, pow(R_T)) \rightarrow P^*$$

Let $\pi = sfNF(t, head(flow), R_{T(current)})$

$$\begin{aligned} &NF(\ t,\ flow,\ R_{T(current)}) = flows(\ \pi\ ) :: NF(\ t,\ tail(flow),\ rules(\ \pi\ )\ ) \\ &NF(\ t,\ \epsilon,\ R_{T(current)}) = \epsilon \end{aligned} \tag{3.4}$$

3.3.4 A firewall example — stateless

Applying this to a practical example may illustrate the math.

A firewall is both an algorithm and a type of network function $FW \in T$. It has a set of rules $R_{FW(current)} \in pow(R_{FW})$ designed to facilitate its functionality.

Firewall rules $r \in R_{FW(current)}$ adopt the form {address, action} where the address is the common *address* shared by packets in a *flow* while the action is from the set $\{\theta, `drop', `forward'\}$. Such that;

$$f_rule(flow, R_{FW(current)}) \mapsto \theta \vee$$
$$f_rule(flow, R_{FW(current)}) \mapsto `drop' \vee$$
$$f_rule(flow, R_{FW(current)}) \mapsto `forward'$$

.

A stateless firewall implementation of $sfNF$ (no dynamic state change so $R'_{FW(current)} = R_{FW(current)}$) is described next, dynamic state is introduced shortly. The default position for this firewall is to *'drop'* unknown packets, other implementations of a stateless firewall may default to *'forward'*.

For convenience, we repeat equation (3.3) here.

$$sfNF : (T, P, pow(R_T)) \rightarrow (P^*, pow(R_T)))$$

$$\begin{aligned} &sfNF(t, p, R_{FW(current)}) = \{\ \exists\ p', R'_{FW(current)}\ | \\ &\quad t = FW \wedge \\ &\quad p' \in (P \cup \{\epsilon\}) \wedge \\ &\quad R'_{FW(current)} = R_{FW(current)} \wedge \end{aligned}$$

$$
\begin{aligned}
&(\\
&\quad (f_rule(flow,\ R_{FW(current)}) = \theta\ \wedge\ p' = \epsilon) \vee\\
&\quad (f_rule(flow,\ R_{FW(current)}) = `drop'\ \wedge\ p' = \epsilon\ ) \vee\\
&\quad (f_rule(flow,\ R_{FW(current)}) = `forward'\ \wedge\ p' = p)\\
&)\}
\end{aligned}
$$

A walk through the stateless firewall

In this stateless firewall example, the recursive function NF (3.1) treats each packet in the flow by calling $sfNF$ (3.3) followed by calling NF again. When the flow is empty it will call the NF stop condition which accepts and outputs the empty sequence. Each packet is treated identically and the results are concatenated. Consequently depending on the rule for the $flow$ the output of NF_{FW} will be either the input flow or an empty sequenceϵ.

We start with defining a sequence of three packets.

Let $flow = [p_1,\ p_2,\ p_3]$

And the algorithm defined earlier (3.4).

Let $\pi = sfNF(t,\ head(flow),\ R_{FW(current)})$

$$
\begin{aligned}
&NF(\ t,\ flow,\ R_{FW(current)}) = flows(\pi) :: NF(\ t,\ tail(flow),\ rules(\pi)\ )\ )\\
&NF(\ t,\ \epsilon,\ R_{T(current)}) = \epsilon
\end{aligned}
$$

Next NF is substituted for $sfNF$ functions and the three packet sequence $flow$ is applied. Note the last instance of the function NF is the stop condition and results in ϵ.

$$
\begin{aligned}
p_{result} = {} & flows(\ sfNF(t,\ head([p_1,\ p_2,\ p_3]),\ R_{FW(current)})\ ) ::\\
& flows(\ sfNF(t,\ head([p_2,\ p_3]),\ rules(sfNF(t,\ head([p_1,\ p_2,\ p_3]),\ R_{FW(current)}))\ ) ::\\
& flows(\ sfNF(t,\ head([p_3]),\ rules(sfNF(t,\ head([p_2,\ p_3]),\ R_{FW(current)}))\ ) :: \epsilon
\end{aligned}
$$

Note that $rules(sfNF(t,\ head([p_2,\ p_3]),\ R_{FW(current)}))$ in this stateless firewall example, will always return $R_{FW(current)}$. This simplifies the equation.

$$
\begin{aligned}
p_{result} = {} & flows(\ sfNF(t,\ head([p_1,\ p_2,\ p_3]),\ R_{FW(current)})\ ) ::\\
& flows(\ sfNF(t,\ head([p_2,\ p_3]),\ R_{FW(current)}\ ) ::\\
& flows(\ sfNF(t,\ head([p_3]),\ R_{FW(current)}\ ) :: \epsilon
\end{aligned}
$$

This will result in one of the following outcomes;

- Where $f_rule(flow,\ R_{FW(current)})$ is null θ. There is no rule allowing this flow. This results in $sfNF$ assigning the default value, the empty sequence ϵ to p'. Consequently all the members of packet sequence $flow$ are dropped. $p_{result} = \epsilon$.

- Where $f_rule(flow,\ R_{FW(current)})$ is $`drop'$ the flow is dangerous and all the members of packet sequence $flow$ are dropped. $p_{result} = \epsilon$.

- Where $f_rule(flow,\ R_{FW(current)})$ is $`forward'$. The flow is permitted and $sfNF$ returns the packet value of $p' = p$ which is concatenated with previous packets to form the packet sequence p^*. All the members of packet sequence $flow$ are therefore forwarded. $p_{result} = flow = [p_1,\ p_2,\ p_3]$.

3.3.5 A firewall example — stateful

Next we describe a helper function *dynamic* that modifies the rules in a NF. It takes a rule $r \in R_T$ and the current set of rules $R_{T(current)} \in pow(R_T)$ and if r is null it returns the unmodified set of rules $R'_{T(current)} = R_{T(current)}$ otherwise r is either added if it does not already exist in $R_{T(current)}$ or it is removed.

$$dynamic : (\ R_T \cup \{\theta\},\ pow(R_T)) \rightarrow pow(R_T) \tag{3.5}$$

$$\begin{aligned}
& dynamic(\ r,\ R_{T(current)}) = \{\exists\ R'_{T(current)} \mid \\
& \quad R'_{T(current)} \in pow(R_T) \wedge \\
& \quad (\\
& \quad\quad (r = \theta \wedge R'_{T(current)} = R_{T(current)}) \vee \\
& \quad\quad (r \in R_{T(current)} \wedge R'_{T(current)} = R_{T(current)} \setminus \{r\}) \vee \\
& \quad\quad (r \notin R_{T(current)} \wedge R'_{T(current)} = R_{T(current)} \cup \{r\}) \\
& \quad)\ \}
\end{aligned}$$

A dynamic firewall needs a function to identify the replies generated by outgoing flows. We define this as a helper that takes a $flow$ and returns the *address* for its expected reply.

$$reply : (\ FLOW) \rightarrow address$$

Looking again at $sfNF$ we can now modify the function to allow trusted hosts to create connections with untrusted hosts.

$$\begin{aligned}
& sfNF(t,\ p,\ R_{FW(current)}) = \{\ \exists\ p',\ R'_{FW(current)} \mid \\
& \quad t = FW \wedge \\
& \quad p' \in (P\ \cup\ \{\epsilon\}) \wedge \\
& \quad (\\
& \quad\quad (\\
& \quad\quad\quad (\ f_rule(flow,\ R_{FW(current)}) = \text{`}forward\text{'} \wedge \\
& \quad\quad\quad\quad reply(flow) \notin domain(R_{FW(current)})\ ) \\
& \quad\quad\quad\quad\quad \Longrightarrow\ R'_{FW(current)} = dynamic(reply(flow) \mapsto \text{`}forward\text{'},\ R_{FW(current)}) \\
& \quad\quad)\ \wedge\ (\\
& \quad\quad\quad \neg(\ f_rule(flow,\ R_{FW(current)}) = \text{`}forward\text{'} \wedge \\
& \quad\quad\quad\quad reply(flow) \notin domain(R_{FW(current)})\ ) \\
& \quad\quad\quad\quad\quad \Longrightarrow\ R'_{FW(current)} = R_{FW(current)} \\
& \quad\quad) \\
& \quad)\ \wedge\ (\\
& \quad\quad (f_rule(flow,\ R_{FW(current)}) = \theta\ \wedge\ p' = \epsilon\ ) \vee \\
& \quad\quad (f_rule(flow,\ R_{FW(current)}) = \text{`}drop\text{'}\ \wedge\ p' = \epsilon) \vee \\
& \quad\quad (f_rule(flow,\ R_{FW(current)}) = \text{`}forward\text{'}\ \wedge\ p' = p) \\
& \quad)\}
\end{aligned}$$

Removing rules at the end of a flow takes considerably more math and is protocol dependent. We provide this in chapter 5 where a firewall model is formalised within a tool suitable for creating and proving the correctness of formal models.

At this point a generic network function has been formally modelled and illustrated first with a stateless firewall, then a stateful firewall. Next is a discussion on formalising chains of network functions.

3.3.6 Formally describing a chain of network functions

A service chain describes one or more network functions connected in sequence. Like individual network functions, such a chain operates in isolation acting only on the network packets it sees and it is transparent to the user.

Packet mutation and dynamic rules mean the order of the network functions within the chain must remain consistent. There may also be an optimal ordering, for example, a firewall which is designed to be attack resistant, should be exposed to the external internet to filter packets before they are treated by the rest of the chain. Consistent order means a packet traversing a chain of middleboxes one way, traverses it in reverse order on return (see Figure 3.7). This property precludes the possibility of cycles within the chain. Proprietary hardware connected 'on the wire' implicitly has this consistency trait (but may not be optimally ordered), but with software based network functions this is not a given and must form a rule when chaining functions.

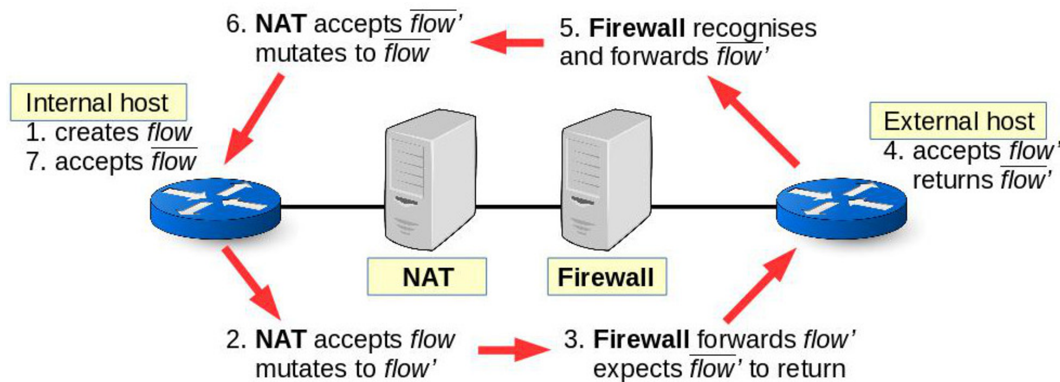


Figure 3.7: Passing a flow through a chain of network functions and receiving the reply.

Chaining and packet mutation

Formally, a chain is a list of functions passing packets. The next function takes the result of the previous function as input. For example, the following equation shows two functions performing NAT and firewall roles as packets leave a network. The NAT mutates the packets and the firewall on recognising the mutated packets will allow them if allowed by its rules.

In summary, the flow arrives as $flow$ and leaves the chain and domain mutated to $flow'$.

$$NF(NAT, flow, R_{NAT(current)}) \mapsto flow'$$
$$NF(FW, flow', R_{FW(current)}) \mapsto flow'$$

Chaining and transparency

Transparency means the end hosts in a valid communication are unaware of the presence of network functions. Continuing the example where the first host sent a message $flow$, here we treat the reply $flow'$.

On returning to the network the packet sequence $flow'$ is recognised by the stateful firewall and is forwarded. NAT using retained state, recognises $flow'$ and mutates it back to $flow$.

$$NF(FW, flow', R_{FW(current)}) \mapsto flow'$$
$$NF(NAT, flow', R_{NAT(current)}) \mapsto flow$$

Chaining consistency

Should ordering not be observed and the NAT is visited first, the NAT will recognise the returning $flow'$ and transform it back to $flow$.

The firewall now takes $flow$ as input, however the stateful firewall will not recognise $flow$ as it has never seen it, only the post mutated version $flow'$. Therefore the firewall will treat it as an unsolicited flow from an untrusted external host and output the empty packet sequence ϵ.

In summary, not following a consistent order in this example results in the returning $flow'$ becoming the empty sequence ϵ. (Less formally, the packet is dropped.)

$$NF(NAT,\ flow', R_{NAT(current)}) \mapsto flow$$
$$NF(FW,\ flow, R_{FW(current)}) \mapsto \epsilon$$

The next step is to develop these models further using tools.

3.4 Modelling Tools

3.4.1 Rodin

Rodin[8] [110] has been developed as an Eclipse based research tool that supports the creation of mathematical proofs and the process of stepwise model refinement. It enables reasoning over and analysing a model, proving in a mathematically rigorous way that the required properties, expressed as invariants, are satisfied. The tool is open source, licensed under the Eclipse Public License - v 1.0 (a copyleft license) and has been supported by various European Union research initiatives since 2004. Being a tool developed by researchers it is neither fully featured nor intuitive to use and the documentation is light. The learning curve is steep.

Underlying principles of Rodin include fast feedback to the modeller, proof obligation generation and automatically discharging trivial proof obligations. Proof obligations that are not discharged are flagged as errors and hints provided to allow either the model to be rectified or the designer to provide a manually assisted proof.

3.4.2 Event-B

Event-B [103] is a calculus for modelling derived from the B-method [101] and inspired by ideas from Action systems [111]. Both the B-method and Event-B were created by Jean-Raymond Abrial with Event-B representing an evolution of the B-method towards becoming easier to learn and utilise.

When using Event-B, the developer creates a state machine representing a high-level abstract model of the problem. Each state in the machine, called an event in Event-B, has predicates based on global variables and actions that change global variables. When an event's predicates are satisfied, it becomes a state the machine may transition to. Transitioning to an event makes changes to the model which may in turn influence predicates which will change what transitions are available next.

3.4.3 An example model built in Rodin, using Event-B

We examine a simple model representing the passing of packets between two domains. Figure 3.8 shows the labelled transition system. Each transition is a directed arc.

[8] Available for download at http://www.event-b.org/

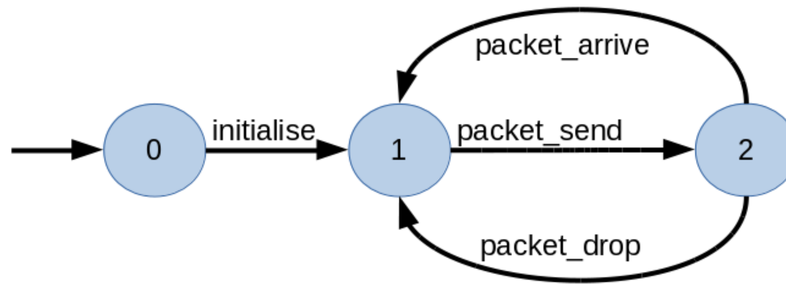


Figure 3.8: Labelled transition system, packet events as transitions between states

An Event-B model as crafted in the Rodin tool is shown over Figures 3.9 to 3.12. The model starts with defining the environment or context the network function will operate in, the global value constants; followed by the state machine with its four events. It is the first model in what will be a series of stepwise refinements and it defines abstract properties of the network — an internal network, an external network, events that allow message passing between the two, which utilise packets comprised of a source and destination address. In this first model, many details are abstracted away, for example, network addresses, packet data, protocols, connection paths, trust relationships and more.

```
 1 context c_e00_domains
 2
 3 constants int ext NODES
 4
 5 axioms
 6   @axm0_1 NODES = 0‥1000 // nodes are a finite set
 7   @axm0_2 int = 1‥5 // internal nodes are known/listed
 8   @axm0_3 ext = (NODES ∖ (int ∪ {0}))
 9   theorem @axm0_4 int ∩ ext = ∅
10   theorem @axm0_5 NODES = int ∪ ext ∪ {0}
11 end
```

Figure 3.9: Rodin and Event-B, model context

Each model has a context that defines the constants used in the model (see Figure 3.9). In this case we create the constants $int, ext, NODES$ (line 3) to represent the abstract idea of hosts and addressing. $NODES$ are defined in line 6 as a set of 1,000 abstract hosts. In lines 7 and 8 int is defined as $NODES$ 1 to 5 while ext is all other $NODES$ except 0. The 0 node is treated as a special case, reserving it for use as the null address. Line 9 states that a node may be either int or ext but not both and line 10 confirms that $NODES$ is comprised of int, ext and null nodes.

As an examples of using $NODES$ to represent abstract addresses, a firewall is interested in three classes of address; trusted untrusted and dangerous. In later refinements we define trusted as a set of nodes including int nodes and we define dangerous as node 666. The remaining nodes are considered untrusted. A benefit of treating a single node as an unbounded finite set of addresses, is that it allows the set of abstract $NODES$ to be finite, with subsequent benefits for modelling, such as limiting state explosion.

Variables are defined in two places in the state machine. Global variables such as src, dst are named as shown in Figure 3.10 (line 3) and formally defined in lines 6 to 11. Local variables are defined in events. Events are triggered by global variables satisfying predicates, called guards in Event-B. For this model we require a source src and destination dst, which are defined in lines 6 and 7 as elements of $NODES$. As a pair, the $\{src,\ dst\}$ will represent a packet. Obviously we are abstracting away many details such as protocols and data carried,

```
 1 machine e00_domains sees c_e00_domains
 2
 3 variables src dst
 4
 5 invariants
 6   @inv00_1 src ∈ NODES // universe of node addresses
 7   @inv00_2 dst ∈ NODES
 8   @inv00_3 src ≠ 0 ∧ src ∈ int ⇒ dst ∈ ext // src and dst are in different domains
 9   @inv00_4 dst ≠ 0 ∧ dst ∈ int ⇒ src ∈ ext
10   @inv00_5 src=0 ⇒ dst=0 // 0,0 is the null packet, 0 |-> 0
11   @inv00_6 dst=0 ⇒ src=0
12
13 events
14   event INITIALISATION
15     then
16       @act00_1 src ≔ 0
17       @act00_2 dst ≔ 0
18   end
```

Figure 3.10: Rodin and Event-B, model machine — invariants and initialisation

and retaining only the details required for communication between hosts. Our interest is in the transitions from one domain to another $\{int, ext\}$, so lines 8 and 9 state that the source and destination must be in different domains. We have use for a null packet and represent it with null source and destination addresses. The packet with only one null address is malformed so lines 10 and 11 state that if one address is null, so is the other.

Initialisation is the first event (lines 16 and 17), it results in both *src* and *dst* being assigned the null node value of 0, creating the null packet.

The Rodin tool allows checking proofs regularly for correctness. For example, should *dst* be initialised as 1 instead of 0, then the model fails, specifically it fails invariants 4 and 5. Invariant 4 (line 9) fails because it states that if *dst* is an element of *int* then *src* must be an element of *ext* which it is not. Invariant 5 (line 10) fails because if *src* is null, so must *dst* be null which it is not.

```
20   event packet_send
21     any x y
22     where
23       @grd00_1 src = 0 ∧ dst = 0
24       @grd00_2 x ∈ (int ∪ ext) ∧ y ∈ (int ∪ ext) // int U ext excludes 0
25       @grd00_3 x≠0 ∧ y≠0 // making this explicit
26       @grd00_4 x ∈ int ⇒ y ∈ ext // x & y are in different domains
27       @grd00_5 y ∈ int ⇒ x ∈ ext
28     then
29       @act00_1 src, dst ≔ x, y
30   end
```

Figure 3.11: Rodin and Event-B, model machine — packet_send

The event *packet_send* is shown in Figure 3.11. On line 21 two local variables x and y are created. The predicate (labelled *where*) is satisfied where both *src* and *dst* are null (line 23). Predicates for local variables like x and y (lines 24 to 27) define their value and are always satisfied (or if poorly written are never satisfied).

The initialisation event assigned both *src* and *dst* the null value meaning the first predicate (line 23) passes. The predicates on x and y (lines 24 to 27) will result in assigning x and y nodes that are designed to satisfy the invariants imposed on *src* and *dst*. As the predicates pass, the event is usable and if selected the action (labelled *then*) will assign the values of x

and y to *src* and *dst* (line 29).

The third predicate (line 25) stating $x \neq 0 \ \wedge y \neq 0$ should perhaps be obvious given the set $\{int \cup ext\}$ does not include 0 but was made explicit because it was found to assist Rodin's automated provers which otherwise required creating manual proofs whenever the model is updated.

```
32   event packet_arrive
33     any x y
34     where
35       @grd00_1 src ≠ 0 ∧ dst ≠ 0
36       @grd00_2 x = 0 ∧ y = 0
37     then
38       @act00_2 src, dst ≔ x, y // null packet?
39   end
40
41   event packet_dropped
42     where
43       @grd00_1 src ≠ 0 ∧ dst ≠ 0
44     then
45       @act00_1 src, dst ≔ 0, 0 // null packet
46   end
47 end
```

Figure 3.12: Rodin and Event-B, model machine — packet_arrived, packet_dropped

Next are the *packet_arrive* and the *packet_dropped* events, shown in Figure 3.12. Initially we are not concerned with the path taken by the packet, only that it travels from one domain to another, so paths are abstracted and introduced in a later refinement.

In both cases the event predicates (line 36 and 43) are that *src* and *dst* are not null. This means that after *packet_send* both events are valid choices (recall that predicates for local variables are always satisfied). If *packet_dropped* is selected, *src* and *dst* are set to the null address. *packet_arrive* has the same effect but uses variables, setting the foundation for allowing a reply in a later refinement, for example, by setting x and y to *src* and *dst* we can swap the *src* and *dst* values in the action.

This model reflects message sending across two domains using a source and destination address and represents a mathematically rigorous first abstraction that can be built on.

3.4.4 Model refinement

To refine a model is to make the abstraction more concrete. This is the process of stepwise refinement where further detail is added to the abstract model, possibly adding new values, adding a new event or by expanding an event into several new events. These changes are incremental to allow for the creation of new proof obligations. Once all proof obligations are satisfied, the iterative process of stepwise refinement repeats.

Based on the example above there are many refinements to make, for example we need an event that defines a path that every packet traverses, we also need to represent protocols, handshakes, more packet detail and ultimately create a model of the system under test.

A valid question is when do you stop refining? With MDD the answer is when the model can be compiled into code. With MBT the task is to recreate the observable behaviours of the system under test which typically requires far less detail.

3.5 Model-Based Testing

> "Model-based testing is the automation of the design of black-box tests." Utting *et al.* (2010) [6] p8.

Manual black-box testing involves designing a subset of tests which it is hoped will expose the most crucial bugs in the available time-frame. This is limited by being a process that is subject to human implementations, commercial realities and time constraints. Model-based testing in contrast aims to formally model the system early in the software development life-cycle with subsequent benefits for validating system architecture and enabling the mechanical generation of tests.

Model-Based Testing (MBT) shares several benefits of model-driven development (MDD); abstraction, understandability, accuracy, predictiveness and inexpensive [98–100,103]. However rather than generating models for the purpose of code generation, MBT is interested in higher level abstractions of software's observable behaviours. The goal is to validate the correctness of a hand-crafted implementation, rather than replace it.

A key advantage of MBT is that it exposes underlying assumptions and failures in communication, early. These cause bugs that may only become apparent much later. Boehm (1981) showed the cost of fixing a bug is 20-100 times more expensive in operation than in the requirements phase [112]. It also prompts a better understanding of the system which tends to lead to better architecture. Using modelling early may result in saving cost and time compared to practices that adopt the code and fix strategy.

Modelling may be perceived as costly and time consuming due to the rigour and education levels required (typically PhD level). In comparison manual code writing skills are cheaply available and easily trained. MBT capitalises on the skills disparity by allowing the cheap labour to detail the code model, while the engineer refines formal models to validate first the architecture then the implementation.

The intuition behind MBT is to build an abstract model of the environment (the parts that interact with the system) and the system under test (SUT), generate traces of paths over the model that will form tests, then utilise a test harness to apply the tests to the hand-crafted SUT and compare the results. This is a repeatable black-box testing process that is aimed at validating a systems behaviour and finding bugs.

There is no guarantee that MBT will prove that hand crafted code is error free, however it provides an extra layer of rigour and benefits similar to the practice of creating early prototypes of code in order to learn about the problem and potential solutions, but faster, without the expense and without the risk the prototype will be put into production.

MBT may be undertaken in an Agile, document light environment using user stories, as well as in Waterfall, a document heavy environment with formal requirements. In both cases communication and common understanding of the domain is tested with the act of modelling exposing any weaknesses. MBT modelling will be faster than the development team which must implement a great deal more detail, without dictating the implementation to developers.

The MBT engineer can become involved late in the software development life cycle, however there is still a need to understand both the domain and the requirements in whatever form they take. This is essential in order to model the SUT. Consequently, bringing an MBT engineer into the project late may be a false economy when the architecture and code base are not as flexible, early decisions made may not be documented and many of the found bugs will be reported at a late stage will be expensive to fix. However, there is no doubt that more bugs will be found.

3.5.1 Existing research

The book *Practical Model-Based Testing*, appears to offer the most comprehensive overview for this field [6]. While Neto *et al.* (2008) provide a broad overview of literature up to 2006/7 [5]. Commercial industrial tools exist for MBT, such as; Microsoft's Spec Explorer, which is packaged in Microsoft's Visual Studio; and Conformiq's Designer.

Model checking has been applied to various aspects of networking, for example, Canini *et al.* (2011, 2012) use model checking combined with symbolic execution in their NICE methodology for testing OpenFlow controller applications (learning switch, load balancer and energy-efficient traffic engineering). They found 11 previously unknown bugs in established software [8, 113]. However the NICE methodology does not extend to allow testing and comparing implementations of the models, such as an in-line load balancer with the SDN equivalent. Hoang *et al.* (2009) used Event-B and Rodin to model a network discovery algorithm [114]and Kang *et al.* (2013) modelled the OpenFlow Switch specification (v1.0) [115].

MBT in contrast has had little exposure to networking. Wieczorek *et al.* (2009) used MBT to test service-oriented architecture (SOA) integration from choreography models [13], but no other research has been found.

The closest paper to this research was published by Sethi, Narayana & Malik (2013) and describes model checking (again not MBT) an SDN controller based stateful firewall where the algorithm uses two switches to provide firewall properties [116]. The source for the two-switch firewall architecture was not provided.

3.5.2 Other testing methods

Black-box testing, is not concerned with how the system is implemented, merely that it generates correct results. This compares with developer efforts to validate their own code, which will include both white-box and black-box testing. Developer testing is important and must be undertaken, it is the first line of defence and the means a professional developer will use to ensure their code is of the highest standard.

The second line of defence is the test professional who performs manual black-box testing, perhaps with the help of scripted tests. By using domain knowledge and experience the tester working under time constraints is responsible for preventing critical bugs from escaping in order to keep problems in house.

Both developers and testers rely on intuition to determine when testing is sufficient. Test cases stop being generated when the writer feels there are no more benefits, runs out of time or runs out of ideas.

The last line of defence is reactive and involves triaging customer reported bugs. Many applications are in permanent beta testing with teams of developers rotating the role of maintenance and responding to customer bug reports. Failures in code quality at this level are managed not prevented.

The danger in relying only on developers and testers is that they both have an inherent level of experience and tend to adopt narrow views of the system, resulting in testing for problems they have experience of, have catered for and that fit with their interpretation. To be fair industry training for testers endeavours to break this pattern.

Developer testing is typically based on unit tests being run against the SUT, for example, Java JUnit tests. This is done as the system is developed, sometimes as part of test driven development. This rich source of tests however are often lost during maintenance, replaced by a smaller set that assumes the tests for surrounding code will capture any defects and anything serious or obvious will be caught by the manual testers. Time constraints play a key part, particularly in maintenance activities which are not perceived as adding value for

product owners. In comparison updating MBT tests only involves changing the abstract model and results in generating an entirely new test suite.

3.5.3 Modelling the software-under-test and its environment

First a model of the environment the software operates in is developed, in order to prompt and capture important aspects of its observable behaviour, then the software is modelled. This combined model can be reasoned over, mathematically proven and developed further as new insights or use cases are exposed or introduced. The resulting model is a series of states and transitions that are much smaller than the system is or will be.

It is not sufficient to model purely the software as this does not fully capture its purpose, its environment or its observable behaviours. Modelling the environment forces the software's purpose and the assumptions made about its properties to be made explicit. The environment is then able to prompt the black-box and observe the black-boxes subsequent behaviour.

The environment is often made up of a wide variety of equipment, people, other software, weather, etc. Understanding this environment well enough to create a formal model can be a challenge, it can be difficult to capture all these aspects formally. As a consequence the resulting model may therefore be only an approximation, leaving scope for unforeseen events that may lead to non-deterministic behaviour.

If modelling to a set of documented requirements, the model may be annotated with requirements identifiers which can aid in using a requirements driven test strategy when generating tests.

Modelling a sub-system (for example, a network function) may be preferred to creating a large model (for example, a network), the smaller size makes the generated test cases faster to run, it may provide a good abstraction for the larger system and it allows time for more rigorous test strategies. If the abstractions are carried through to the software artefact, it reduces software complexity, it can be worked on in isolation by developers, may have fewer bugs, be easier to replace and maintain and may consequently have a longer software lifespan. This approach demands actively looking for low level functionality that can be represented as black-box abstractions and led the author to question the dominant architecture used in SDN implementations of in-line network functionality [3].

3.5.4 The MBT test process

MBT extends testing by taking the engineer's interpretation of the system and contrasting it with the developer's interpretation. That is, the formal model is contrasted with the code model. The engineer also follows a process to enable the direct comparison of the two models or any other implementation, in order to determine behavioural equivalence.

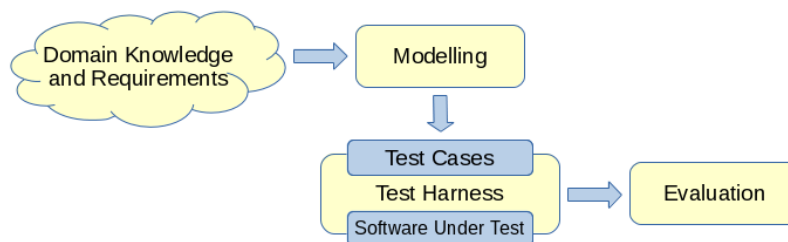


Figure 3.13: The Model Based Testing process

The stages of the MBT testing process is shown in Figure 3.13. The process is largely

linear, starting with domain knowledge and an understanding of the requirements or user stories. Model creation is the most challenging aspect while automatically generating a test suite is the simplest. These are followed by the technical challenge of creating a test harness that interprets the tests for the SUT and records the results. The process will typically result in the MBT engineer being ready to test the SUT before the developers have finished creating the more detailed software artefact to be tested.

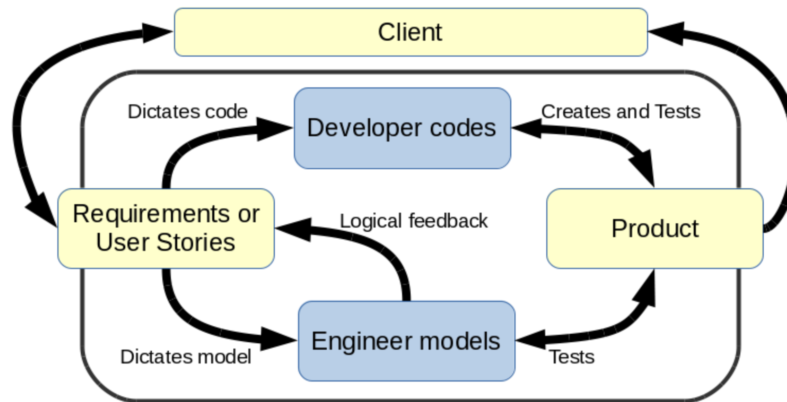


Figure 3.14: Incorporating modelling into the software development cycle

The software development process is shown in Figure 3.14 and encompasses both Agile and Waterfall style methodologies, the main difference being the speed and number of cycles seen by the client.

Creating the test harness requires some technical capability. The result of test case generation is a suite of tests which will not be directly executable on the SUT. The test harness must therefore transform this into executable tests. An additional complication when testing network functionality is that the test harness must replicate a network and generate network packets of various types.

In contrast to creating the model or test harness, test cases are easily generated. The practical limit becomes a temporal one in that the potentially thousands of tests generated for a system, combined with set-up and tear-down, may take an impractical time to run. One cause may be ‘state explosion’ which is explored further in Section 5.4.1. The underlying goal however is to not rely on developers limiting tests based on intuition. Tests are instead limited through the intelligent application of test strategies (see Section 3.5.5) designed to achieve specific testing goals.

The tests take the form of valid paths across the model, starting and finishing in the environment which prompts and observes the behaviour of the SUT as a black-box. Any paths that hold for the model can be expected to hold for the SUT and this idea is the foundation of testing.

Testing against the SUT will reveal errors in one of two places. The software under test or the MBT model. Finding errors in either place is considered normal — just as developers make mistakes, so do engineers — the key is that both parties are using very different methodologies and are likely to make different mistakes. In effect the project gets the benefit of developing two interpretations of the software, without the expense.

3.5.5 MBT test strategies

When the volume of tests generated become impractical to run in a reasonable time frame, test strategies facilitate creating test subsets that achieve specific and measurable outcomes.

For example, exercising every method, changing every state variable, exercising every conditional, traversing every common path, testing parameter boundaries, plus others. By considering the software being tested and the likely problems that may be surfaced, it is possible to use test strategies to generate a subset of tests that will create a high level of confidence in the software.

There are a large variety of test strategies described by Utting *et al.* (2010) which broadly fall into one of six criteria;

Structural model coverage-criteria This seeks to maximise coverage of the model in one dimension, for example;

- control-flow testing which tests statement, decision or path coverage such as if/else branching.
- data-flow testing which tests all definitions, all uses and all definition-use paths.
- transition-based testing which tests all states, all configurations where there are parallel states, all transitions, all transition pairs in and out of a state, all loop free paths coverage, all one-loop paths coverage, all round-trips coverage, all paths coverage.
- UML class diagram coverage which tests association-end multiplicity, generalisation coverage testing every subclass, class attribute coverage.
- UML sequence diagram coverage which tests start to end message paths.

Data coverage criteria This selects data to be representative of what is being tested, for example;

- testing all values may be impractical, while testing one value may have merits if it helps minimise the number of tests.
- boundary values, testing all boundaries, multidimensional-boundary coverage using the minimum and maximum of each value, all edges coverage testing the predicates applied to values, one-boundary coverage testing each predicate once.
- Statistical Data coverage, random-value testing.
- Pairwise testing, reducing the set of tests needed by pairing up values.

Fault model criteria Tests to demonstrate the absence of specific bug classes.

Requirements-based criteria Tagging the model with requirements tags, enables tests to be done that test the implementation of those requirements.

Explicit test case specifications Similar to manual testing, but scripted to match the auto-generated test scripts, fine grained control of the tests generated is enabled at a higher labour cost. It is possible to specify or restrict paths as desired.

Statistical test generation methods By adding probabilities to a FSM, a Markov chain is generated and can be used for testing the most likely paths.

Mixed tests The previous six classes of test families, cover a large number of tests. The optimal solution may be to mix and match testing strategies to suit the software under test in its domain.

3.5.6 State explosion

One often stated concern of generating tests from models is the state explosion problem, the capacity to generate tests in numbers large enough to render them impractical to run. This may be due, for example, to an infinite number of input variables between 0 and 1. Solutions may include using representative data or partitioning the system and testing subsystems first or choosing abstractions that minimise the state space. For example, testing a network function by testing every conceivable layer 3 Internet address will suffer from state explosion. The IPv4 address state space is 4.3 billion or $4.3x10^9$, making this a poor use of MBT. Finding security concerns such as back doors using port-knocking[9], potentially makes a good case for open source code and code inspection.

Firewalls can provide a good example of reducing state space using abstraction. Stateful firewalls may recognise three types of address space and two domains. Trusted, untrusted and dangerous addresses while the two domains are internal and external. Together these reduce the address state space from $4.3x10^9$ to 5 (not 6 as internal domains typically treat unknown hosts as dangerous/quarantined, see Figure 2.13).

3.5.7 Test metrics

The metrics a business uses to measure testing success, changes when MBT is adopted. The number of tests, passing or not, are no longer relevant as any number of tests may be automatically generated and reporting high numbers of irrelevant failures may be counter-productive. Model, requirement and SUT test coverage per test strategy, become the new preferred measures.

Being able to defend the test strategies used also becomes important. These strategies are repeatable, can be articulated and can be debated by a wide range of interested parties.

3.5.8 Defensible testing in industry

Advances in processor power and supporting technologies are enabling programs of greater complexity, including safety critical systems such as cars, aeroplanes and medical equipment [117]. As systems get more complex, the tendency is for them to also contain more errors, leading to disasters such as the Toyota Motor Corporation's unintended acceleration problems (89 killed), the Ariane 5 rocket disaster (see Figure 3.15) and the Therac-25 medical radiation disaster (6 killed) [118–120].

To senior stakeholders all software is a black-box and testing is ultimately done by the consumer, whether that be Alice applying the brakes in her Toyota or Bob the product owner launching the latest Ariane rocket. Confidence in quality is often gained through personal relationships and trust in the senior developers professionalism (not always the case — Toyota appears to have contracted out their brake and throttle software and applied minimal oversight), plus the software's time in the wild. This may be reassuring, but it may not be sufficient in the event of a disaster.

Software complexity is also making it more difficult than ever to prove, either before or after an accident, that all reasonable quality steps have been taken. The current industry practice is to rely solely on good developer practices. This can be problematic when developers lack skill, are under commercial pressure or know the code will never be examined. For example, Toyota's brake and throttle software was described in a NASA report as using

[9] Port-knocking is an access protocol that accepts a connection from a specific IP address after it receives packets sent to ports in a specific order. The state space for say knocking on 3 ports becomes $4.3x10^{(9x3)}$

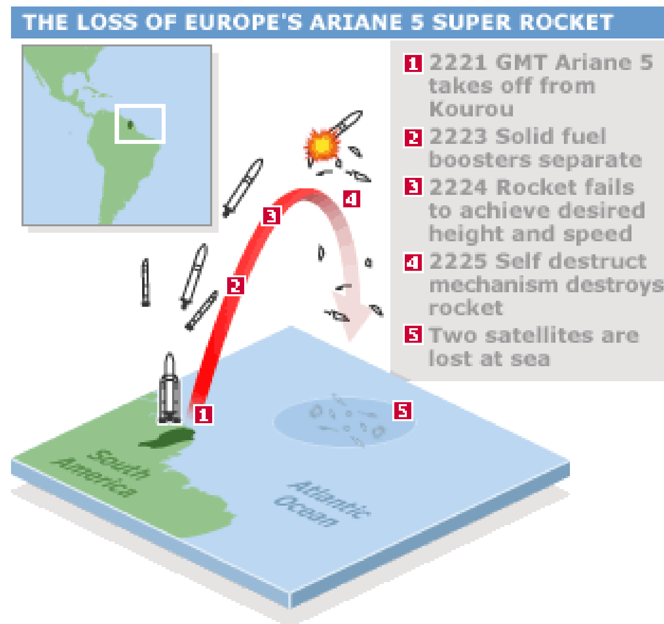


Figure 3.15: The Ariane 5 explosion, caused by a software bug, 4th June 1996.
Source: BBC http://news.bbc.co.uk/2/hi/science/nature/2634945.stm

a *proprietary developed coding standard* meaning it did not conform to industry accepted coding standards as reflected in the code inspection tools NASA used [118]. In later lawsuits it was described as spaghetti code by expert witnesses Michael Barr using metrics to back up the definition [10].

Corporate risk is typically managed through licensing. The following is an except from Microsoft's end-user license agreement for Windows 10 software[11] and is representative of the IT industry. For example, Xero[12] and most other software companies have a similar clause limiting liability to the purchase price or last fee paid. Bold added for emphasis.

> Limitation of Liability, paragraph four - Except for any repair, replacement, or refund the manufacturer or installer, or Microsoft, may provide, **you may not** under this limited warranty, under any other part of this agreement, or under any theory **recover any damages or other remedy**, including lost profits or direct, consequential, special, indirect, or incidental damages. The damage exclusions and remedy limitations in this agreement apply even if repair, replacement or a refund does not fully compensate you for any losses, if the manufacturer or installer, or Microsoft, knew or should have known about the possibility of the damages, or if the remedy fails of its essential purpose. Some states and countries do not allow the exclusion or limitation of incidental, consequential, or other damages, so those limitations or exclusions may not apply to you. **If your local law allows you to recover damages** from the manufacturer or installer, or Microsoft, even though this agreement does not, **you cannot recover more than you paid for the software** (or up to $50 USD if you acquired the software for no charge).

It is clearly cost effective in the face of a disaster, to limit liability where you can to the

[10]See presentation slides authored by expert witness Michael Bar (2013) for the lawsuit Bookout v. Toyota, slide 24. http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf

[11]https://www.microsoft.com/en-us/Useterms/OEM/Windows/10/UseTerms_OEM_Windows_10_English.htm

[12]https://www.xero.com/nz/about/terms/

purchase price of the software. But it has limits as a first line of corporate defence, consumer confidence can be dented by the minimalist corporate ambulance at the bottom of the cliff.

Given such protection it appears that corporate will to ensure safe code may only come about after many more deaths, lawsuits and fines. For example, Toyota's unintended acceleration has contributed to 89 deaths plus generated fines and remedies that as at 2014 has exceeded US$3 billion with ~400 injury cases still in talks [13].

When justifying the safety and security of software, defensible test processes will help, as will the adoption of defensible risk management strategies in determining when or if a bug should be fixed. MBT may have a niche in being conducted by an engineer who can generate reports on the the products fitness for purpose and an analysis of failing tests. This may also be subject to peer review in a manner similar to more mature engineering disciplines. This independence from the development team who provide the implementation may help senior stakeholders gain confidence in the product, in a defensible manner.

3.6 Applying MBT to networking

Networking might be characterised as dominated by practitioners that are tools trained (rather than trained on fundamentals), that follow best practice (the ideas of a few are popularised and adopted by the many) and are risk averse (prefer others to undertake development and implementation risk).

MBT offers a means to reduce risk, however MBT and networking are not natural partners. Network practitioners typically do not develop products and they are unlikely to be interested in developing models of products.

However many interesting functions in networking are highly developed and thoroughly tested in production environments. With the correct models and test harness there is potential for networking to utilise MBT as a push button technology generating useful results, for example, comparing implementation A with implementation B across a range of modelled behaviours.

This is new territory, MBT has not been applied to networking in industry. To better understand the problems MBT may face in networking, the next two sections outline the experience of multiple MBT practitioners in the software industry.

3.6.1 Industry Experience

There have been several studies and conference presentations describing the experiences of MBT practitioners in industry. They all report that MBT works, saves money and results in better code. However MBT is not 'sticking', in that the long term adoption and development of this strength is not happening in industry. This raises concerns that while MBT technical challenges may have been resolved, research that builds on these accomplishments may suffer the same fate. Five authors and presenters are now summarised to illustrate common experiences they share around industry engagement, which are then explored further in Section 3.6.2.

Robinson (2003) draws on his several years of MBT experience at Microsoft, to describe five obstacles around the mismatch between testers in industry and MBT [121]. These are; existing testers are not technical and "failed to meet the bar" for developers; testing is viewed as a back-end, ad-hoc activity; testers operate in a severe time crunch; formal requirements are rare; and current test metrics do not map to MBT.

[13] See presentation slides authored by expert witness Michael Bar (2014), KILLER APPS Embedded Software's Greatest Hit Jobs, slide 40. http://www.barrgroup.com/files/killer_apps_barr_keynote_eelive_2014.pdf

Stobie (2005) discusses that both testers and developers resist MBT [122]. Developers get frustrated with false positives — these are bugs that developers view as not "real bugs". This is made worse, in the eyes of developers, when false positives cannot easily be turned off. With testers, Stobie found they are resistant to change, do not have the underlying training to think or model abstractly, may prefer an ad-hoc approach and may claim failure when MBT fails to meet unrealistic expectations.

Hartman & Nagin (2006) and Greiskamp (2012) each have well over ten years experience using the technology with respectively IBM and Microsoft. Both claim the research into MBT is largely done [123,124] and they independently conclude the issues facing MBT now relate to usability — in that industry consistently passes the technology to testers who do not have abstract modelling skills. Testers who then acquire modelling skills, which are in short supply in industry, appear to then move on to more important roles.

Hartman & Nagin in their keynote speech at MBT 2006 in Vienna, describes [123]:

A 1999 study with a very experienced tester;

- tools are used by PhD graduates
- violent resistance by the tester
- I can do better by hand
- poor user interface
- achieved efficiencies for the testers, replaced much of the manual test case writing
- never used again

A 2001 study with three industrial teams;

- mountains of bugs uncovered
- 60% of the bugs were documentation bugs
- never used again

A 2003 study in industry;

- the Champion got a promotion
- never used again

Hartman concludes the barriers to be; tools are still bleeding edge; personell require higher education and sophistication; and there is a process shift to investing in testing up front. He expresses continuing confidence in MBT, but "keep the models away from (average) testers."

Greiskamp (2012) in a keynote presentation to MBT in Practice, observed that of 10 teams that adopted MBT over his time at Microsoft, despite excellent results, 7 later dropped it [124]. He further observes that the adoption is often bound to individuals who inevitably move on to other teams and roles. Greiskamp describes the Microsoft tool as "horribly complex" and containing "rocket science", while also stating that push button technology will not suffice.

Binder (2011) produced an MBT user survey to establish a profile of users [7]. From it he created a hypothetical composite MBT user, quoted in its entirety here:

I work for a large business organisation. We follow an agile process to develop the software we test. The system under test is embedded in another product sold to our customers. It is programmed in C++, runs on dedicated computers, is networked, and uses a Windows OS.

We felt MBT could help reduce testing cost/work and bug escapes. We use a commercial MBT tool that we've integrated with our design repository and test harness.

We've added MBT to the mix of our existing manual testing and traditionally programmed test drivers. Our effort for each testing mode is roughly equal.

At present, we're using MBT in a pilot project to test version 1.0 of our product. We taught ourselves to develop test models and use the tool by reading documentation and experimenting. It took us each about 100 hours of this self-study to become minimally proficient with the tool.

We use MBT to test system-scope functionality. The tool generates code and data for an adapter that abstracts the SUT APIs and then drives the APIs and evaluates results from the APIs. We measure requirements coverage for our test runs. The output of this tool is not integrated with our build.

We've had some difficulty developing test models. Sometimes our model "blows up" (combinatorial runaway) and we've found that our MBT test suites are not effective for certain kinds of bugs. However, this is manageable and not any worse than what we expected. With respect to other first generation MBT problems (updating, integration, and inadequate oracle), either we have not seen these problems or they're not obstacles.

In all, we think MBT has been moderately effective so far. For the most part, our expectations for both challenges and improvement have been met. MBT has reduced bug escapes about 60%, testing costs by 15%, and testing time by 30%. Among our co-workers, developers and managers are neutral about MBT but other testers view it as effective. Our users/customers are not aware of MBT. Overall, we are very likely to continue using MBT.

Binder's survey notes that Waterfall and Agile development processes are equally represented. Of the application domains 90% of respondents indicated domains where high reliability is expected, for example embedded controllers. Respondent answers to the biggest limitations of MBT tend to fall in the areas of; modelling is complicated, lack of industry and management support and commitment to quality, poor requirements and difficulty in acquiring the modelling skills — claiming approximately 100 hours of self directed study is required.

These five authors all discuss serious issues with applying MBT in industry. They can be summarised as:

1. Inadequate skill levels
2. Lack of management interest
3. Cost structure up front

4. Mountains of bugs
5. Cultural perceptions of Testing in IT
6. Complicated tools
7. Poor technology diffusion

3.6.2 Hurdles to adopting MBT for SDN

We have described the experiences of practitioners in industry. The technology works, produces good results but is difficult to use and widely misunderstood. We now discuss the seven issues that have been outlined.

1. Inadequate skill levels

The skills required for MBT, are not those of a programmer. The limited programming skills that are needed, can typically be sourced within a company and relate only to creating a test harness to enable the test suite to interface with the SUT.

Equally the skills required for MBT are not those of a Tester. A Tester takes their domain knowledge and a programmers work to devise a range of manual and possibly scripted tests to ensure the most obvious bugs are exposed before work is released. The goal of a Tester is to pre-empt and minimise the scale and impact of released bugs within a tight time schedule.

The skills required for MBT, are in modelling—the skill to decompose a target system and formally model sub-systems of interest using mathematical abstractions. This is a skill set that is not highly developed in either programmers or testers. Nor do firms typically seek to develop these skills in developers and testers, which instead appear to be concentrated in Engineers, System Architects and Business Analysts.

It can be argued that Programmers do in fact model. For example, in Agile practices a team may choose to model a problem; to better understand it, to define interfaces, experiment with solutions and define work breakdowns or sprint deliverables. This practice helps Agile teams to generate good design (principle 9 of the 12 principles of Agile)[14]. But Agile also values simplicity and working software over documentation (Principle 10 of the 12 Principles of Agile). Taken together, the ideal Agile modelling exercise can therefore be described as quick, dirty and throw away. Sufficient for the immediate purpose, but falling short of the rigour required for mathematical modelling.

In contrast to modelling, generating the tests in MBT is automated and trivial.

To summarise; Programmers and testers should not generally be expected to have the prerequisite modelling experience or training to easily succeed in applying MBT.

2. Lack of management interest

Obtaining champions within industry only goes so far. Several of the experiences above, discuss the fate of projects once the champion moves on. Relying on champions appears to have two flaws. Management may view them with scepticism — after all they have an agenda to push — and those capable of being champions for MBT perhaps tend to be better educated and more sophisticated than their colleagues which may lead to faster promotion or re-assignment to work that is more pressing in the eyes of management.

[14] Principles behind the Agile Manifesto. Accessed: Oct 2014. http://agilemanifesto.org/principles.html

This effect has its roots in two potential causes. The first is that testing is traditionally downplayed in IT, its goal is to catch the most obvious bugs and in today's market it is expected that most products are in continuous Beta testing. The imperative is to get the product into the marketplace, quality comes later — if there is market interest. The second is that highly educated employees are still rare, for example, the local Wellington employment environment appears to push students to value 2 or 3 year qualifications [125]. Consequently those with 4 or more years of education are rare and possibly considered wasted on what may be perceived as traditional testing.

Management buy-in to MBT might be evident when existing champions are replaced by new champions. In effect when management starts building an MBT businesses capability.

3. Cost structure up front

There is resistance to applying MBT up front. This appears to be related to management correlating MBT to the traditional testing model which is applied late. This is in contrast to when the modelling of software is most beneficial, which is at the start.

Applying MBT late hits problems related to sunk costs. Sunk cost is an term used in economics to refer to a cost that has already been incurred and cannot be recovered. Sunk cost may refer to, for example, capital expenditure, consultants, developer wages or project time. As sunk costs increase, a project becomes less capable of making changes. The architecture late in the project becomes difficult to change as does the code implementation. At the extremes, at the start of a project all options are open, in the final week all options are closed except polishing the product.

Applying MBT late in a project significantly reduces its ability to impact on project success. It may also mean that the architectural experience is no longer freely available, nor the domain knowledge. The documentation is often not rigorous enough, particularly in an Agile environment, nor up-to-date.

Weighing against this are potential stakeholder concerns that improved architecture and early bug fixes offer difficult to quantify savings versus the certainty of an engineer costs up front. It is perhaps a concious decision to delay testing of the architecture until after the code has been implemented, perhaps with the hope that any flaws will impact the projects budget less if dealt with later and the missed deadlines caused by problems that may have been picked up, will be inconsequential to the project's success.

What is the cost of checking abstract models up front? With architectural modelling expertise already employed, domain knowledge already available and the flexibility to easily adapt the model in the face of new information, the additional cost appears to be low.

4. Mountains of bugs

This is mentioned as a problem by Stobie (2005) and Hartman & Nagin (2006), but no reference was found exploring this in the academic literature. On the surface it appears to be a triaging problem with a persistent memory. It could be this is already resolved or not as big an issue as Stobie and Hartman expressed. For example, Utting *et al.* (2010) suggests a test strategy that allows selecting paths to ignore (see Section 3.5.5). Hartman's observation of 60% of the identified bugs being documentation bugs, may also be indicative of applying MBT late in the process when there is little perceived value in fixing the documentation.

5. Cultural perception of Testing in IT

A common denominator is that MBT is handed to testers in industry, who are sometimes perceived as akin to failed programmers [121, 122]. While programmers typically do not

view a dedicated testing role as a promotion.

Testers are generally non-technical people and this has led to calls for MBT technology to be reduced to push button technology, to suit their capabilities [121]. However on the face of it, this seems out of reach given today's technology.

Of more concern is that the overwhelming impression from these articles and presentations is that the perception of Testers and testing within IT is poor. Any association with testing appears to run a high risk of the technology being assumed to be of little use unless non-technical people can understand it, this may be a non-starter when the core skill is the technical skill of creating abstract mathematical models rigorous enough to be suitable for machine reasoning. The very name "Model-Based Testing" appears to re-enforce the perception that the technology "should" be accessible by non-technical people. Even worse, technical people associating with a discipline that contains "test" in the title, appear to run the risk of being stereotyped as non-technical. Until this changes, industry may find it difficult to attract technical people to the role. It may be even more difficult to retain them and the experience they have acquired.

6. Complicated Tools

The core of the complicated tools problem appears to lie with the issue of attempting to give non-technical people the ability to use tools that require PhD levels of education. This perhaps limits MBT testing by non-technical people to simple models, late in the software development life cycle.

Of the tools surveyed many are academic tools and several are niche tools. Two stand out as being both generic and in use in industry; Microsoft's Spec Explorer, which is packaged in Microsoft's Visual Studio; and Conformiq's Designer. Reviewing these two tools in depth may be an area for future work.

When comparing tools, the academic literature has not yet converged to an agreed set of criteria to measure MBT tools against, beyond merely listing the MBT tasks required of the tools [1, 126, 127].

7. Technology Diffusion

There are several models of technology diffusion to explain how a product or process like MBT gains traction and acceptance in the marketplace [128]. For example, the epidemic model adopts the notion that slow adopters merely hear about the innovation later. Underpinning this view is the idea that the speed of information flow dictates the pace of adoption and that this information diffusion is often driven through a word of mouth process. One hypothesis is that this is aided;

> "...where software knowledge is easily learned and transmitted, for populations which are densely packed and where mixing is easy, where early users spread the word with enthusiasm, and in situations where the new technology is clearly superior to the old one and no major switching costs arise when moving from one to the other." Geroski (2000) p607

This model of diffusion may be related to existing successful process improvements, for example Object Oriented methodologies or Agile processes. But where the similarity ends with regards to MBT, is that MBT's target market—Testers and perhaps Programmers;

- Do not typically have a background in abstract modelling,
- Are generally only interested in testing after code has been written and

- Need a 100 hour training investment to become minimally proficient with MBT.

3.7 Summary

Formal methods and modelling already play a role in some safety critical systems. MBT takes this a step further and generates tests from the model that can be performed on the SUT. Doing so can provide rigorous black-box testing of products. It is not the right process for every technology but does suit domains which are well understood or are safety critical.

We have examined three in-line network functions, NAT, load balancer and a firewall in order to draw out common features and have formally described an abstract network function. In the process we identify several interesting properties such as their being decoupled from the network, they perform packet forwarding and packet mutation, may hold state and are transparent to the network. Chaining network functions is common in practice, for example an industrial firewall, so we establish properties for chains of functions, again they are decoupled, forward and mutate packets, are transparent to users and in addition they must maintain a consistent order.

MBT is the process we intend to apply to network functions. We have discussed existing research where model checking and MBT has been applied to networking, some benefits of MBT and examined where MBT fits in the software testing ecosystem. Modelling the SUT is not a trivial task and the modelling tool used in this research, Rodin, and the modelling language Event-B are both discussed. The test process has been outlined and the technical challenge of creating the test harness. Test strategies aim to provide strategies for both test coverage and test metrics which helps direct the available test time where the volume of tests otherwise generated is large. State explosions are mitigated by test strategies, utilising abstractions and picking representative variables. Defensible testing is the idea that MBT testing is repeatable and can be replicated by an independent party. Test strategies can also be articulated to senior corporate stakeholders who may also utilise independent engineers to gain confidence in their product, in a way that generates a corporate paper trail.

Industry experience has generated excellent results from MBT, but several key problems discussed include; inadequate skill levels; lack of management interest; up-front cost structure; mountains of bugs; cultural perceptions of testing; complicated tools; and poor technology diffusion.

Next we discuss the research problem, what can be achieved by applying MBT to SDN and legacy in-line network functions?

Chapter 4

Research Direction

After providing background on networking, SDN, network functions, formal methods and MBT, these threads are pulled together and shape the research direction of this thesis.

The questions to be answered revolve around the SDN approach to network functions and questioning whether this approach is desirable and if not, why not. A number of SDN problems have been highlighted in the research conducted by others and are discussed in Section 2.7. In this research I intend to explore the idea of state divergence in SDN, in particular the convergence of network function state with end host state and how this is (or is not) managed.

4.1 SDN's third layer of state divergence

One of the goals of formal methods is to prove properties of the network, in particular the absence of black holes and loops and reachability between end hosts. To achieve this consistently with a dynamic distributed system is difficult.

SDN offers the promise of being able to prove these properties in the controller. The controller may hold a graph of the networks current state (or may provide switch statistics to an application), therefore it may be possible to test properties over this graph — perhaps even to test for flow rule violations before they are implemented [85].

However, background research has highlighted two layers of state divergence, potentially both small, verging on trivial. And this research finds itself focused on a third layer of state divergence, again potentially small and verging on trivial. None-the-less small compounding problems are greater than their sum and in any non-trivial networking environment small problems may occur frequently. Add to this the security concerns that arise when state divergence can be determined to be non-random.

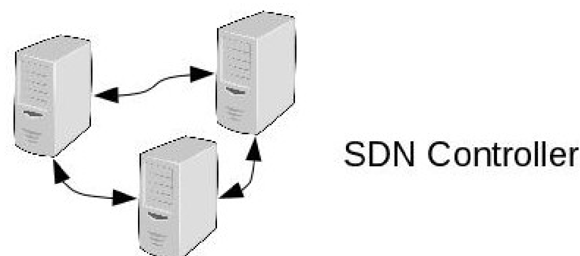


Figure 4.1: The first layer of SDN state divergence.
Distributed controllers holding different views of the forwarding plane

The first layer of state divergence (Figure 4.1) is within the logically centralised distributed SDN controller. This controller is an example of a distributed database (holding network state) where in order to provide redundancy in the event of controller failure, controller state must be shared amongst multiple instances of the controller. This raises problems related to controller state convergence, a concept that is widely explored in distributed databases. Controller state divergence is when two controllers hold different views of the network. Several strategies may be adopted to ensure state across controllers is consistent, however the most common strategy accepts what is hoped to be a small amount of divergence and is referred to as *eventually consistent*.

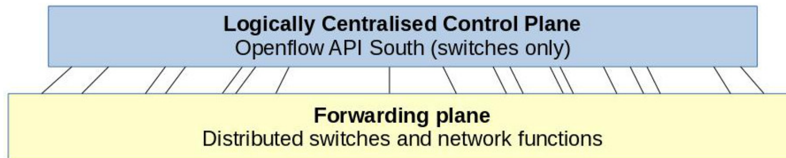


Figure 4.2: The second layer of SDN state divergence.
Forwarding plane switches not in the same state as the controller view

The second layer of state divergence (Figure 4.2) arises between the control plane and forwarding plane with research demonstrating update delays in the forwarding plane; beyond what might be expected for latency. In effect, the distributed (and *eventually consistent*) controller may be making decisions based on a view of the network that reflects instructions sent, not instructions implemented.

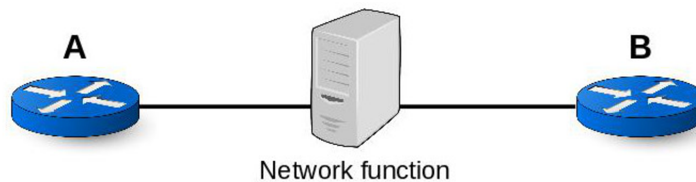


Figure 4.3: The third layer of SDN state divergence.
Network functions not holding the same connection state as end hosts

The third layer of state divergence (Figure 4.3) may arise between network functions and end hosts. Network functions may deduce connection state from flows passing between hosts. This may prove impractical where the network function is stateless or the algorithm and state are moved to a remote third party such as a SDN controller, meaning that although the network function is observing the flows it may not be able to respond in a timely fashion, or at all.

The failure to manage these three layers of state divergence may lead to further divergence and/or the failure of desirable network properties such as no loops, no black holes and reachability.

An intriguing question is whether there is a fourth layer of state divergence between the SDN controller and management applications, however this is not explored further in this research.

4.2 The hypothesis

The hypothesis is that the lack of state convergence between network functions and end hosts may cause security problems, providing reachability when the SDN controller and

end hosts do not expect it; a flaw that may be exploited by an attacker.

Formal methods provide a way to explore these issues, using formal models and MBT to test for expected behaviour in an implementation. An MBT tool will also allow comparisons between implementations, for example an in-line implementation versus an SDN implementation, to the extent the behaviours of interest are captured in the model. If in-line firewalls pass tests that SDN firewalls fail, that may prove to be interesting and possibly demonstrate support for the hypothesis.

4.3 Implementing MBT to test the hypothesis

The next chapter discusses creating a formal model for a firewall which may then be used to automatically generate tests. Potentially thousands of tests may be generated, tracing all paths and testing all variables, branches, loops and more. The mechanistic generation of tests may ensure all cases can be covered and ensure code coverage is high, perhaps up to 100%. A practical limitation is the time required to run the tests and test strategies allow the potential to create a sub-set of tests to meet specific and measurable test criteria within the time frames available.

A test harness is created that takes those tests and applies them to an implementation. The result of testing the implementation may then be compared to the generated test script and any differences analysed.

The technical challenge in this research is the development of the prototype formal model and the prototype test harness. Once completed, using such an automated tool to conduct black-box tests on an implementation, is relatively straight-forward.

Ideally we will generate a range of tests from a formal model of a stateful firewall and be able to test an implementation of an in-line stateful firewall. We expect the in-line stateful firewall to pass all the tests. We will then test an SDN firewall which we expect will fail some tests, in particular tests relating to state divergence between the firewall and end hosts.

Chapter 5 discusses the creation of the model, the tests, the test harness and finally the results.

Chapter 5

Applying Model Based Testing

Automated black-box testing is a reliable and repeatable method of testing a code object's behaviour. However within the networking field it requires first that the formal model represents a commonly accepted ideal. Second that a test harness exists that can accept a test script and generate tests for an implementation.

A stateful firewall was chosen as the example network function as it involves an algorithm and dynamic state that SDN researchers suggest (see Appendix A) should be in the control plane rather than the forwarding plane.

5.1 Generating a formal model of a stateful firewall

References that describe stateful firewall behaviour include RFC's, standards from NIST and textbooks [15, 78–83]. Given the complexity and variety of firewall behaviour described by these resources, the model developed during this research is best viewed as a prototype rather than the ideal. Within the scope of the behaviour modelled it is believed to be accurate, however there are a range of behaviours that still need to be added. None the less it is a starting point.

The model is created using stepwise refinement which is moving from abstract concepts to concrete behaviours incrementally. It is also under constant revision as the engineer gains a better understanding of the model's environment, the SUT's (Software Under Test) functionality and how the environment prompts and observes that functionality.

The model is comprised of two parts; a model of the network environment and a model of the firewall. The model of the network environment is considered first.

5.1.1 The network environment

The environment allows modelling a network's interactions with the SUT without having to consider immediately the effects of the SUT's behaviours on the environment. The aim is to simplify the modelling task and instil an element of modularity. The immediate goal is to provide an environment suitable for a firewall, later it is intended that the environment be extended to allow other models to be created, such as a NAT or load balancer.

Modelling is an iterative process where the engineer's understanding of the domain grows and leads to adopting better modelling abstractions which are conceptually simple, cohesive and easily adapted.

Each refinement reflects a meta abstraction and is labelled according to a main feature;

1. Domains

2. Choke point
3. Protocols
4. TCP start and finish
5. Packets
6. Trust domains
7. TCP attacks
8. Target state

Domains describes the internal and external domains and contains events that pass packets from one to the other. The **choke point** constrains all packets to passing through a specific event, ultimately the firewall will occupy this point. **Protocols** introduces UDP and TCP as the sample protocols while **TCP start and finish** describes the TCP handshakes. **Packets** introduces unique packet IDs which allows the model to keep packet state, for example TCP flags and source and destination nodes. **Trust domains** describe the firewalls view of the network as trusted or dangerous. Any nodes not specified are considered unknown (stateless firewall) or untrusted (stateful firewall). **TCP attacks** introduces two simple attacks utilising SYN and FIN flags plus tests firewall state convergence with the two end host's state. Finally **Target state** is utilised by a model checker to indicate an end point to a valid path transitioning across the model.

Each machine is described in more detail next. Referring to the full Event-B model in Appendix B may be useful.

1. Domains

The context for this initial machine describes two domains $\{internal, external\}$. All packets in the model will travel from one domain to the other, for example, if a packet's src (source) is $internal$ then dst (destination) is $external$ or vice-versa. The null host 0 belongs in neither domain and a packet with only one null address field is malformed. Packets must be a well formed packet where $(src \neq 0 \wedge dst \neq 0)$ or the null packet where $(src = 0 \wedge dst = 0)$.

Any packets that do not cross between the domains are ignored by this model.

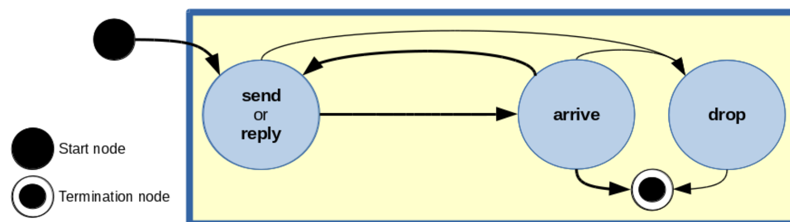


Figure 5.1: An event machine for sending and receiving network packets.

The machine establishes the events shown in Figure 5.1 which describes changes in packet state within the network. There are two events for sending packets $\{packet_send, packet_reply\}$ and the events $\{packet_arrive, packet_drop\}$.

This first machine abstracts away a lot of detail to achieve simple network packet passing behaviour. Each machine that follows will build on the previous, detailing the abstractions sufficiently to accurately represent first the environments impacts on the SUT, then replicate the black-box behaviour of the SUT.

2. Choke point

The first machine is not concerned with which path the packets take, the notion of paths is abstracted away and there may be one or many paths the packet may take. This machine refines the first (making it incrementally less abstract) by adding a path event that all packets must pass through. This reflects that firewalls are typically placed on a chokepoint to ensure it sees all traffic between two domains. Ultimately the firewall SUT will occupy this point. Figure 5.2 shows the new FSM, where every packet traversing between the domains is seen by the chokepoint.

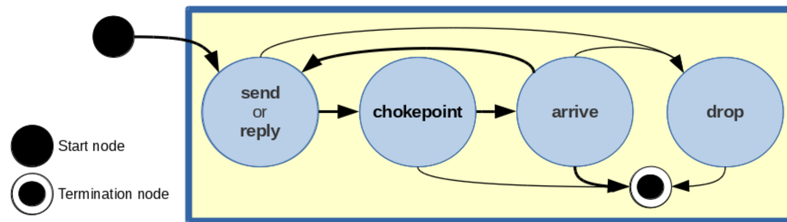


Figure 5.2: An event machine for sending and receiving network packets.

3. Protocols

Here two representative network protocols are represented using events. The UDP event is relatively simple, meaning $protocol_UDP$ is complete (see Appendix B). While the TCP event $protocol_TCP$ names events and local variables that will be used in the next refinement which describes the TCP protocol in greater detail.

4. TCP start and finish

This refinement defines the three TCP events that enable a TCP session to start, continue or finish.

The sequence of packets forming TCP's three-way handshake, is modelled in $protocol_TCP_start$ and $protocol_TCP_finish$. The $packet_reply$ event is adapted to recognise and reply to the various TCP packets.

The TCP fields of SYN, ACK and FIN are boolean flags $\{0, 1\}$ as they are used by TCP to open and close TCP sessions. A stateful firewall will also use these to determine when to add and remove rules for TCP flows.

To associate flags with a given packet, a unique $packet_id$ is defined which allows defining sets of ($packet_id \mapsto value$) to define header fields such as the SYN field. For example, if the first TCP packet generated has a SYN field with a value of 1, the model will append ($1 \mapsto 1$) to the set of ($packet_id \mapsto value$) called pkt_tcp_SYN.

On reflection, $packet_id$ was placed here for convenience rather than a sound design decision. A future iteration of the model may see $packet_id$ introduced immediately after (1) Domains, along with the machine for (5) Packets.

5. Packets

Uses the unique $packet_id$ number to allow a packets source and destination address to be recorded for a packet. Initially this was not required as no packet state was kept, however this became important when the firewall was modelled as it keeps state, represented by $NODES$ and $PROTOCOL_TYPE$. For example, there is a set of ($packet_id \mapsto NODE$) called fw_src_ip.

6. Trust domains

Describes the trust domains used by a firewall as $\{trusted, \ untrusted, \ dangerous\}$. Both the trusted and dangerous domains specify hosts. The untrusted domain is the set of all remaining hosts. As seen in (the earlier) Figure 2.13 all internal domains are either trusted or dangerous (alias quarantined). External domains may be in any one of the three domains.

7. TCP attacks

The initial environment modelled well behaved hosts and the packets they generate. This machine was added later, after the firewall was modelled to handle good behaviour, in order to represent badly behaved hosts which will generate traffic that also needs to be modelled.

Some important bad behaviours are not modelled, for example, the effects of a DoS attack which achieves its goals through resource depletion — running the host out of memory (RAM) or overloading its buffers. To model resource depletion requires modelling hardware in addition to software which is achievable but considered out of scope for this research.

The events modelled in this machine include SYN packets used for flooding attacks and FIN packets used for reconnaissance. It also models attempts to exploit poor state convergence between the network function and end hosts. In particular it attempts to continue to use a closed TCP connection, allowing an external host to potentially keep the connection alive.

8. Target state

To ensure generated test paths do not terminate within the SUT, the environment model defines path termination points in the environment where they may be observed by an outside observer.

5.1.2 The firewall model

With the environment modelled, meaning it is possible to transition states representing packet creation and forwarding through the network, the firewall can be created.

> "If I had more time, I would have written a shorter letter." Blaise Pascal (1657)

The firewall consists of two machines, the first names and sets up the events within a firewall while the second contains the detailed math. The stateful firewall machine is very complex and as observed earlier this is not ideal for future expansion and maintenance of the model. The quote above attributed to Blaise Pascal, seems very appropriate. Future iterations will seek to split the stateful firewall machine into several machines.

1. Firewall events
2. Stateful firewall

1. Firewall events

This first machine sets up the event framework for the firewall. Six events are specified to reflect a FSM (see Figure 5.3) where packets are accepted, firewall state may change and packets are ultimately either forwarded or dropped.

Time is represented in the firewall model using a ticker that increments each time the firewall event is entered. This enables the model to simulate the passing of time (more correctly of events) which is used to expire old firewall rules using a time-out field.

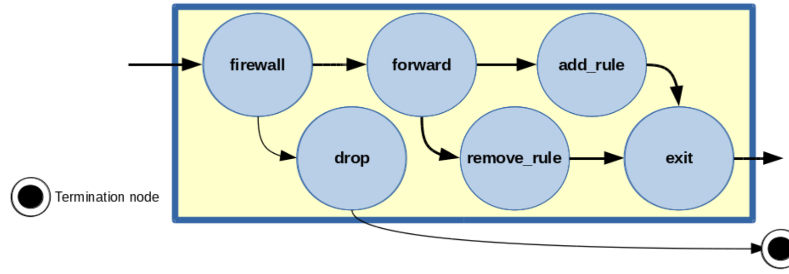


Figure 5.3: An event machine for a stateful firewall (occupies a chokepoint).

2. Stateful firewall

The example formal math that follows is from the $firewall$ event and in explaining it, may assist with following other examples. The full model, including definitions, is provided in Appendix B.

Here the global boolean $fw_forward$ is set. It is used as a predicate in the event $firewall_forward$ (see Figure 5.3) where if $fw_forward = true$ the event may be transitioned to, if $fw_forward = false$ it cannot be activated and instead the transition is to $firewall_drop$.

1.	$fw_forward := bool((src \in trusted \land dst \notin dangerous) \lor$
2.	$($
3.	$src \in ext \land src \notin trusted \land$
4.	$(\exists\, ident,\ protocol \cdot$
5.	$ident \in fw_rules\ \land$
6.	$protocol \in PROTOCOL_TYPE\ \land$
7.	$ident \mapsto src \in fw_src_ip\ \land$
8.	$ident \mapsto dst \in fw_dst_ip\ \land$
9.	$ident \mapsto protocol \in fw_protocol\ \land$
10.	$packet_id \mapsto protocol \in pkt_protocol$
11.	$)$
12.	$)$
13.	$)$

Table 5.1: Firewall forward predicate

The function returns true if the first line (1) is true, where the source is trusted and the destination is not dangerous. Otherwise it returns true if the source is external, not trusted (3) and there exists in firewall state an existing firewall rule (*ident* = rule id) (5) that has this packets details (4-10), essentially the packet's five tuple of $\{src,\ dst,\ protocol\}$ where *src* and *dst* are abstractions of IP:port.

The six firewall events are detailed in this machine and are briefly described next.

Firewall replaces the chokepoint event established in the environment model. This locates it on a path segment between the two domains where it will observe all packets exchanged. It identifies if a packet should be forwarded or dropped by referring to its ACL (*Trusted* and *Dangerous* hosts) and dynamic rules, then sets the appropriate predicate to either forward or drop the packet.

Forward examines the packets protocol and TCP flags if any. On seeing a UDP or TCP SYN packet arriving from a trusted host for the first time, the predicate for $firewall_add_rule$

is set. If the existing rule times out or on seeing the TCP closing handshake the predicate for $firewall_remove_rule$ is set. If neither of the above apply, the predicate for $firewall_exit$ is set.

Add rule adds a rule to the firewall's dynamic state. This takes the form of a five tuple, for example, $\{packet_id \mapsto src \in fw_src_ip,\ packet_id \mapsto dst \in fw_dst_ip,\ packet_id \mapsto protocol \in fw_protocol\}$ where *src* and *dst* are abstractions of IP:port. By adding to the firewalls dynamic state, this rule will allow packets from external untrusted hosts that are replying to internal host requests. The predicate for $firewall_exit$ is set.

Remove rule removes the rule (the five tuple) from the firewall's dynamic state. The predicate for $firewall_exit$ is set.

Drop forwards to *packet_drop*, a state in the environment that can observe and record the behaviour, in this case the lack of an output packet. The predicate for the environment event *packet_drop* is set, this ensures the environment can "see" the result..

Exit sets the predicate for the environment event *packet_arrive*.

5.1.3 Creating test cases

The first completed model of a firewall in Event-B was applied to the modelling software ProB. The model performed stateful firewall functions including dynamically altering state to allow external untrusted hosts to reply to internal trusted hosts. It did not include attacks from the environment nor did the firewall remove rules on observing a TCP closing handshake, but did it close sessions due to timeout.

It was expected that the MBT plugin for Rodin[1] would allow the generation of MBT test cases from the Event-B model. Unfortunately it was found the plugin (developed for Rodin 2.0) would not work for Rodin 3.2. Reverting to earlier versions of Rodin introduced problems with the model developed so far and in order to keep moving forward ProB was adopted instead for test generation.

ProB is an open source model checker that integrates with Rodin and accepts Event-B models. It is a stand-alone application that allows visualisation of the model and provides model checking strategies independent of those in Rodin. ProB was not required for its model-checking or visualisation — Rodin was sufficient in both cases — but for its ability to explore the state space of Event-B models and generate traces (that will form a test script) in the form of an XML file (see C.1). However, it was discovered that ProB only generates a breadth first exploration of state, terminating when all transitions have been traversed at least once. Consequently we had only 14 test cases (see Appendix C.1) which do not cover all possible paths. A search for other model-checkers that can accept Event-B was conducted, however, while research papers were found, no other Event-B compatible model checkers were found.

Writing an extension for ProB or addressing the problems within Rodin's MBT plugin may be required to allow the use of the strategies covered in Section 3.5.5, but this will be left for future work. While not ideal, the test cases generated are sufficient to continue with developing the test harness and they may be supplemented with targeted tests.

[1]http://wiki.event-b.org/index.php/MBT_plugin

5.2 Creating the test harness

The test harness is custom code created for this research. It is written in C with the test analysis server written in Ruby. It is intended to replicate a network environment, reading from the XML test script and create traffic to be passed through the SUT. It accumulates state regarding the tests and generates a web page reporting the tests pass, fail results. The roles performed by the test harness are broadly:

1. Incorporate the software under test
2. Read the XML test document
3. Generate and send network packets
4. Receive packets and reply
5. Accumulate XML state
6. Analyse and display results in a web page

To accomplish these roles the test harness needs to be able to record state as the tests progress and aggregate that state into an XML document. These were the first two problems considered, neither are trivial given this test harness is a distributed system over a network.

5.2.1 Recording state

This is the ability to record state as the test harness processes the test document, generating and responding to generated packets. The SUT has its behaviour deduced by the environment — if a test prompts the expected behaviour, it is reasonable to assume the SUT followed a process equivalent to that represented in the model. Cheating such a system is difficult as the cheat is still required to demonstrate the behaviour the model expects.

Not all state is easy to capture, for example, the TCP protocol's opening and closing handshakes proved difficult. These protocols are automatically completed between the hosts without any opportunity to record the states created by the protocol. However while this allows the hosts to abstract this detail away, the firewall SUT will observe every packet in the handshake.

The naive first approach was to use socket programming in C, specifically raw sockets which allows the creation of custom packets that may be used to create the handshake sequence. By creating and responding to the packets individually, state may be recorded at the same time. This failed as it was discovered the custom TCP protocol algorithm was superseded by the faster responding OS kernel's implementation.

The behaviour observed (see Figure 5.4) is complicated. It involves race conditions and packet sequence numbers that must be matched for a packet to be accepted. The image above shows a blue path that is started in application space and taken over by the kernels on both hosts (normal behaviour). Each kernel will see the current packet before any application and they also respond faster than the applications. The diagram indicates a TCP session is successfully created. However, the responses from application space are rejected. The first because the recipient will base sequence numbers on the first accepted SYN/ACK packet, meaning the second will be incorrect. The second response is based on seeing Host B's SYN/ACK, the resulting ACK packet is therefore a copy of the ACK packet already sent by the kernel and is rejected because the sequence numbers are the same (sequence numbers need to advance).

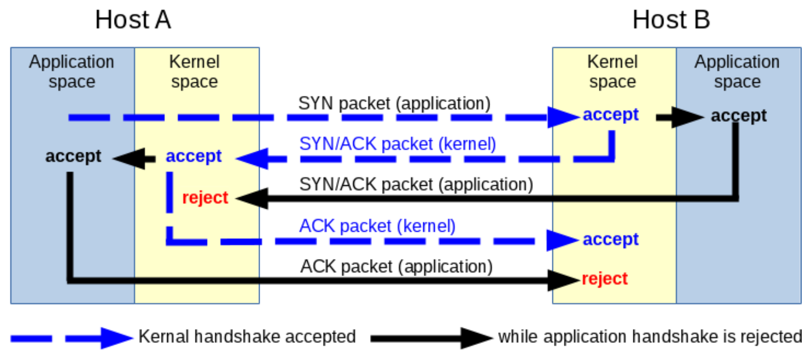


Figure 5.4: An application racing the kernel to establish a TCP session, will fail.

That TCP protocols are are processed in kernel space was a surprise. Kernel space is used for security and efficiency reasons, however this means that while successfully starting a TCP session can be logged, the individual packets used to establish the session cannot. This is a disconnect as a firewall operates on the individual packets and the formal model developed recreates the entire handshake.

Forking the Linux kernel[2] was considered. Hacking the kernel may provide the ability to record state from message passing as it built up in the TCP handshake. Investigation however showed a potentially steep learning curve and long build cycles involving recompiling and testing an operating system.

Also considered was SystemTap[3], a system that places listeners on kernel events of interest that will then generate a report based on requirements detailed in a scripting language. The main benefit of SystemTap is that it can be run against an already operating kernel and there is a library of scripts that work with network functionality. However, it requires root privileges and creates security issues (for example, it may expose passwords). It is possible this may work within VMs, however the approach was abandoned as the prototype applications were being developed on university systems.

As a compromise, it was decided that the successful instantiation of a TCP session could be proven by the subsequent passing of TCP traffic. However this means the prototype test harness loses detail during TCP handshakes, detail that may be useful when analysing failures. This may be an area for future improvement.

5.2.2 Aggregating state

Within the test harness the distributed end hosts create test state. Three state aggregation strategies were considered; collecting state from each end host as a separate process, perhaps using follow-on packets; using a back channel to aggregate data; and aggregating the data on the test packets.

Collecting state as a separate process would slow the processing of the test script. This might be mitigated by collecting all the state after completion of the tests, but still leaves the problem of establishing which state belongs to which test — test details may need to be carried in packet data.

A back channel, or communications direct with the test generator offers speed advantages in that processing the tests and gathering results could be performed in parallel — perhaps also requiring test details to be carried in packet data to enable matching results

[2]https://github.com/torvalds/linux
[3]https://sourceware.org/systemtap/

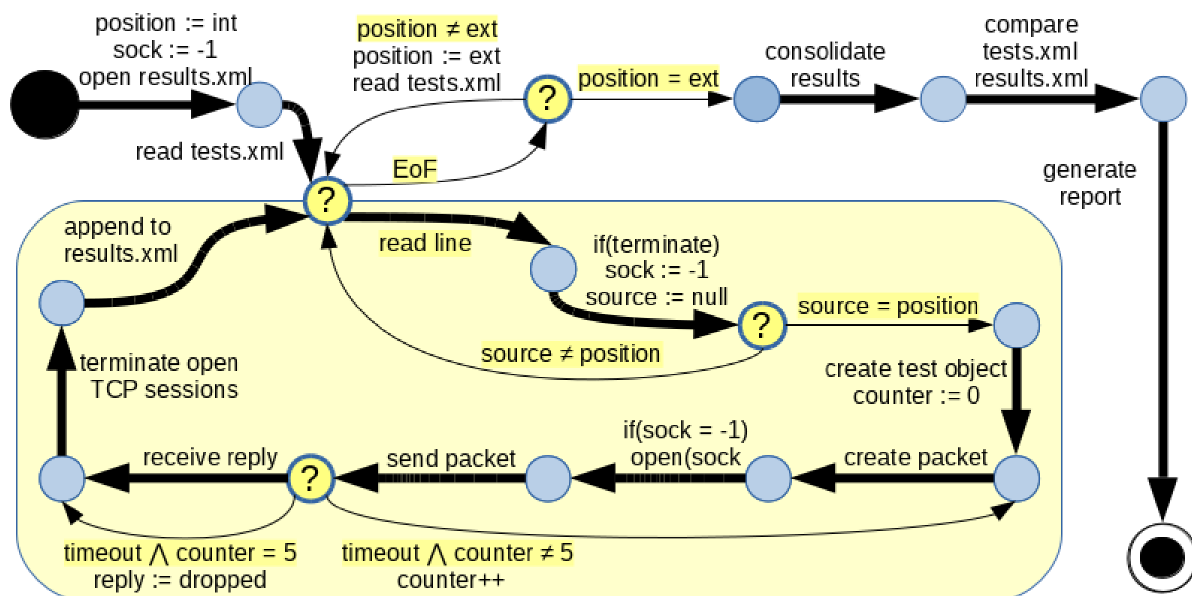


Figure 5.5: Sketch of MBT test server algorithm

with tests. This may be explored in future, but it adds complexity compared to the third option.

The third option of appending test state to packet data as the test is conducted was chosen due to the ease of implementation. However, one consequence of this approach is that a packet lost on returning to the test generator will include the test state appended by the response server. This may make it difficult to determine the cause of some failures and is an obvious weakness of this approach. At present the test analysis only determines pass or fail. More sophisticated analysis may also require more sophisticated data collection.

A fourth option became apparent later, after option three was implemented. Ultimately the test harness consists of multiple VMs hosted on one host. That host can provide a shared file for all hosted VMs, with read/write access. This shared file can then be used for aggregating test results. This may resolve the issues resulting from the approach taken and is an area for future improvement.

5.2.3 Test-harness work flow

Before coding was started the work flow of the test generator was sketched out to guide and better understand its functionality. This is illustrated in Figure 5.5 with transitions noted in pseudo code and branching conditions highlighted.

The work flow starts with the test generator creating a *results.xml* file, then opening the *tests.xml* file and iterating through the tests, actioning any starting from an internal server. Once these are complete, the algorithm loops and the test generator actions all tests starting from an external server.

Once both loops are completed the results are aggregated, sorted and compared before presenting the results on an html page.

The test generator algorithm broadly followed this plan, the only part not implemented was the time-out when awaiting replies — a timeout may be caused by either the packet or the packet's reply being dropped, in both cases the firewall behaviour would be recorded as *<dropped_packet>*. This was not implemented as the controlled environment meant time-outs due to networking problems do not occur.

5.2.4 Test and response servers

Three languages were considered for the test generator and response servers, Python, Ruby and C. The three are used extensively in the networking discipline and the author is familiar and comfortable with all three. C was chosen because of its low level access to socket programming and its capacity for creating and sending custom packets utilising raw sockets. While attempts to use raw sockets to replicate the TCP handshakes were not successful this allows, for example, the test generator to send multiple SYN or FIN packets outside the normal TCP protocols to simulate network attacks.

The test generator and response servers are written to run as terminal applications and include a custom utility library of scripts with over 100 unit tests. This also involved writing a unit test framework to operate in a similar manner to JUnit or Ruby tests. Bash scripts are utilised to compile the C source code and instantiate each server with supplied or default parameters.

The test generator functionality was added incrementally, starting with basic messaging between test and response servers using both TCP and UDP. Reading the file *tests.xml* was then added and the capacity to generate packets from the test script. As the test generator successfully generated packets and populated the data payload with test state, the response server code was extended to recognise the incoming packets and respond with additional test state appended to the data payload.

Both the test generator and response servers share code where possible, both compiling from the same set of C files. This includes the utility library written for the project and common definitions such as the state language used to describe state transitions both the tests to be conducted and the outcome of those tests.

The response server is designed to play five roles; it responds as an internal server which is trusted or dangerous (alias quarantined) and as all three of the external server types, trusted, untrusted and dangerous (see Figure 2.13 in Chapter 2). It is intended to be replicated five times in the test harness to perform these five separate roles, each with a unique IP:port address. Each response server maintains an open port that listens for UDP and TCP traffic from the test generator. On receiving a packet it copies the state recorded in the packet's data payload and appends new state summarising the actions it has taken before attaching it as packet data to the reply.

5.2.5 TCP's unexpected behaviours

Initially the test harness suffered from slow and inconsistent behaviour due to the volume of tests using the same IP:port addresses. It became apparent the TCP protocol was not keeping up with the tests being run.

TCP is a complex protocol designed to provide reliable communications over an unreliable network. In the face of packets and connections being lost, the protocol seeks to seamlessly recover using, for example, timeouts or wait periods. When attempting to utilise TCP in non-standard ways (for example, writing tests that only pass if the packet or connection is dropped), unexpected behaviours surfaced.

Utilising the network tool *netstat* on the response servers started to reveal the problems, the first of which was the ~120 seconds wait incurred as TCP attempts to recover lost connections, the **TCP connection retry time-out**. The second occurs if the connection has not been closed correctly where the TCP ports enters the **TIME-WAIT** status for a period (another ~120 seconds), waiting for the packets that will allow the connection to properly close. Given many firewall tests are testing to ensure packets and connections are dropped, this generates TCP problems that must be resolved before the next test can be run. Not doing so led to the unexpected behaviours observed.

TCP connection retry time-out

Tests that should result in dropped packets, will reflect a loss of TCP connectivity (in the open network, other network causes for dropped packets exist which are not examined further). The OS will then attempt to re-establish connectivity.

The connection retry period is controlled by the kernel operating system, in this case Ubuntu v14. It may be reduced by minimising the kernel variable *net.ipv4.tcp_syn_retries* from from the default value of 6 (~120 seconds) to 1, reducing the retry period to ~3 seconds. The test harness is a controlled environment and it is considered the faster time-out will not adversely affect the accuracy of the results. However this still incurs a ~3 second delay for each test that results in a dropped TCP connection which extends the time needed to run a suite of MBT tests.

Future work will address issues that may arise when using the test harness over the Internet, such as the impact of lost connections and latency delays that because of the setting adopted, if over ~3 seconds would cause a test to fail.

TCP TIME-WAIT

TIME-WAIT is a TCP state entered into by the host that initiates the close. In effect it stays in this state and delays finishing the close by $2MSL$ (maximum segment lifetime — an arbitrary value selected by the OS developer, Linux has adopted 60 seconds).

The waiting period for closing a TCP port is influenced by socket settings established when the port is opened. Initially socket settings were used to reduce this period to 1 second however it was found that while the socket was closed, the TCP ports still remained open. This was observed using the Linux netstat tool which enabled open processes to be identified by PID (Process ID). Despite killing the process, the socket continued to remain open for ~120 seconds in the TIME-WAIT state. Normally this allows late packets to be processed, however the test harness has no need for this behaviour and worse, the following tests displayed inconsistent behaviour.

After a review of the failing cases it was found that the underlying problem was tests that did not include steps to close the TCP session. Given earlier solutions were not working, a housekeeping algorithm was added. This identifies those individual tests that start a TCP session and deliberately fail to close it. Housekeeping then closes the session once the test completes to avoid disrupting the following tests.

5.2.6 Analysis Server and Results Presentation

The analysis server is written in Ruby, chosen because it offers easier processing and analysis of text strings than either C or Python. It takes as input the two XML documents representing the tests and results. It compares the two and establishes how many tests were passed. These results are then written to a JSON file detailing the test name and the pass, fail result.

This allows the possibility of analysing the two documents further and providing possible reasons for why a test failed. Time constraints prevented deeper exploration in this area.

Ruby is also used to present the results, using the Rails web framework. It presents an html page with embedded Javascript that uses AJAX to retrieve the JSON analysis file and create a pie chart showing the pass and fail test results.

5.2.7 Discussion

At this point we have implemented three servers that can be hosted on multiple terminals on one host computer. The test generator parses the *test.xml* document and masquerades as various hosts as needed by the tests. The response servers respond as trusted, untrusted or

dangerous hosts and are identified using port number; localhost:6000 for the trusted server, localhost:6600 for the untrusted server and localhost:6660 for the dangerous server. The distinction of internal versus external hosts is meaningless without a firewall however this did not otherwise prevent testing of the algorithms.

When running the MBT tests roughly 50% failed. This was expected as the firewall component that enforces reachability constraints was not present. To address this and incorporate the first firewall now required porting the servers and firewall to a network of virtual machines.

5.3 Networking the test harness

Three options for creating the networked test harness were considered. Mininet which is often used in SDN networking experiments; VirtualBox which would require some configuration to enable networking between VMs; and a hybrid of the two using Mininet to route to VMs.

Mininet is commonly used to test SDN networks and advice was received that Mininet utilises VMs for each host, therefore these should be able to host network functions. However an extensive online and literature search failed to find examples of Mininet being used natively to create this type of test bed for network functions. One solution, describes a custom add-on to Mininet designed to facilitate hosting network functions [129].

In contrast VirtualBox has extensive documentation and tutorials explaining how to set up networking between VirtualBox machines. The third option of creating a hybrid was considered but rejected in favour of a single approach using VirtualBox. It was hoped this would reduce test harness complexity.

5.3.1 Virtual machines

A virtual machine (VM) is a complete operating system installed on a host in order to create a hosted computing environment. Multiple VMs may exist on a single host and each VM may host applications. In this way a Windows computer may install a Linux VM as a guest, enabling Linux based software to be run within the VM, on the parent computer. Another use may be to allow multiple applications to run on a host whilst ensuring isolation between applications.

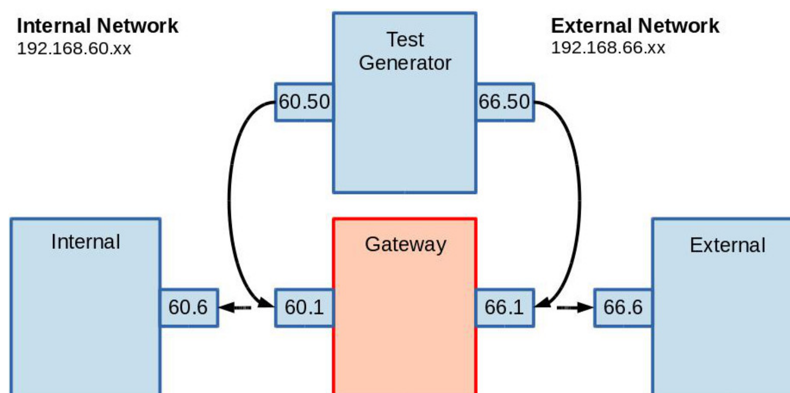


Figure 5.6: The test harness framework

For the test harness several VMs are hosted on a single host to create a test network. These VM's are all booted from an Ubuntu image (Ubuntu v14) and are networked together per Figure 5.6.

There are two networks shown, 192.168.60.0/24 for the internal network and 192.168.66.0/24 for the external network. The two hosts labelled *internal* and *external* may each represent larger networks on either side of the gateway that connects these two hosts.

The test generator has a NIC in both networks. It is a multi-homed host, meaning the host can use either NIC in order to masquerade as an internal or external host. Generated packets will have as a source address either an internal address or an external address. There is no direct link between the two NICs (unlike the gateway in Figure 5.6). In this way the test generator may, for example, create traffic from the internal address 192.168.60.50 for the external address 192.168.66.6 that can only be routed via the gateway VM.

The gateway VM passes packets from one gateway NIC directly to the other acting as the sole path between the internal and external hosts. This VM will later host the firewall SUT.

Networking Challenges

On reflection the assumption that networking the four VM's would be easy was not helpful. Initially three hosts were set up including one hosting the chosen test firewall, IPCOP[4] — setting up a firewall was perceived as the most unfamiliar task. After a day or so attempting to resolve networking issues, the firewall was replaced with an Ubuntu OS to route directly between the two domains, simplifying the task of debugging the network. But at the same time the test generator VM (which needed to be multi-homed) was added — unbeknown to the researcher this added significant network problems caused by the choice of the Linux operating system.

Reading blogs, text books and experiments to resolve the networking issues, consumed around five days. Early in this period experimenting broke the universities Internet through misuse of the DHCP (dynamic host configuration protocol) feature on VirtualBox; one of the variety of connection options available. The experiment failed and unaware of its effects (or purpose), the DHCP server was left over-night without closing either the VM or the host computer. The following morning one of the universities senior systems administrators visited personally. This resulted in some valuable one-on-one help with setting up the VMs which was leveraged later into help critiquing the routing being used between the VMs.

Eventually it was discovered that the Ubuntu operating system gets confused when there are two IP addresses assigned to the one host (the test generator) and the host is not intended to be a gateway.

The phrase 'multi-homed' turned out to be the final key and was discovered in a blog. The issues are described in RFC1122 Section 3.3.4.2 "Multihoming Requirements" [130] which details two models relevant to multihoming and the treatment of incoming packets; a *strong host model* and a *weak host model*. A strong host OS will associate IP addresses with the NIC, discarding incoming packets where the destination address does not match. A weak host OS will associate IP addresses with the host and will accept packets destined for those IP addresses on any NIC. RFC6419 Section 3.2.2.2 "Outbound and Inbound Addresses" [131] advises that Linux implements the *weak host model* and can be configured to support the *strong host model*.

Several issues had to be overcome to achieve the *strong host model* in order to achieve a multi-homed host in Linux. These included; reverse path routing, resolving source address selection, and ARP flux. ARP flux is detailed next as an example of a complex problem.

ARP flux

[4]IPCOP was chosen as it Linux based, appears to be well supported, is open source under the GNU Free Documentation License, Version 1.2 and it is fully featured. http://www.ipcop.org/

Address resolution protocol (ARP) is a layer two protocol, enabling communication between neighbouring hosts by utilising the hardware address (MAC address) on their Ethernet cards (NICs). Normal behaviour is to hold neighbour IP and MAC addresses and associate them with a specific port to send messages out on (eth0, eth1). If a message comes in with an unrecognised IP address, the ARP protocol broadcasts to its neighbours a "who has" request in order to update its tables. The recipients of this request will only reply if the listening NIC card is associated with that IP address. If no reply is received, the packet cannot be forwarded.

When applied to a multi-homed host with an OS that has adopted the *weak hosting model* (Linux), the ARP protocol searches all NICs for associated IP addresses and responds as soon as it finds a match. This resulted in the external NIC replying "I know the internal IP address", leading an external untrusted host able to make direct contact via that NIC, in the process circumventing the firewall.

The fix was to set two flags within the kernal (arp_ignore, arp_announce)[5] which then allows ARP replies only if the source IP of the incoming ARP packet is part of the the logical network configured on this interface. For example an interface configured with the IP address 192.168.66.50 and netmask 255.255.255.0 will respond to ARP requests with a source within the range 192.168.66.0 to 192.168.66.254.

5.3.2 Incorporating the Firewall

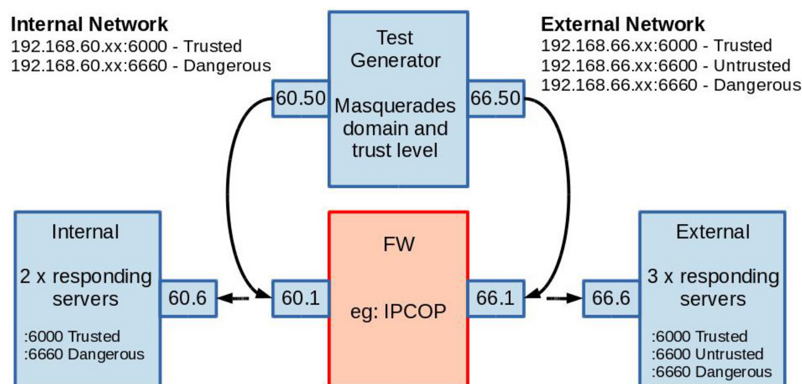


Figure 5.7: The test harness, including response servers and the firewall

In Figure 5.7 the gateway VM's Ubuntu OS is replaced by the firewall IPCOP which is configured to identify the internal and external networks, and its ACL is populated with details of trusted and dangerous hosts in both networks.

A firewall controls access using the five tuple defined as $\{src\ IP,\ src\ port,\ protocol,\ dst\ IP,\ dst\ port\}$. Each IP and port uniquely identifies a source or destination host and service (for example, 192.168.66.6:6660 uniquely identifies a host service). This feature is utilised within the test framework as shown in Figure 5.7. Each virtual machine has its own IP address, with the gateway and test generator VMs having two. The internal and external hosts each run multiple instances of the response server (as a service on the host) with each instance of the response server utilising a unique port. In this way a firewall can be told through configuring its ACL that 192.168.66.6:6000 should receive different treatment to 192.168.66.6:6660.

[5] https://wiki.openvz.org/Multiple_network_interfaces_and_ARP_flux

5.3.3 Performing the tests

It should be remembered that with MBT, the complexity and challenges are in creating the formal model and creating the test harness. With these in place creating and performing the tests is relatively trivial. The model test strategy (see section 3.5.5) creates the *tests.xml* file which is then passed to the test harness which takes care of the detail of running the tests through the SUT (see Appendix C).

Without the firewall 50% of the tests were failed, representing tests that require a firewall to, for example, stop flows to dangerous hosts and unsolicited flows from untrusted hosts.

When the firewall (IPCOP) is subsequently installed all the tests in Appendix C.1, pass.

5.4 Revisiting the firewall model

Upon completing the prototype test harness and demonstrating it performs as expected, attention returned to the formal model. The intention was to test two example attacks and the supposition that SDN firewall state is not always the same as end host state.

The two attacks modelled in the environment were repeated SYN and repeated FIN packets. The third test where an external host attempts to re-open a closed session, is also modelled in the environment.

Repeated SYN packets

Attackers may use repeated SYN packets for a variety of purposes, for example, scan a network [132], conduct a SYN Flood (a DoS attack) [133] or exfiltrate data from within a network by disguising it as a SYN packet. The event $ext_ATTACK_SYN_flood$ explores a SYN Flood attack by creating multiple SYN packets from the same address.

Repeated FIN packets

Attackers may use repeated FIN packets to attempt to covertly scan a firewall [132]. The event $ext_ATTACK_FIN_scan$ explores FIN scanning by creating repeated FIN packets. The attackers utilise normal behaviour for a closed port, which is to reply with a RST packet, while an open port merely drops the packet. Open ports are therefore identified by their lack of response.

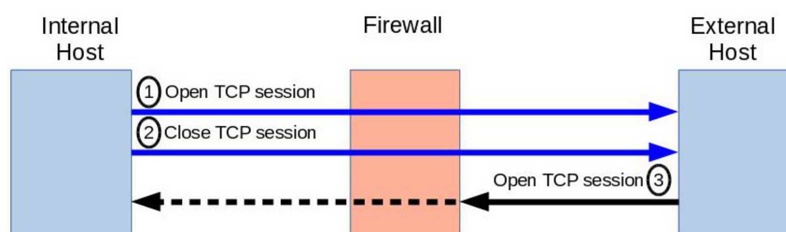


Figure 5.8: The external host should not be able to open a TCP session.

Reopening closed TCP sessions

Attackers here may utilise a lack of or slow convergence between firewall and end hosts. This is a problem that may be seen in SDN as an architectural problem where requiring network functions to rely on state held by third parties may force the adoption of strategies such as rule time-outs or requesting instructions from the third party holding the relevant state. These leave, for example, a firewall continuing to accept packets and therefore connection requests from untrusted external hosts, which may result in perpetually open TCP sessions. This is recreated in event $ext_ATTACK_reopen_closed_TCP_session$ which opens (1)

and closes (2) a TCP session, followed by the external untrusted host attempting to establish a new TCP session (3) while the firewall is still converging with the end host state (see Figure 5.8).

5.4.1 State explosion

With the initial Event-B modelling for the three new events completed (passed model checking which proves the math is sound — built the thing right), the model was passed to ProB to develop the test document (which will demonstrate the behaviours are as expected — built the right thing). It was anticipated this may highlight further issues to be resolved in the three new events.

However, ProB and the hardware used (Intel i7, 4GB RAM, Windows 7) failed to cope with the exploration of the increased state space. This is referred to as state explosion.

An analysis of the new events shows that the event *ext_ATTACK_reopen_closed_TCP_session* adds considerably to the depth of the state exploration to be undertaken by ProB, while the three events add to the breadth. It may be recalled that ProB performs a breadth first state exploration. Every level of depth added, exponentially increases the state required to be explored. In this case the model sets up and closes a TCP session, the longest path so far successfully explored, then attempts to send a *SYN* packet to re-open the session. A firewall should prevent this; with the model going through the states *ext_untrusted_packet_to_int*, *protocol_TCP_start*, *firewall*, *firewall_drop*, *packet_drop* and *terminate*. In total the depth required to search the new state space is increased by 5 (ignoring *terminate* which is present regardless).

This effectively halted model development with time becoming short. However there are potential solutions.

Potential state explosion solutions

The are several potential mitigation strategies available to this state explosion problem.

1. Improved hardware, for example, add more RAM or explore utilising grid computing
2. Use abstraction, for example, reduce the TCP handshakes to two events {open, closed}
3. Create an explicit test case for the scenario (see Section 3.5.5)

The first two strategies would take time that was not available to the researcher, exploring these is therefore left as potential future work. An explicit test strategy however could be created to test a scenario and is valid in circumstances where a models computational needs out-strips the equipment available. Appendix C.2 contains several additional tests the author considered important while developing the test harness plus one for the scenario illustrated in Figure 5.8.

5.4.2 Refactoring the test harness

The tests created by ProB utilising its breadth first strategy were known to be insufficient and adding the explicit test case illustrated in Figure 5.8, exposed a deficiency in the test harness.

Up to this point all tests are initiated by the test generator masquerading as an internal or external host, as needed. The new test case starts with the test generator masquerading as an internal host opening and closing a TCP session with an external untrusted host. This is followed by the external untrusted host attempting to start a new session. This requires

the response server to not just respond to a flow initiated by the test generator, but to also initiate a flow where indicated by the test.

Refactoring involved modifying the data payload to include the entire test, then enabling both the test generator and responder to parse it to determine who initiates the next packet. Where the next step of the test involves a host that is not the first host (the role taken by the test generator), the response server initiates a packet while the test generator listens for it.

Subsequently running the new tests on the test harness (without IPCOP), bought new unexpected problems related to the TCP TIME-WAIT state. The original problems described in Section 5.2.5 were resolved using a housekeeping algorithm, after the test finished, to close sessions deliberately not closed by test cases. In this case, because the test is still running, the housekeeping algorithm has not been called. Consequently the connection attempt by the external host is rejected by internal host which is still in the TCP TIME-WAIT state.

Further research, starting with Stevens *et al.* (2004) on Unix Network Programming [134] and online articles [6] [7] [8] [9] reinforced the earlier understanding that attempting to adjust or negate the TIME-WAIT period was not recommended. The authors describe two reasons for the TIME-WAIT state; to allow reliable connection and to allow old packets to expire in the network.

Adjusting or negating the TIME-WAIT mechanism may be avoided by using ranges of ports rather than a single port, for example, dangerous hosts may occupy port numbers 6660 to 6670 with every new test using the next port. Extending the test harness for this preferred option is left for future work, however, this would not satisfy a testcase which deliberately reuses an IP:port which has just been closed.

A second approach utilises a socket option called SO_LINGER which enables setting the TIME-OUT value to 0 and terminates connections using *RST* packets. This was earlier described as the equivalent of rudely hanging up the phone. However implementing it overcame the problems observed with the TIME-WAIT status.

Refactoring the test generator and response servers resulted in the work flow shown in Figure 5.9. Over the course of this test both servers alternate between active and listening sockets, and back again in preparation for the next test. The end result is a test that fails where there is no firewall installed (the last TCP open request succeeds) and passes when a stateful firewall (for example, IPCOP) is installed (state convergence between the firewall and end hosts is good and the last TCP open request is dropped).

It might be observed that to prove the firewall works correctly relies on the absence of data arriving from the response server (ie: the listening test generator times-out). It would be preferable to rely on the response server data, particularly if the tests become more complex, however any data transfer at present is as data on test packets which is of course dropped by the firewall. The issue of how to aggregate distributed data was discussed in Section 5.2.2, this test perhaps reinforces the need for a different approach which we leave for future work.

With the test harness performing as expected and the firewall IPCOP passing all tests it is timely to revisit the hypothesis.

[6] TCP option SO_LINGER (zero) - when it is required http://stackoverflow.com/questions/3757289/tcp-option-so-linger-zero-when-its-required. Retrieved May 2016.

[7] TIME_WAIT and its design implications for protocols and scalable client server systems. http://www.serverframework.com/asynchronousevents/2011/01/time-wait-and-its-design-implications-for-protocols-and-scalable-servers.html. Retrieved May 2016.

[8] The TIME-WAIT state in TCP and Its Effect on Busy Servers. http://www.isi.edu/touch/pubs/infocomm99/infocomm99-web/. Retrieved May 2016.

[9] UNIX Socket FAQ - Please explain the TIME_WAIT state. http://developerweb.net/viewtopic.php?id=2941. Retrieved May 2016.

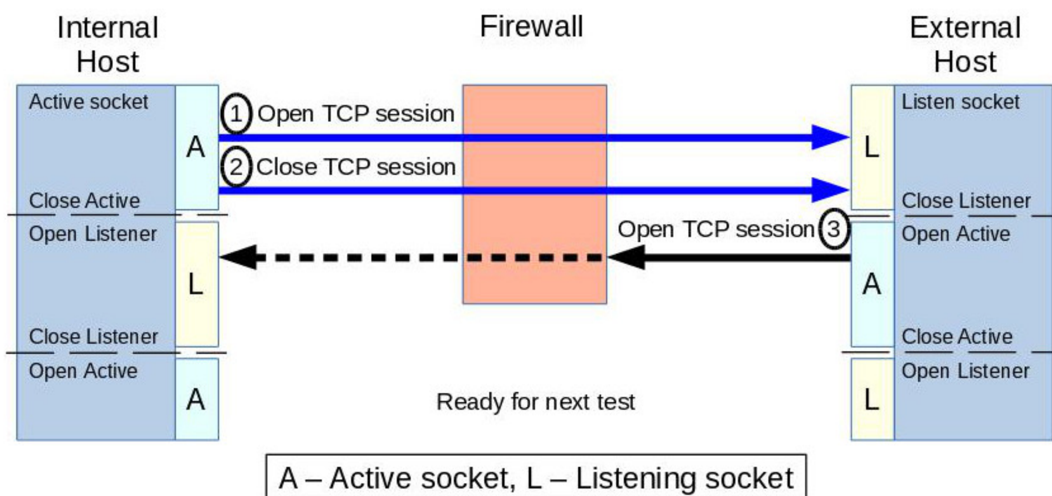


Figure 5.9: Test and response servers swap active and listening roles, mid test.

5.5 Revisiting the hypothesis

Recall that the hypothesis outlined in Chapter 4 expects to find state divergence between the SDN firewall and end hosts. Having modified the end hosts to void the TIME-WAIT behaviour may at first glance appear to void the test, particularly as TIME-WAIT will not typically be voided in the wild; meaning even if a SYN packet gets through the firewall because of its lack of state convergence, it may be interpreted by the receiving host as a *lost or wandering duplicate* and rejected. The net effect would be the attacker fails to re-establish the session.

One interpretation of this is that under certain conditions it proves that the combination of the two disparate algorithms means there is no security hole. In fact this may be the case where the internal host is guaranteed to initiate closing the TCP session. However it also proves that the network function has not converged state with the end hosts, relying on the end host to (perhaps) ensure a firewall failure does not compromise security.

On further reflection it may be recalled that the TIME-WAIT status applies only to the end host that initiated closing the TCP session. An attacker that instigates the close is not hindered by TIME-WAIT, as it is not a state the target host will enter (and its effects can be voided on the attacking host).

Wang *et al.* (2011) describes a battery draining attack using the firewall flaw we search for in this test [93]. The experiments were reported as being performed on two live carrier networks. Given our new understanding of the impact of the TIME-WAIT mechanism it would appear the experiment may have applied the strategy of always initiating TCP session closure using the attacking host.

5.6 Testing Multiple Firewalls

The intention, based on the hype generated by existing research (see Appendix A), was to test the SDN equivalent of a stateful firewall, compare its results with an in-line stateful firewall and perhaps demonstrate that it has flaws that can be explained by the use of SDN architecture.

During the course of this research it became apparent that despite the hype and in contrast to the dogma advocated by researchers, there are no SDN equivalents (yet?) to stateful in-line firewalls. Only two SDN firewalls could be found to test and on close inspection they

are merely RESTful interfaces to a controller which will then direct switches to filter packets. Both firewalls were tested in the test harness, in order to test the test harness prototype, and compare both implementations against the firewall model, each other and the Open Source Linux firewall IPCOP.

5.6.1 Adjusting the test harness for SDN applications

The existing test harness passes traffic through the firewall using the IP layer (layer 3). Initially attempts were made to utilise the capacity for Open vSwitch and SDN to also route using IP addresses, however this proved more difficult than expected. Instead the harness was adapted to utilise native switch functionality. The integrity of the tests is maintained because it tests connectivity between end hosts regardless of the nature of the network function between them. The adjustments were as follows:

1. Open vSwitch was hosted on a VirtualBox instance with ports to the internal host, external host and the controller. Only the controller was connected via layer 3.
2. The internal, external and test generator hosts had their network interfaces adapted with the netmasks adjusted to 255.255.0.0 and gateways dropped.
3. Settings within VirtualBox continued to provide the virtual wiring between VirtualBox instances, ensuring the only route between internal and external hosts is via the Open vSwitch host.
4. To allay concerns that this network may enable the test generator to circumvent the switch, the tests were run with each port on the test generator(internal, external) closed in turn and packet flows monitored using the tool tcpdump.

It is acknowledged that this approach is not ideal and future work will endeavour to adopt a more robust test harness solution.

5.6.2 The Ryu firewall

The Ryu firewall is provided with the Ryu controller and is detailed in the *Ryu Book*[10].

It may be described as creating a firewall object within the controller that responds to queries via the REST API (Representational State Transfer), a commonly accepted API for passing and retrieving data over the Internet. Based on the instructions provided (for example, an ACL comprised of flow rule parameters in JSON format) the firewall will convey flow rules to switches. Should a switch request instructions from the firewall object, it will reply with an appropriate rule. The default setting is to deny all traffic.

Timeouts are commonly used in firewall applications. Open vSwitch rule timeouts are set to a default 10 seconds [23], with a hard timeout setting also possible (but not used by the Ryu firewall). If a rule is not used for 10 seconds, the rule is dropped. Future flows are then referred to the controller firewall application and a new rule may be installed. The controller firewall application has no default rule timeout, meaning it will keep renewing the rule until the firewall application is advised not to by a third party (the Network Operator or another application). This is the same behaviour as in-line switches acting as packet filters (also discussed in Section 3.2.3). There is no dynamic behaviour.

For the test, the Ryu firewall was configured using its REST API to accept packets where both source and destination are *Trusted* hosts. This is the equivalent to actioning an ACL

[10]https://osrg.github.io/ryu-book/en/Ryubook.pdf Retrieved May 2015

white list. All other packets are dropped by default, which will include any packets from or to hosts on the ACL black list(*Dangerous* hosts).

In total 6 out of 19 tests were passed (the tests are in Appendix C.1 and C.2). An analysis of the tests that failed reveals the Ryu firewall performs as expected and drops all communication involving *Untrusted* and *Dangerous* hosts. This includes attempts by internal hosts to visit *Untrusted* external hosts (this might include benign hosts such as; www.catlovers.com, www.metro_buses.com, www.my_childs_creche.com).

The next test changed the default to allow communication between all hosts. The Ryu firewall was then configured to drop any packet where the source or destination host is on the ACL's black list (*Dangerous* hosts). This time the white list (*Trusted* hosts) are allowed by default alongside all unknown hosts (*Untrusted* hosts).

In total 16 of 19 tests were passed. An analysis of the tests reveals that the more permissive policy allows a wider range of communication through. However, the three tests that fail are informative; the first two are attempts by *Untrusted* external hosts to gain access using first TCP then UDP protocols — both attempts succeed[11]. The third tests if firewall access still exists for an *Untrusted* external host after the initial TCP connection is finished — the *Untrusted* host still has access.

We conclude that the Ryu Firewall user may adopt one of two strategies; restrict access only to white listed users which severely limits the utility of the network service for users; or block only black listed users which allows untrusted hosts to have unfettered network access. **In addition** the Ryu firewall's use of time-out as a rule removal mechanism is demonstrated to be susceptible to the attack described by Wang *et al.* (2011) where the attacker is in a position to re-open a recently closed TCP connection and maintain it in an open state for extended periods. **This demonstrates** an example of state divergence at the firewall. When both hosts have agreed the connection is closed — by passing and acknowledging *FIN* packets — the firewall should reflect this state and treat new connection attempts from the outside attacker appropriately by dropping them.

5.6.3 The Floodlight firewall

The Floodlight firewall has many similarities with the Ryu firewall. It is pre-installed with the Floodlight controller and is described on the Floodlight project page[12].

The Floodlight firewall also offers a REST API that accepts flow rule parameters in JSON format and installs rules into the controller firewall application. One of the acknowledged limitations of the Floodlight firewall (this will also be a limitation of the Ryu firewall) is that despite deleting a rule from the controller, the reliance on the switch timeout means an existing flow will persist for as long as it has continuous traffic through the switches[13] (see Section 2.7.10).

The results of the tests are identical to the Ryu firewall. This means that to the extent we have modelled and tested firewall behaviour we can claim these two firewalls are equivalent. This does not preclude the possibility that a more detailed model and more thorough automated test generation (ie: more than 19 tests) may find discrepancies, yet we can claim they both share at least one fundamental flaw, **they both introduce a third layer of state divergence.**

In contrast to both SDN firewalls, the in-line stateful firewall IPCOP passes all 19 tests.

[11]Criminals may easily register IP addresses for hacking purposes which because they do not appears on black lists, are then considered *Untrusted*.

[12]https://floodlight.atlassian.net/wiki/display/floodlightcontroller/ACL+%28Access+Control+List%29+REST+API

[13]https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Firewall

5.6.4 Testing other firewalls

Time prevented adapting the test harness to test proprietary firewall hardware, SDN firewalls using switch hardware (as opposed to VMs) or remote firewalls, however it is anticipated the test harness can be adapted to do so. This is left for future work.

Chapter 6

Results and Discussion

The author started and has finished this thesis with the belief that SDN offers significant benefits to networking. This view has been influenced by existing research and surveys of existing research.

However examining surveys of existing research also led to questioning the assertion repeated by many researchers (see Appendix A) that all network functionality should be moved out of the forwarding plane into the control plane. That this view is influential is reflected in the increasing numbers of research papers proposing or using such an architecture to provide SDN security [84–91]. Throughout the course of this research the author has sought to clarify the blanket assertion that network functions should be purged from the forwarding plane and has demonstrated that such a blanket approach is flawed.

6.1 Key areas of interest

Network functions are at the centre of this research and while the focus has been on firewalls, chapter 3 also examined the NAT and load balancer. This was in order to find commonalities between the three and derive a generic formal model of network functions that informed the later work. Utilising this model may also prove useful when proving properties of a network. We subsequently chose the firewall network function to illustrate in depth the problems perceived with the current SDN dogma, this is discussed further in section 6.1.1.

The unifying theme of this research can be loosely described as the problems that arise when using a remote decision maker to resolve local issues. There are parallels in other domains where local decision making is arbitrarily centralised, for example, some business models and the tensions between local and central governments. In contrast, business franchising centralises the determination of new locations, franchise rules and activities that benefit from a centralised view. For example, brand advertising. While the franchisee builds and uses local knowledge.

In practical terms the problems manifest as network functions not being in the same state as the end hosts expect (akin to local council not paying attention to local citizens) leading to state divergence. This third level of SDN state divergence is discussed in Section 6.1.2

MBT was chosen as the tool to explore this domain and potentially find a range of issues. It perhaps needs to be said that if only one issue was expected, there were faster ways of exposing it. However the hope was to discover a range of interesting problems and this was stymied only by the inability to take advantage of full path traversal through the model when generating tests. To generate only fourteen tests was a disappointment. None-the-less as a prototype the tool exceeded expectations and resolving remaining problems is a high priority. The future potential for MBT is discussed further in Section 6.1.3.

6.1.1 Testing the SDN dogma — firewalls

The author's paper *Global and Local Knowledge in SDN* [3], makes the observation that packet information found natively in the forwarding plane can be used immediately by network functionality in the forwarding plane. The SDN approach either uses stateless functionality, trading off capability to be stateless or stores the local knowledge in a remote third party (typically the controller) contributing to a variety of state divergence problems between controllers, controller and forwarding plane, and network function and end hosts. State divergence may cause loops, black holes and/or impact reachability.

Some network functions have stateless versions. Switches operating as packet filters may stop flows according to an ACL (alias stateless firewalls as implemented by Ryu and Floodlight). Load balancers may provide static mapping between requesting hosts and service hosts. NATs may rely on static mapping of internal addresses to NAT ports. The author has not yet seen a method allowing a stateless cache. In each case seen so far, the stateless functionality has involved a capability trade-off (see Section 3.2).

It became apparent during the course of this research that while researchers were promoting the SDN ideal, industry was being far more circumspect and research solutions appear to be falling short — it is possible the trade-offs may exceed the benefits. For example, when searching for SDN firewalls to test only two SDN stateless firewalls were found; the RYU and Floodlight firewalls. At present no SDN versions of a stateful firewall appear to exist. Informal discussions with two Australian Telecommunication providers using and selling SDN and NFV services, resulted in their separate, reluctant, admissions that for network security they rely still on directing traffic to in-line proprietary firewalls.

The concept of a firewall appears to be muddied. Researchers (not just the developers of the RYU and Floodlight firewalls) appear prone to pick the lowest possible specification that may be called a firewall (a switch packet filter) and produce an application that can read and interpret an ACL, generating appropriate flow rules (the RYU and Floodlight firewalls do not do this, they rely on the network operator or another application to compile the ACL into a JSON object containing the requisite OpenFlow parameters). Other researchers then appear to conclude this means all firewalls and firewall use cases can now be accomplished by using OpenFlow switches to offer packet filter functionality (see Appendix A).

The Faucet controller, which is based on RYU but is focused on layer 2 (data link), appears to also include the ability to direct switches to act as switch packet filters. This is similar to the RYU and Floodlight firewalls, albeit using a YAML structure rather than JSON to specify the requisite OpenFlow parameters. Time constraints prevented testing Faucet (paired with an OpenFlow switch and actioning an ACL) for behavioural equivalence, but this may form future work.

As this research demonstrates SDN firewall applications, using controllers to direct SDN switches to act as switch packet filters, are not equivalent to a stateful in-line firewall. Regrettably we cannot test an SDN stateful firewall as none yet exist, however no doubt one will appear soon.

6.1.2 SDN's third layer of state divergence

The third layer of state divergence is network function state and end host state where network functions may deduce end host state from passing packets. Of concern are scenarios where the two end hosts believe, for example, that a communication has finished, while the network function believes otherwise (see Sections 2.7.10, 4.1). Perhaps because it is not capable of observing and deducing state (for example, a stateless network function). This may lead to security concerns and the tests conducted through MBT confirm the vulnerability (see Sections 5.5 and 5.6).

The first two layers of state divergence have been discussed (state divergence in the logically centralised control plane, Sections 2.3.1, 2.7.6 and state divergence in the control to forwarding plane, Sections 2.3.2, 2.3.3, 2.7.9. The concern raised in this research is that these three types of state convergence may compound problems in SDN leading to black holes, loops and reachability problems (the tests conducted fall under reachability problems). How severe these problems may be has not been established and is left to future work, however flow volumes in networking at scale mean such issues cannot be trivialised.

Another question relates to whether this third layer of state divergence is applicable to all network functions, a subset or perhaps only to firewalls. This is another area where more research is required.

6.1.3 MBT in Networking

At the heart of this research is MBT, a technology that appears to be poorly understood and utilised in the software industry. It is anticipated this research shows a potential new use for the technology in enabling behavioural comparisons to be made between implementations within the network industry.

Of interest to network engineers is the potential for formal models based on widely accepted standards, generating tests that can then be applied to network function implementations. If models and test harness exist, this may be close to push button technology.

Of interest to start-ups in the networking, NFV and SDN space is the potential for their network function implementations to be directly compared with those from well established names in the industry.

MBT is not a stand-alone solution for comparison of implementations. Formal modelling cannot provide performance tests, nor hardware reliability tests, merely behavioural tests (including potentially finding bugs). However it is black-box behavioural tests that are missing from the network engineers toolbox.

6.2 Contributions

This thesis makes multiple contributions to SDN research, including using formal methods, modelling several network functions (one in detail) and using MBT as a framework to explore the implementation of network functionality in both legacy networks and SDN. The case for SDN to purge network functions from the forwarding plane is examined and found to be a widely held yet unproven dogma.

Several examples of common middleboxes are examined and contrasted with their SDN implementations to find the architecture used in SDN appears to limit the ability of OpenFlow switches to perform as network functions. There is a capability trade-off that is not explored in the SDN literature, a pre-requisite for discussing how novel SDN management applications may overcome the resulting problems.

To illustrate the difference a stateful firewall is formally modelled and utilising model-based testing, an in-line implementation of a firewall is compared with two SDN implementations. It is found that the SDN firewalls have traded away key firewall behaviours in order to be controller applications, including that of state convergence with end hosts — meaning these firewalls may be kept perpetually open by an attacker.

Specifically the main contributions of this thesis are:

1. An analysis of SDN literature to find ten problems that may be exacerbated by moving network function algorithm and state from the forwarding plane to the controller (see Section 2.7).

2. A formal model of a generic network function (see Section 3.3), based on the analysis of load balancer, NAT and firewall functionality (see Section 3.2), a highly detailed formal model of a firewall (see Section 5.1) and a MBT test harness (see Section 5.2).
3. The first application of MBT to network functions. The creation of tests from a formal model of a network function and applying them to a test harness that tests in turn, three implementations of that network function (see Section 5.6).
4. Possibly the first use of MBT as a tool for determining behavioural equivalence between black-box implementations, to the extent that behaviours have been modelled (see Section 2.6). Useful in this domain because of the volume of vendors offering network functionality with similar behavioural properties.
5. The creation of abstractions of middlebox functionality (see Section 3.3) which may assist in proving fundamental network properties such as no loops, no black holes and reachability in the presence of dynamic network functionality. These are the subject of ongoing research.
6. A third layer of state divergence in SDN. The first two layers are discussed (state divergence in the logically centralised control plane, Sections 2.3.1, 2.7.6 and state divergence in the control to forwarding plane, Sections 2.3.2, 2.3.3, 2.7.9) and the third (state divergence between network function and end hosts) is described (Sections 2.7.10, 4.1, 6.1.2) and demonstrated (Sections 5.4, 5.5).

6.3 Future Work

Research directions
There is a wealth of interesting directions to take this research:

- More research on the limitations SDN architecture imposes on how network functions are implemented and exploring the forwarding plane verses control plane trade-offs.
- Are concerns expressed on the third layer of SDN state divergence widely applicable to all network function types or a subset or perhaps only to firewalls.
- The three layers of SDN state divergence appear to compound to create problems that increase black holes, loops and reachability problems in SDN networks. This supposition should be tested.
- Can novel SDN management applications mitigate the trade-offs in adopting stateless network functions?
- Using SDN to manage in-line network functions may be an interesting avenue.
- Will the generic properties of network functions detailed in Chapter 3, prove to be valid over a wider range of implementations?
- Repeat the surveys conducted by Sherry *et al.* (2012) and Sekar *et al.* (2012) into middlebox ubiquity in current networks, taking into account todays potentially greater availability of NFV and cloud services.
- Using better network function abstractions to prove network properties, such as no loops, no black holes and reachability.

- Find (or develop) and test a stateful SDN firewall.

Improving the prototypes

There were also several aspects of the modelling and prototyping that will benefit from more work:

- The ProB tool did not provide the range of test strategies expected. Extending ProB or updating the MBT add-on to Rodin to run with current versions may be useful. Alternatively there may be benefit in exploring other tools as alternatives to Rodin/ProB.
- Mitigate state explosion by examining new hardware and other methodologies for exploring a model's paths without suffering from resource constraints (see Section 5.4.1). For example, perhaps Grid computing or utilising a cloud service.
- Develop the model environment further, to better reflect the range of bad behaviours observed in the Internet (for example, from attackers) and include more protocols. Another area for improvement is reducing the states traversed by merging existing states into one for the TCP opening and closing handshakes.
- Modelling the firewalls hardware may allow resource depletion attacks to be modelled, for example RAM and CPU.
- Model network functions such as NAT, load balancer and cache and compare various implementations.
- The test harness might be expanded to test proprietary network functions, hardware implementations of OpenFlow switches and perhaps remote network functions over the Internet.

Appendices

Appendix A

15 Surveys of SDN research

At least 15 survey papers on SDN research have been published over 2014/15; surveying a mean of 167 papers. [135–149], with one surveying 581 papers [144]. Reviewing these surveys is insightful, in many respects they provide an excellent summary of SDN research. However it is perhaps unsurprising for a relatively new field that this review may lead to critically examining underlying assumptions.

Middleboxes provide the non-routing functions within a network. Examples include in-line firewalls, caches, network address translators (NATs) and load balancers. These use local knowledge in decision making in order to improve properties of the network. For example, caching may be provided by a middlebox. Its function is to store commonly requested web content in order to decrease latency for the consumer plus conserve the computational and bandwidth resources of the provider. This cannot be achieved with an OpenFlow switch which does not have the computational ability to utilise a cache algorithm nor the RAM to hold the dynamic state.

15 of the 16 surveys, state or allude within the first 3 pages that the SDN forwarding plane is intended to be composed only of switches and routers — no middleboxes. The remaining survey incorrectly claims the SDN control plane has a well defined interface over middleboxes [139]. It appears the community strongly correlates the capabilities of the OpenFlow interface, with the capabilities of SDN. Five of the papers directly imply that OpenFlow switches are sufficient to provide (unqualified) firewall functionality [138,139,142,144,149].

For example, Nunes *et al.* (2014) (an otherwise excellent survey) largely ignores the literature on middleboxes and appears to consider that they must by default be removed from networks, hinting the function can be performed by controllers [146]. Many surveys rely on articles such as Limoncelli (2012), whom asserts that middlebox functionality can merely be achieved by switches, without providing research evidence [150].

> "OpenFlow ... because it can drop packets, it can act as a firewall."
> Limoncelli (2012)

These assumptions appear to arise from the enthusiastic support of and reliance on OpenFlow which offers a switch centric view of SDN. There is no OpenFlow mechanism for network functionality other than switches. Consequently OpenFlow researchers appear to easily adopt the idea that all middleboxes should be purged from the forwarding plane and implemented as control plane applications.

The arguments that middleboxes should be purged include:

1. they increase network complexity [144] p.17

2. choke points create performance issues [145] p.500

3. integrating into the controller is preferred [146] p.1626

4. OpenFlow switches can rewrite packets. [150] p.46

These arguments to purge middleboxes are at odds with their ubiquity in current networks. Only three survey papers reference Sherry *et al.* (2012) who shows that nearly half of all network nodes are middleboxes [11]. The remainder appear to assume middlebox numbers in networks are low. Sherry's finding is supported by a similar survey by Sekar *et al.* (2012) [10].

Key phrase	Median occurrence of phrase in fifteen survey papers
middlebox	2
firewall	3
NFV	2
service chain	0

Table A.1: Review of fifteen papers surveying SDN research

This apparent ignorance of the volume of middleboxes in existing networks plus the enthusiastic adoption of OpenFlow's switch centric view has perhaps created this next point — that there is little interest in middlebox, NFV or service chaining research in SDN research. Table A.1 shows a measure of SDN's interest. It seems remarkable that in 15 survey papers with over 2500 references, at least 581 of which are unique, the phrase *middlebox* (the phrase covering near half the hardware in existing networks) is mentioned a median of 2 times.

SDN researchers occasionally tout examples of stateless middleboxes and promote them as the SDN alternative, typically as 'proof' that middleboxes can and must be purged. However they rarely discuss that this stateless property may come at the cost of other properties. Chapter 3 discusses the difference between stateful and stateless firewalls, how stateful NATs minimise flow table entries with benefits to switch speeds and how stateless load balancers use static, coarse distribution of flows which slows down the incorporation of new new service hosts.

In summary, these 15 survey papers demonstrate there is a lack of SDN research to support the SDN dogma that all network functions should be purged from the forwarding plane and placed in the control plane. These concerns appear to be shared by others, recently the Open Network Foundation (2015) published a paper TR-518 "Relationship of SDN and NFV" to address some of these issues and assist the two fields to capitalise on each others strengths instead of, we quote, "reinventing the wheel" [4].

Appendix B

Firewall in Event-B

The Event-B model of a firewall and its networking environment.

Note that for reasons related to conserving space, the machine presented here is the final machine (a snapshot) which includes all previous machines and thier various refinements. Consequently this machine includes both the environment and the model of the SUT plus all earlier refinements. Where events are extended by later events (for example, *protocol_TCP* is extended by *protocol_TCP_start*, *protocol_TCP_continue* and *protocol_TCP_finish*) the superceded events are not carried forward by Rodin into later machines.

B.1 Firewall Context

Defines fixed global values and sets.

CONTEXT c_fw55_stateful_firewall

SETS PROTOCOL_TYPE T ACTION

CONSTANTS int ext NODES egress ingress chokepoint UDP TCP ID PACKET_ID PROTOCOL FW_PROTOCOL PKT_PROTOCOL HEADER PKT_TCP_SYN PKT_TCP_ACK PKT_TCP_FIN NFB TCP_SESSION_REGISTER SRC_IP DST_IP trusted untrusted dangerous test_UDP test_TCP FW_TIME pass drop FW_TIMEOUT interval FW_RULE_ID FW_SRC_IP FW_DST_IP FW_PROTOCOL

AXIOMS

@$axm0_1$ $NODES = 0 \cdot \cdot 1000$ // nodes are a finite set

@$axm0_2$ $int = 1 \cdot \cdot 5$ // internal nodes are known/listed

@$axm0_3$ $ext = (NODES \setminus (int \cup \{0\}))$

theorem @$axm0_4$ $int \cap ext = \varnothing$

theorem @$axm0_5$ $NODES = int \cup ext \cup \{0\}$

@$axm1_1$ $egress = \{5\}$ // all nodes forming the chokepoints between int, ext are known/listed

@$axm1_2$ $ingress = \{999\}$

@$axm1_3$ $egress \subset int \wedge ingress \subset ext$

@$axm1_4$ $chokepoint = egress \rightarrowtail ingress$ // the paths between the two domains, total injective used (one to one)

@$axm2_1$ $ID = \mathbb{N}1$

@$axm2_2$ $PACKET_ID = \mathbb{N}$ // 0 is the null packet

@$axm2_3$ $partition(PROTOCOL_TYPE, \{UDP\}, \{TCP\})$

@$axm2_4$ $PROTOCOL = PACKET_ID\ x\ PROTOCOL_TYPE$

@$axm3_1$ $PKT_TCP_SYN = PACKET_ID\ x\ \{0,1\}$

@$axm3_2$ $PKT_TCP_ACK = PACKET_ID\ x\ \{0,1\}$

@$axm3_3$ $PKT_TCP_FIN = PACKET_ID\ x\ \{0,1\}$

$@axm4_1\ NFB = T\ x\ \{0,1\}$
$@axm5_1\ TCP_SESSION_REGISTER = NODES\ x\ NODES$
$@axm5_2\ SRC_IP = PACKET_ID\ x\ NODES$
$@axm5_3\ DST_IP = PACKET_ID\ x\ NODES$
$@axm6_4\ trusted = int \cup ingress \cup \{500,\ 501,\ 502,\ 503\}$ // Trusted external nodes are listed, always allow packets to pass
$@axm6_5\ dangerous = \{666,\ 700,\ 701,\ 800,\ 801\}$ // Dangerous external nodes are listed, always drop packets
$@axm6_7\ egress \subset trusted \land ingress \subset trusted$
$@axm6_8\ untrusted = NODES \setminus (trusted \cup dangerous)$
$theorem\ @axm6_9\ NODES = trusted \cup dangerous \cup untrusted$
$@axm50_1\ test_UDP = TRUE$
$@axm50_2\ test_TCP = TRUE$
$@axm55_1\ FW_TIME = \mathbb{N}$
$@axm55_2\ ACTION = \{drop, pass\}$ // second drop to create an option for src = untrusted, dst = trusted
$@axm55_3\ pass \neq drop$
END

B.2 Firewall Model

MACHINE fw55_stateful_firewall
REFINES fw50_firewall___6_evts
SEES c_fw05_stateful_firewall
VARIABLES src dst last_transit is_reply enter_chokepoint exit_chokepoint packet_id pkt_protocol set_pkt_protocol pkt_tcp_SYN pkt_tcp_ACK pkt_tcp_FIN TCP_sessions src_ip dst_ip spoof_IP_evt target_state terminate session_closed fw_machine fw_tick fw_finished fw_action fw_add_rule fw_rules fw_FIN fw_src_ip fw_dst_ip fw_protocol fw_timeout fw_remove_rule fw_drop fw_forward fw_exit
INVARIANTS
$inv_fw00_1 : fw_action \in ACTION$
$inv_fw00_6 : fw_add_rule \in BOOL$
$inv_fw05_1 : fw_rules \subseteq FW_RULE_ID$
$inv_fw05_2 : fw_src_ip \in \mathbb{P}(FW_SRC_IP)$
$inv_fw05_3 : fw_dst_ip \in \mathbb{P}(FW_DST_IP)$
$inv_fw05_4 : fw_protocol \subseteq FW_PROTOCOL$
$inv_fw05_5 : fw_timeout \subseteq FW_TIMEOUT$
$inv_fw05_6 : fw_FIN \in BOOL$
$inv_fw00_7 : fw_remove_rule \in BOOL$
$inv_fw00_10 : fw_drop \in BOOL$
$inv_fw00_11 : fw_forward \in BOOL$
$inv_fw00_12 : fw_exit \in BOOL$
EVENTS
INITIALISATION
BEGIN
$act00_1 : src := 0$
$act00_2 : dst := 0$
$act00_3 : last_transit := FALSE$
$act00_4 : is_reply := FALSE$

$act05_1 : enter_chokepoint := FALSE$
$act05_2 : exit_chokepoint := FALSE$
$act10_1 : pkt_protocol := \varnothing$
$act10_2 : packet_id := 0$
$act10_3 : set_pkt_protocol := FALSE$
$act10_4 : pkt_tcp_SYN := \varnothing$
$act10_5 : pkt_tcp_ACK := \varnothing$
$act10_6 : pkt_tcp_FIN := \varnothing$
$act15_1 : TCP_sessions := \varnothing$
$act20_1 : src_ip := \varnothing$ // unique to host, uses node ID
$act20_2 : dst_ip := \varnothing$
$act30_1 : spoof_IP_evt := FALSE$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
$act50_3 : session_closed := FALSE$
$act_fw00_1 : fw_machine := FALSE$
$act_fw00_3 : fw_tick := 0$
$act_fw00_4 : fw_finished := FALSE$
$act_fw00_2 : fw_action := pass$
$act_fw00_5 : fw_add_rule := FALSE$
$act_fw05_1 : fw_rules := \varnothing$
$act_fw05_4 : fw_src_ip := \varnothing$
$act_fw05_5 : fw_dst_ip := \varnothing$
$act_fw05_6 : fw_protocol := \varnothing$
$act_fw05_7 : fw_timeout := \varnothing$
$act_fw52_8 : fw_FIN := FALSE$
$act_fw00_6 : fw_remove_rule := FALSE$
$act_fw00_10 : fw_drop := FALSE$
$act_fw00_11 : fw_forward := FALSE$
$act_fw00_12 : fw_exit := FALSE$

END

packet_send_int_to_trusted

ANY x y

WHERE

$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y =\neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in trusted \cap int$
$grd25_2 : y \in trusted \cap ext$

THEN

$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :\mid packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$

$act50_2 : terminate := FALSE$
END
packet_send_int_to_untrusted
ANY x y
WHERE
$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y = \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in trusted \cap int$
$grd25_2 : y \in untrusted \cap ext$
THEN
$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
END
packet_send_int_to_dangerous
ANY x y
WHERE
$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in trusted \cap int$
$grd25_2 : y \in dangerous \cap ext$
THEN
$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
END
packet_send_ext_trusted_to_int
ANY x y
WHERE
$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y \neq 0$ // making this explicit

$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in trusted \cap ext$
$grd25_2 : y \in trusted \cap int$

THEN

$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

packet_send_ext_untrusted_to_int

ANY x y

WHERE

$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int U ext excludes 0
$grd00_3 : x \neq 0 \wedge y \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in untrusted \cap ext$
$grd25_2 : y \in trusted \cap int$

THEN

$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

packet_send_ext_dangerous_to_int

ANY x y

WHERE

$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in dangerous \cap ext$
$grd25_2 : y \in trusted \cap int$

THEN

$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$

$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

spoof_ext_untrusted_IP

ANY x y

WHERE

$grd00_1 : src = 0 \wedge dst = 0$
$grd00_2 : x \in (int \cup ext) \wedge y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \wedge y \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in untrusted \cap ext$
$grd25_2 : y \in trusted \cap int$
$grd30_1 : x = 900$ // sets src address

THEN

$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :| packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
$act30_1 : spoof_IP_evt := TRUE$

END

ext_ATTACK_SYN_flood

ANY _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd10_1 : set_pkt_protocol = TRUE$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd30_1 : spoof_IP_evt = TRUE$
$grd30_3 : _pkt_tcp_ACK = \{(packet_id) \mapsto 0\}$
$grd30_4 : _pkt_tcp_SYN = \{(packet_id) \mapsto 1\}$
$grd30_5 : _pkt_tcp_FIN = \{(packet_id) \mapsto 0\}$

THEN

$act10_1 : pkt_protocol := \{(packet_id) \mapsto TCP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$act30_1 : spoof_IP_evt := FALSE$

END

ext_ATTACK_FIN_scan

ANY _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd10_1 : set_pkt_protocol = TRUE$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd30_1 : spoof_IP_evt = TRUE$
$grd30_3 : _pkt_tcp_ACK = \{(packet_id) \mapsto 0\}$
$grd30_4 : _pkt_tcp_SYN = \{(packet_id) \mapsto 0\}$
$grd30_5 : _pkt_tcp_FIN = \{(packet_id) \mapsto 1\}$
THEN
$act10_1 : pkt_protocol := \{(packet_id) \mapsto TCP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$act30_1 : spoof_IP_evt := FALSE$
END
ext_ATTACK_reopen_closed_TCP_session
ANY x y
WHERE
$grd00_1 : src = 0 \land dst = 0$
$grd00_2 : x \in (int \cup ext) \land y \in (int \cup ext)$ // int ∪ ext excludes 0
$grd00_3 : x \neq 0 \land y \neq 0$ // making this explicit
$grd00_4 : x \in int \Rightarrow y \in ext$ // x & y are in different domains
$grd00_5 : y \in int \Rightarrow x \in ext$
$grd25_1 : x \in untrusted \cap ext$
$grd25_2 : y \in trusted \cap int$
$grd50_1 : session_closed = TRUE$
THEN
$act00_1 : src, dst := x, y$
$act00_2 : last_transit := FALSE$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :\mid packet_id' = packet_id + 1$
$act10_2 : set_pkt_protocol := TRUE$
$act20_1 : src_ip := \{(packet_id + 1) \mapsto x\}$
$act20_2 : dst_ip := \{(packet_id + 1) \mapsto y\}$
END
protocol_UDP
WHEN
$grd10_1 : set_pkt_protocol = TRUE$
$grd30_1 : spoof_IP_evt = FALSE$
$grd50_1 : test_UDP = TRUE$ // switch to turn on/off UDP messaging
THEN
$act00_1 : last_transit := TRUE$
$act10_1 : pkt_protocol := \{(packet_id) \mapsto UDP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
END
protocol_TCP_start
ANY _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd10_1 : set_pkt_protocol = TRUE$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd15_1 : src \mapsto dst \notin TCP_sessions$
$grd15_2 : _pkt_tcp_SYN = \{packet_id \mapsto 1\}$
$grd15_3 : _pkt_tcp_ACK = \{packet_id \mapsto 0\}$
$grd15_4 : _pkt_tcp_FIN = \{packet_id \mapsto 0\}$
$grd30_1 : spoof_IP_evt = FALSE$
$grd50_1 : test_TCP = TRUE$ // switch to turn on/off TCP messaging

THEN

$act10_1 : pkt_protocol := \{(packet_id) \mapsto TCP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$inv15_1 : TCP_sessions := TCP_sessions \cup \{src \mapsto dst, dst \mapsto src\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

protocol_TCP_continue

ANY _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd10_1 : set_pkt_protocol = TRUE$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd15_1 : src \mapsto dst \in TCP_sessions$
$grd15_2 : _pkt_tcp_SYN = \{packet_id \mapsto 0\}$
$grd15_3 : _pkt_tcp_ACK = \{packet_id \mapsto 0\}$
$grd15_4 : _pkt_tcp_FIN = \{packet_id \mapsto 0\}$
$grd50_1 : test_TCP = TRUE$ // switch to turn on/off TCP messaging

THEN

$act10_1 : pkt_protocol := \{(packet_id) \mapsto TCP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

protocol_TCP_finish

ANY _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd10_1 : set_pkt_protocol = TRUE$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd15_1 : src \mapsto dst \in TCP_sessions$

$grd15_2 : _pkt_tcp_SYN = \{packet_id \mapsto 0\}$
$grd15_3 : _pkt_tcp_ACK = \{packet_id \mapsto 0\}$
$grd15_4 : _pkt_tcp_FIN = \{packet_id \mapsto 1\}$
$grd50_1 : test_TCP = TRUE$ // switch to turn on/off TCP messaging
THEN
$act10_1 : pkt_protocol := \{(packet_id) \mapsto TCP\}$
$act10_2 : set_pkt_protocol := FALSE$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$inv15_1 : TCP_sessions := TCP_sessions \setminus \{src \mapsto dst, dst \mapsto src\}$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
$act50_3 : session_closed := TRUE$
END
firewall
ANY _a
WHERE
$grd05_1 : enter_chokepoint = TRUE$
$grd05_2 : src \neq 0 \wedge dst \neq 0$
$grd05_3 : _a = src \mapsto dst$
$grd10_1 : set_pkt_protocol = FALSE$
THEN
$act05_1 : enter_chokepoint := FALSE$
$act05_2 : exit_chokepoint := TRUE$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$
$act_fw20_1 : fw_machine := TRUE$
$act_fw00_3 : fw_tick := fw_tick + 1$
$act_fw52_1 : fw_add_rule := FALSE$
$act_fw52_2 : fw_remove_rule := FALSE$
$act_fw52_3 : fw_exit := FALSE$
$act_fw52_4 : fw_forward := bool((src \in trusted \wedge \neg dst \in dangerous) \vee (src \in ext \wedge src \notin trusted \wedge (\exists ident, protocol \cdot ident \in fw_rules \wedge protocol \in PROTOCOL_TYPE \wedge ident \mapsto src \in fw_src_ip \wedge ident \mapsto dst \in fw_dst_ip \wedge ident \mapsto protocol \in fw_protocol \wedge packet_id \mapsto protocol \in pkt_protocol)))$
$act_fw52_5 : fw_drop := bool(src \in dangerous \vee dst \in dangerous \vee (src \in ext \wedge src \notin trusted \wedge \neg(\exists ident, protocol \cdot ident \in fw_rules \wedge protocol \in PROTOCOL_TYPE \wedge ident \mapsto src \in fw_src_ip \wedge ident \mapsto dst \in fw_dst_ip \wedge ident \mapsto protocol \in fw_protocol \wedge packet_id \mapsto protocol \in pkt_protocol)))$
END
firewall_forward
ANY _rule_timed_out _first_FIN_pkt _second_FIN_pkt _fw_add_rule _fw_remove_rule
WHERE
$grd_fw20_1 : fw_machine = TRUE$
$grd_fw50_4 : fw_forward = TRUE$
$grd_fw52_0 : fw_exit = FALSE$
$grd_fw52_1 : _rule_timed_out = \{r \mid r \in fw_rules \wedge (\exists t \cdot t \in fw_timeout \wedge t = r \mapsto fw_tick + 1)\}$

$grd_fw52_2 : _first_FIN_pkt = bool(packet_id \mapsto TCP \in pkt_protocol \wedge packet_id \mapsto 1 \in pkt_tcp_FIN)$

$grd_fw52_3 : _second_FIN_pkt = \{r \mid r \in fw_rules \wedge (fw_FIN = TRUE \wedge r \mapsto TCP \in fw_protocol \wedge packet_id \mapsto TCP \in pkt_protocol \wedge packet_id \mapsto 0 \in pkt_tcp_FIN \wedge packet_id \mapsto 1 \in pkt_tcp_ACK \wedge ((r \mapsto src \in fw_src_ip \wedge r \mapsto dst \in fw_dst_ip) \vee (r \mapsto dst \in fw_src_ip \wedge r \mapsto src \in fw_dst_ip)))\}$ // removes on the final ACK...

$grd_fw52_4 : _fw_add_rule = bool((\exists n \cdot n \in int \wedge src_ip = \{packet_id \mapsto n\}) \wedge (\exists n \cdot n \in untrusted \wedge dst_ip = \{packet_id \mapsto n\}) \wedge \neg(packet_id \mapsto TCP \in pkt_protocol \wedge packet_id \mapsto 1 \in pkt_tcp_ACK))$

$grd_fw50_2 : fw_add_rule = FALSE$

$grd_fw50_3 : fw_remove_rule = FALSE$

$grd_fw52_5 : _fw_remove_rule = bool((_rule_timed_out \neq \varnothing) \vee (_second_FIN_pkt \neq \varnothing))$

THEN

$act_fw20_1 : fw_action := pass$

$act_fw52_2 : fw_exit := bool((_fw_add_rule = FALSE) \wedge (_fw_remove_rule = FALSE))$

$act_fw50_3 : fw_add_rule := _fw_add_rule$

$act_fw52_4 : fw_FIN := _first_FIN_pkt$ // only deals with sequential FINs, not mixed traffic

$act_fw52_5 : fw_remove_rule := _fw_remove_rule$

END

firewall_add_rule

ANY add_rule_id

WHERE

$grd_fw20_1 : fw_machine = TRUE$

$grd_fw50_1 : fw_add_rule = TRUE$

$grd_fw52_2 : fw_exit = FALSE$

$grd_fw05_3 : add_rule_id = \{r \mid r \in FW_RULE_ID \wedge ((card(fw_rules) = 0 \wedge r = 1) \vee (card(fw_rules) > 0 \wedge max(fw_rules) < max(FW_RULE_ID) \wedge r = max(fw_rules) + 1) \vee (card(fw_rules) > 0 \wedge max(fw_rules) = max(FW_RULE_ID) \wedge r = max(fw_rules)))\}$

THEN

$act_fw05_5 : fw_protocol := fw_protocol \cup (add_rule_id\, x\, \{r \mid r \in PROTOCOL_TYPE \wedge (\exists t \cdot t \in pkt_protocol \wedge t = packet_id \mapsto r)\})$

$act_fw52_2 : fw_exit := TRUE$

$act_fw05_1 : fw_rules := fw_rules \cup add_rule_id$

$act_fw05_2 : fw_timeout := fw_timeout \cup (add_rule_id\, x\, \{fw_tick + interval\})$

$act_fw05_3 : fw_src_ip := fw_src_ip \cup (add_rule_id\, x\, \{r \mid r \in NODES \wedge (\exists t \cdot t \in dst_ip \wedge t = packet_id \mapsto r)\})$

$act_fw05_4 : fw_dst_ip := fw_dst_ip \cup (add_rule_id\, x\, \{r \mid r \in NODES \wedge (\exists t \cdot t \in src_ip \wedge t = packet_id \mapsto r)\})$

END

firewall_remove_rule

ANY _remove_rule_ids

WHERE

$grd_fw20_1 : fw_machine = TRUE$

$grd_fw50_1 : fw_remove_rule = TRUE$

$grd_fw52_2 : fw_exit = FALSE$

$grd_fw52_3 : _remove_rule_ids = \{r \mid r \in fw_rules \wedge ((r \mapsto src \in fw_src_ip \wedge r \mapsto dst \in fw_dst_ip) \vee (r \mapsto dst \in fw_src_ip \wedge r \mapsto src \in fw_dst_ip))\}$

THEN

$act_fw05_5 : fw_protocol := \{x \mapsto y \mid x \mapsto y \in fw_protocol \land x \notin _remove_rule_ids\}$

$act_fw52_2 : fw_exit := TRUE$

$act_fw05_1 : fw_rules := (fw_rules \setminus (_remove_rule_ids))$

$act_fw05_2 : fw_timeout := \{x \mapsto y \mid x \mapsto y \in fw_timeout \land x \notin _remove_rule_ids\}$

$act_fw05_3 : fw_src_ip := \{x \mapsto y \mid x \mapsto y \in fw_src_ip \land x \notin _remove_rule_ids\}$

$act_fw05_4 : fw_dst_ip := \{x \mapsto y \mid x \mapsto y \in fw_dst_ip \land x \notin _remove_rule_ids\}$

END

firewall_drop

ANY _a

WHERE

$grd00_1 : src \neq 0 \land dst \neq 0$

$grd00_2 : _a = src \mapsto dst$

$grd_fw20_1 : fw_machine = TRUE$

$grd_fw50_2 : fw_add_rule = FALSE$

$grd_fw50_3 : fw_remove_rule = FALSE$

$grd_fw52_3 : fw_drop = TRUE$

THEN

$act00_1 : src, dst := 0, 0$ // null packet

$act00_2 : is_reply := FALSE$

$act00_3 : last_transit := FALSE$

$act05_1 : enter_chokepoint := FALSE$

$act05_2 : exit_chokepoint := FALSE$

$act10_1 : pkt_protocol := (\{packet_id\}pkt_protocol)$

$act10_2 : pkt_tcp_SYN := (\{packet_id\}pkt_tcp_SYN)$

$act10_3 : pkt_tcp_ACK := (\{packet_id\}pkt_tcp_ACK)$

$act10_4 : pkt_tcp_FIN := (\{packet_id\}pkt_tcp_FIN)$

$act15_1 : TCP_sessions := TCP_sessions \setminus \{src \mapsto dst, dst \mapsto src\}$

$act20_1 : src_ip := (\{packet_id\}src_ip)$

$act20_2 : dst_ip := (\{packet_id\}dst_ip)$

$act50_1 : target_state := FALSE$

$act50_2 : terminate := TRUE$ // NEXT EVENT TERMINATE

$act_fw20_2 : fw_machine := FALSE$

$act_fw20_1 : fw_action := drop$

END

firewall_exit

WHEN

$grd_fw20_1 : fw_machine = TRUE$

$grd_fw52_2 : fw_exit = TRUE$

THEN

$act_fw20_2 : fw_machine := FALSE$

$act_fw52_2 : fw_exit := FALSE$

END

packet_arrive

ANY _a _reply _last_transit x y _terminate _remove_pkt

WHERE

$grd00_1 : src \neq 0 \land dst \neq 0$

$grd00_2 : _reply \in BOOL$

$grd00_3 : _last_transit \in BOOL$
$grd00_4 : x \in NODES \land y \in NODES$
$grd00_5 : is_reply = FALSE$
$grd00_6 : last_transit = FALSE \Rightarrow (_terminate = FALSE \land _reply = TRUE \land _last_transit \in \{FALSE, TRUE\} \land x = src \land y = dst)$ // msg arrives
$grd00_7 : last_transit = TRUE \Rightarrow (_terminate = TRUE \land _reply = FALSE \land _last_transit = FALSE \land x = 0 \land y = 0)$ // reply to msg arrives
$grd00_8 : _a = src \mapsto dst$
$grd05_1 : exit_chokepoint = TRUE$
$grd10_1 : _remove_pkt = \{p \mid p = packet_id \land (\exists t \cdot t \in pkt_protocol \land t = p \mapsto UDP)\} \cup \{p \mid p = packet_id \land (\exists t \cdot t \in pkt_protocol \land t = p \mapsto TCP) \land (\exists s, a, f \cdot s \in pkt_tcp_SYN \land a \in pkt_tcp_ACK \land f \in pkt_tcp_FIN \land ((s = p \mapsto 0 \land a = p \mapsto 1 \land f = p \mapsto 0) \lor (s = p \mapsto 0 \land a = p \mapsto 0 \land f = p \mapsto 0 \land is_reply = TRUE)))\}$
$grd_fw20_1 : fw_machine = FALSE$
$grd_fw20_2 : fw_action = pass$

THEN

$act00_1 : last_transit := _last_transit$
$act00_2 : src, dst := x, y$ // null packet?
$act00_3 : is_reply := _reply$
$act05_1 : exit_chokepoint := FALSE$
$act10_1 : pkt_protocol := _remove_pktpkt_protocol$
$act10_2 : pkt_tcp_ACK := _remove_pktpkt_tcp_ACK$
$act10_3 : pkt_tcp_SYN := _remove_pktpkt_tcp_SYN$
$act10_4 : pkt_tcp_FIN := _remove_pktpkt_tcp_FIN$
$act20_1 : src_ip := _remove_pktsrc_ip$ // remove pkt if this is the reply (arrived = TRUE)
$act20_2 : dst_ip := _remove_pktdst_ip$ // or if UDP pkt
$act50_1 : target_state := FALSE$
$act50_2 : terminate := _terminate$

END

packet_reply

ANY _a x y _last_transit _pkt_tcp_ACK _pkt_tcp_SYN _pkt_tcp_FIN

WHERE

$grd00_1 : is_reply = TRUE$
$grd00_2 : x = dst \land y = src$
$grd00_3 : _a = dst \mapsto src$
$grd00_4 : _last_transit \in \{TRUE, FALSE\}$
$grd10_1 : pkt_protocol \neq \varnothing$
$grd10_2 : _pkt_tcp_ACK \subseteq PKT_TCP_ACK$
$grd10_3 : _pkt_tcp_SYN \subseteq PKT_TCP_SYN$
$grd10_4 : _pkt_tcp_FIN \subseteq PKT_TCP_FIN$
$grd15_1 : _pkt_tcp_SYN = \{packet_id + 1\}x\ \{r \mid r \in \{1, 0\} \land (\exists p \cdot p \in pkt_protocol \land p = packet_id \mapsto TCP) \land (\exists s, a \cdot s \in pkt_tcp_SYN \land a \in pkt_tcp_ACK \land ((s = packet_id \mapsto 1 \land a = packet_id \mapsto 0 \land r = 1) \lor (s = packet_id \mapsto 1 \land a = packet_id \mapsto 1 \land r = 0) \lor (s = packet_id \mapsto 0 \land r = 0)))\}$
$grd15_2 : _pkt_tcp_ACK = \{packet_id + 1\}x\ \{r \mid r \in \{1, 0\} \land (\exists p \cdot p \in pkt_protocol \land p = packet_id \mapsto TCP) \land r = 1\}$
$grd15_3 : _pkt_tcp_FIN = \{packet_id + 1\}x\ \{r \mid r \in \{1, 0\} \land (\exists p \cdot p \in pkt_protocol \land p = packet_id \mapsto TCP) \land (\exists f, a, s \cdot f \in pkt_tcp_FIN \land a \in pkt_tcp_ACK \land s \in$

$pkt_tcp_SYN \land ((f = packet_id \mapsto 1 \land a = packet_id \mapsto 0 \land r = 1) \lor (f = packet_id \mapsto 1 \land a = packet_id \mapsto 1 \land r = 0) \lor (f = packet_id \mapsto 0 \land r = 0)))\}$

$grd15_4 : _last_transit = bool((packet_id) \mapsto TCP \in pkt_protocol \land (((packet_id + 1) \mapsto 0 \in _pkt_tcp_FIN \land (packet_id + 1) \mapsto 0 \in _pkt_tcp_SYN \land (packet_id + 1) \mapsto 1 \in _pkt_tcp_ACK) \lor ((packet_id + 1) \mapsto 0 \in _pkt_tcp_FIN \land (packet_id + 1) \mapsto 0 \in _pkt_tcp_SYN \land (packet_id + 1) \mapsto 0 \in _pkt_tcp_ACK)))$

THEN

$act00_1 : src, dst := x, y$
$act00_2 : is_reply := FALSE$
$act00_3 : last_transit := _last_transit$
$act05_1 : enter_chokepoint := TRUE$
$act10_1 : packet_id :\mid packet_id' = packet_id + 1$
$act10_2 : pkt_protocol := (pkt_protocol \setminus \{r \mid r \in pkt_protocol \land (\exists n \cdot n \in PROTOCOL_TYPE \land r = packet_id \mapsto n)\}) \cup (\{packet_id + 1\} x \{n \mid n \in PROTOCOL_TYPE \land (\exists r \cdot r \in pkt_protocol \land r = packet_id \mapsto n)\})$
$act10_3 : pkt_tcp_ACK := _pkt_tcp_ACK$
$act10_4 : pkt_tcp_SYN := _pkt_tcp_SYN$
$act10_5 : pkt_tcp_FIN := _pkt_tcp_FIN$
$act20_1 : src_ip := (src_ip \setminus \{r \mid r \in src_ip \land (\exists n \cdot n \in NODES \land r = packet_id \mapsto n)\}) \cup (\{packet_id + 1\} x \{n \mid n \in NODES \land (\exists r \cdot r \in dst_ip \land r = packet_id \mapsto n)\})$
$act20_2 : dst_ip := (dst_ip \setminus \{r \mid r \in dst_ip \land (\exists n \cdot n \in NODES \land r = packet_id \mapsto n)\}) \cup (\{packet_id + 1\} x \{n \mid n \in NODES \land (\exists r \cdot r \in src_ip \land r = packet_id \mapsto n)\})$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := FALSE$

END

packet_dropped

ANY _a

WHERE

$grd00_1 : src \neq 0 \land dst \neq 0$
$grd00_2 : _a = src \mapsto dst$
$grd_fw20_2 : fw_action = drop$

THEN

$act00_1 : src, dst := 0, 0$ // null packet
$act00_2 : is_reply := FALSE$
$act00_3 : last_transit := FALSE$
$act05_1 : enter_chokepoint := FALSE$
$act05_2 : exit_chokepoint := FALSE$
$act10_1 : pkt_protocol := (\{packet_id\} pkt_protocol)$
$act10_2 : pkt_tcp_SYN := (\{packet_id\} pkt_tcp_SYN)$
$act10_3 : pkt_tcp_ACK := (\{packet_id\} pkt_tcp_ACK)$
$act10_4 : pkt_tcp_FIN := (\{packet_id\} pkt_tcp_FIN)$
$act15_1 : TCP_sessions := TCP_sessions \setminus \{src \mapsto dst, dst \mapsto src\}$
$act20_1 : src_ip := (\{packet_id\} src_ip)$
$act20_2 : dst_ip := (\{packet_id\} dst_ip)$
$act50_1 : target_state := FALSE$
$act50_2 : terminate := TRUE$ // NEXT EVENT TERMINATE

END

terminate

WHEN

$grd50_2 : terminate = TRUE$

THEN

$act50_1 : target_state := TRUE$

$act50_2 : terminate := FALSE$

$act50_3 : session_closed := FALSE$

END

END

Appendix C

MBT generated tests

C.1 Initial tests

Tests generated from a breadth first exploration of the initial models state space in ProB, detailing a naive firewall and a well behaved environment.

<?xml version="1.0" encoding="UTF-8"?>
<extended_test_suite>
<test_case id="1">
<global>
<step id="1">ext_dangerous_packet_to_int </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_dropped </step>
<step id="5">terminate </step>
</global>
</test_case>
<test_case id="2">
<global>
<step id="1">ext_trusted_packet_to_int </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_arrive </step>
<step id="5">packet_reply </step>
<step id="6">firewall_rule </step>
<step id="7">packet_arrive </step>
<step id="8">packet_reply </step>
<step id="9">firewall_rule </step>
<step id="10">packet_arrive </step>
<step id="11">terminate </step>
</global>
</test_case>
<test_case id="3">
<global>
<step id="1">ext_untrusted_packet_to_int </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_dropped </step>
<step id="5">terminate </step>

</global>
</test_case>
<test_case id="4">
<global>
<step id="1">int_packet_to_ext_dangerous </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_dropped </step>
<step id="5">terminate </step>
</global>
</test_case>
<test_case id="5">
<global>
<step id="1">int_packet_to_ext_trusted </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_arrive </step>
<step id="5">packet_reply </step>
<step id="6">firewall_rule </step>
<step id="7">packet_arrive </step>
<step id="8">packet_reply </step>
<step id="9">firewall_rule </step>
<step id="10">packet_arrive </step>
<step id="11">terminate </step>
</global>
</test_case>
<test_case id="6">
<global>
<step id="1">int_packet_to_ext_untrusted </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_arrive </step>
<step id="5">packet_reply </step>
<step id="6">firewall_rule </step>
<step id="7">packet_arrive </step>
<step id="8">packet_reply </step>
<step id="9">firewall_rule </step>
<step id="10">packet_arrive </step>
<step id="11">terminate </step>
</global>
</test_case>
<test_case id="7">
<global>
<step id="1">int_packet_to_ext_trusted </step>
<step id="2">protocol_TCP_start </step>
<step id="3">firewall_rule </step>
<step id="4">packet_arrive </step>
<step id="5">packet_reply </step>
<step id="6">firewall_rule </step>
<step id="7">packet_arrive </step>

<step id=”8”>packet_reply </step>
<step id=”9”>firewall_rule </step>
<step id=”10”>packet_arrive </step>
<step id=”11”>int_packet_to_ext_trusted </step>
<step id=”12”>protocol_TCP_continue </step>
<step id=”13”>firewall_rule </step>
<step id=”14”>packet_arrive </step>
<step id=”15”>packet_reply </step>
<step id=”16”>firewall_rule </step>
<step id=”17”>packet_arrive </step>
<step id=”18”>terminate </step>
</global>
</test_case>
<test_case id=”8”>
<global>
<step id=”1”>int_packet_to_ext_trusted </step>
<step id=”2”>protocol_TCP_start </step>
<step id=”3”>firewall_rule </step>
<step id=”4”>packet_arrive </step>
<step id=”5”>packet_reply </step>
<step id=”6”>firewall_rule </step>
<step id=”7”>packet_arrive </step>
<step id=”8”>packet_reply </step>
<step id=”9”>firewall_rule </step>
<step id=”10”>packet_arrive </step>
<step id=”11”>int_packet_to_ext_trusted </step>
<step id=”12”>protocol_TCP_finish </step>
<step id=”13”>firewall_rule </step>
<step id=”14”>packet_arrive </step>
<step id=”15”>packet_reply </step>
<step id=”16”>firewall_rule </step>
<step id=”17”>packet_arrive </step>
<step id=”18”>packet_reply </step>
<step id=”19”>firewall_rule </step>
<step id=”20”>packet_arrive </step>
<step id=”21”>terminate </step>
</global>
</test_case>
<test_case id=”9”>
<global>
<step id=”1”>ext_dangerous_packet_to_int </step>
<step id=”2”>protocol_UDP </step>
<step id=”3”>firewall_rule </step>
<step id=”4”>packet_dropped </step>
<step id=”5”>terminate </step>
</global>
</test_case>
<test_case id=”10”>
<global>
<step id=”1”>ext_trusted_packet_to_int </step>

```
      <step id="2">protocol_UDP </step>
      <step id="3">firewall_rule </step>
      <step id="4">packet_arrive </step>
      <step id="5">terminate </step>
    </global>
  </test_case>
  <test_case id="11">
    <global>
      <step id="1">ext_untrusted_packet_to_int </step>
      <step id="2">protocol_UDP </step>
      <step id="3">firewall_rule </step>
      <step id="4">packet_dropped </step>
      <step id="5">terminate </step>
    </global>
  </test_case>
  <test_case id="12">
    <global>
      <step id="1">int_packet_to_ext_dangerous </step>
      <step id="2">protocol_UDP </step>
      <step id="3">firewall_rule </step>
      <step id="4">packet_dropped </step>
      <step id="5">terminate </step>
    </global>
  </test_case>
  <test_case id="13">
    <global>
      <step id="1">int_packet_to_ext_trusted </step>
      <step id="2">protocol_UDP </step>
      <step id="3">firewall_rule </step>
      <step id="4">packet_arrive </step>
      <step id="5">terminate </step>
    </global>
  </test_case>
  <test_case id="14">
    <global>
      <step id="1">int_packet_to_ext_untrusted </step>
      <step id="2">protocol_UDP </step>
      <step id="3">firewall_rule </step>
      <step id="4">packet_arrive </step>
      <step id="5">terminate </step>
    </global>
  </test_case>
</extended_test_suite>
```

C.2 Additional tests

```
  <test_case id="15">
    <global>
```

```
        <step id=″1″>int_packet_to_ext_untrusted </step>
        <step id=″2″>protocol_TCP_start </step>
        <step id=″3″>firewall_rule </step>
        <step id=″4″>packet_arrive </step>
        <step id=″5″>packet_reply </step>
        <step id=″6″>firewall_rule </step>
        <step id=″7″>packet_arrive </step>
        <step id=″8″>packet_reply </step>
        <step id=″9″>firewall_rule </step>
        <step id=″10″>packet_arrive </step>
        <step id=″11″>int_packet_to_ext_untrusted </step>
        <step id=″12″>protocol_TCP_continue </step>
        <step id=″13″>firewall_rule </step>
        <step id=″14″>packet_arrive </step>
        <step id=″15″>packet_reply </step>
        <step id=″16″>firewall_rule </step>
        <step id=″17″>packet_arrive </step>
        <step id=″18″>terminate </step>
    </ global>
</ test_case>
<test_case id=″16″>
    <global>
        <step id=″1″>int_packet_to_ext_untrusted </step>
        <step id=″2″>protocol_TCP_start </step>
        <step id=″3″>firewall_rule </step>
        <step id=″4″>packet_arrive </step>
        <step id=″5″>packet_reply </step>
        <step id=″6″>firewall_rule </step>
        <step id=″7″>packet_arrive </step>
        <step id=″8″>packet_reply </step>
        <step id=″9″>firewall_rule </step>
        <step id=″10″>packet_arrive </step>
        <step id=″11″>int_packet_to_ext_untrusted </step>
        <step id=″12″>protocol_TCP_finish </step>
        <step id=″13″>firewall_rule </step>
        <step id=″14″>packet_arrive </step>
        <step id=″15″>packet_reply </step>
        <step id=″16″>firewall_rule </step>
        <step id=″17″>packet_arrive </step>
        <step id=″18″>packet_reply </step>
        <step id=″19″>firewall_rule </step>
        <step id=″20″>packet_arrive </step>
        <step id=″21″>terminate </step>
    </ global>
</ test_case>
<test_case id=″17″>
    <global>
        <step id=″1″>int_packet_to_ext_untrusted </step>
        <step id=″2″>protocol_TCP_start </step>
        <step id=″3″>firewall_rule </step>
```

```
      <step id=”4”>packet_arrive </step>
      <step id=”5”>packet_reply </step>
      <step id=”6”>firewall_rule </step>
      <step id=”7”>packet_arrive </step>
      <step id=”8”>packet_reply </step>
      <step id=”9”>firewall_rule </step>
      <step id=”10”>packet_arrive </step>
      <step id=”11”>int_packet_to_ext_untrusted </step>
      <step id=”12”>protocol_TCP_continue </step>
      <step id=”13”>firewall_rule </step>
      <step id=”14”>packet_arrive </step>
      <step id=”15”>packet_reply </step>
      <step id=”16”>firewall_rule </step>
      <step id=”17”>packet_arrive </step>
      <step id=”18”>int_packet_to_ext_untrusted </step>
      <step id=”19”>protocol_TCP_finish </step>
      <step id=”20”>firewall_rule </step>
      <step id=”21”>packet_arrive </step>
      <step id=”22”>packet_reply </step>
      <step id=”23”>firewall_rule </step>
      <step id=”24”>packet_arrive </step>
      <step id=”25”>packet_reply </step>
      <step id=”26”>firewall_rule </step>
      <step id=”27”>packet_arrive </step>
      <step id=”28”>terminate </step>
   </ global>
</ test_case>
<test_case id=”18”>
   <global>
      <step id=”1”>int_packet_to_ext_untrusted </step>
      <step id=”2”>protocol_TCP_start </step>
      <step id=”3”>firewall_rule </step>
      <step id=”4”>packet_arrive </step>
      <step id=”5”>packet_reply </step>
      <step id=”6”>firewall_rule </step>
      <step id=”7”>packet_arrive </step>
      <step id=”8”>packet_reply </step>
      <step id=”9”>firewall_rule </step>
      <step id=”10”>packet_arrive </step>
      <step id=”11”>int_packet_to_ext_untrusted </step>
      <step id=”12”>protocol_TCP_finish </step>
      <step id=”13”>firewall_rule </step>
      <step id=”14”>packet_arrive </step>
      <step id=”15”>packet_reply </step>
      <step id=”16”>firewall_rule </step>
      <step id=”17”>packet_arrive </step>
      <step id=”18”>packet_reply </step>
      <step id=”19”>firewall_rule </step>
      <step id=”20”>packet_arrive </step>
      <step id=”21”>int_packet_to_ext_untrusted </step>
```

<step id=”22”>protocol_TCP_start </step>
<step id=”23”>firewall_rule </step>
<step id=”24”>packet_arrive </step>
<step id=”25”>packet_reply </step>
<step id=”26”>firewall_rule </step>
<step id=”27”>packet_arrive </step>
<step id=”28”>packet_reply </step>
<step id=”29”>firewall_rule </step>
<step id=”30”>packet_arrive </step>
<step id=”31”>terminate </step>
</ global>
</ test_case>
<test_case id=”19”>
<global>
<step id=”1”>int_packet_to_ext_untrusted </step>
<step id=”2”>protocol_TCP_start </step>
<step id=”3”>firewall_rule </step>
<step id=”4”>packet_arrive </step>
<step id=”5”>packet_reply </step>
<step id=”6”>firewall_rule </step>
<step id=”7”>packet_arrive </step>
<step id=”8”>packet_reply </step>
<step id=”9”>firewall_rule </step>
<step id=”10”>packet_arrive </step>
<step id=”11”>int_packet_to_ext_untrusted </step>
<step id=”12”>protocol_TCP_finish </step>
<step id=”13”>firewall_rule </step>
<step id=”14”>packet_arrive </step>
<step id=”15”>packet_reply </step>
<step id=”16”>firewall_rule </step>
<step id=”17”>packet_arrive </step>
<step id=”18”>packet_reply </step>
<step id=”19”>firewall_rule </step>
<step id=”20”>packet_arrive </step>
<step id=”21”>ext_untrusted_packet_to_int </step>
<step id=”22”>protocol_TCP_start </step>
<step id=”23”>firewall_rule </step>
<step id=”24”>packet_dropped </step>
<step id=”25”>terminate </step>
</ global>
</ test_case>

References

[1] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[2] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars polar lander fault identification using model-based testing," in *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pp. 128–135, IEEE, 2001.

[3] M. Stevens, B. Ng, D. Streader, and I. Welch, "Global and local knowledge in SDN," in *Telecommunication Networks and Applications Conference (ITNAC), 2015 International*, pp. 237–243, IEEE, 2015.

[4] "Relationship of SDN and NFV," Tech. Rep. ONF TR-518, Open Networking Foundation, Palo Alto, CA, 2015.

[5] A. D. Neto, R. Subramanyan, M. Vieira, G. H. Travassos, and F. Shull, "Improving evidence about software technologies: A look at model-based testing," *Software, IEEE*, vol. 25, no. 3, pp. 10–13, 2008.

[6] M. Utting and B. Legeard, *Practical model-based testing: A tools approach*. Morgan Kaufmann, 2010.

[7] R. V. Binder, "Model-based testing user survey: Results and analysis," *System Verification Associates. System Verification Associates*, 2011.

[8] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, *et al.*, "A NICE way to test OpenFlow applications.," in *NSDI*, vol. 12, pp. 127–140, 2012.

[9] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, p. 21, ACM, 2011.

[10] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 24–24, USENIX Association, 2012.

[11] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[13] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests from choreography models," in *Testing of Software and Communication Systems*, pp. 179–194, Springer, 2009.

[14] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 115–120, ACM, 2004.

[15] B. Carpenter and S. Brim, "RFC 3234 - Middleboxes: Taxonomy and issues," *Network Working Group. Ietf*, 2002.

[16] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 51–62, 2008.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[18] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[19] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *ACM SIGCOMM Computer Communication Review*, vol. 37, pp. 1–12, ACM, 2007.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[21] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, IEEE, 2010.

[22] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer.," in *Hotnets*, 2009.

[23] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 117–130, 2015.

[24] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of Network Function Virtualization," in *Proc. USENIX NSDI*, pp. 459–473, 2014.

[25] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[26] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, "An intent-based approach for network virtualization," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pp. 42–50, IEEE, 2013.

[27] F. J. Ros and P. M. Ruiz, "Five nines of southbound reliability in software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 31–36, ACM, 2014.

[28] D. Meyer, "The software-defined-networking research group," *Internet Computing, IEEE*, vol. 17, no. 6, pp. 84–87, 2013.

[29] E. Brewer, "CAP twelve years later: How the 'rules' have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[30] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 91–96, ACM, 2013.

[31] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 1–6, ACM, 2013.

[32] 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*, vol. 44, no. 3, pp. 87–95, 2014.

[33] T. Wood, K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang, "Toward a software-based network: Integrating software defined networking and network function virtualization," *Network, IEEE*, vol. 29, no. 3, pp. 36–41, 2015.

[34] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Passive and Active Measurement*, pp. 347–359, Springer, 2015.

[35] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 49–54, ACM, 2013.

[36] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes," *arXiv preprint arXiv:1409.7687*, 2014.

[37] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 7–12, ACM, 2012.

[38] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: State distribution trade-offs in software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 1–6, ACM, 2012.

[39] K. Phemius and M. Bouet, "OpenFlow: Why latency does matter," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pp. 680–683, IEEE, 2013.

[40] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 127–132, ACM, 2013.

[41] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 14, no. 6, pp. 1246–1259, 2006.

[42] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 1–6, ACM, 2014.

[43] H. Jamjoom, D. Williams, and U. Sharma, "Don't call them middleboxes, call them middlepipes," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 19–24, ACM, 2014.

[44] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 163–174, ACM, 2014.

[45] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," tech. rep., Technical Report, 2013.

[46] A. Gember, R. Grandl, J. Khalid, and A. Akella, "Design and implementation of a framework for software-defined middlebox networking," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 467–468, ACM, 2013.

[47] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with merlin," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 24, ACM, 2013.

[48] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 147–148, ACM, 2013.

[49] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes.," in *NSDI*, pp. 227–240, 2013.

[50] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Denf, *et al.*, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action," in *SDN and OpenFlow World Congress*, pp. 22–24, 2012.

[51] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, "Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 333–345, 2014.

[52] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, *et al.*, "Rollback-recovery for middleboxes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 227–240, ACM, 2015.

[53] G. Gibb, H. Zeng, and N. McKeown, "Initial thoughts on custom network processing via waypoint services," in *WISH-3rd Workshop on Infrastructures for Software/Hardware co-design, CGO*, 2011.

[54] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing network functionality," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 73–78, ACM, 2012.

[55] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 27–38, ACM, 2013.

[56] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "CloudNaaS: A cloud networking platform for enterprise applications," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 8, ACM, 2011.

[57] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, pp. 279–291, ACM, 2011.

[58] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, *et al.*, "Composing software defined networks.," in *NSDI*, pp. 1–13, 2013.

[59] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 19–24, ACM, 2013.

[60] C. Cascone, M. Bonolax, L. Pollini, D. Sanvito, B. G., and C. A., "OpenState: Platform-agnostic behavioral (stateful) forwarding via minimal OpenFlow extensions," in *ACM Sigcom symposium on SDN research*, ACM.

[61] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[62] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, p. 10, 2014.

[63] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy, "Flow processing and the rise of commodity network hardware," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 2, pp. 20–26, 2009.

[64] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pp. 610–614, IEEE, 2014.

[65] J. Yu, Z. Dong, and N. Chi, "1.96 tb/s (21 100 gb/s) OFDM optical signal generation and transmission over 3200-km fiber," *Photonics Technology Letters, IEEE*, vol. 23, no. 15, pp. 1061–1063, 2011.

[66] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pp. 7–13, IEEE, 2014.

[67] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, *et al.*, "Steering: A software-defined networking for inline service chaining," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pp. 1–10, IEEE, 2013.

[68] P. Quinn and J. Guichard, "Service function chaining: Creating a service plane via network service headers," *Computer*, no. 11, pp. 38–44, 2014.

[69] J. Blendin, J. Ruckert, N. Leymann, G. Schyguda, and D. Hausheer, "Position paper: Software-defined network service chaining," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pp. 109–114, IEEE, 2014.

[70] W. Ding, W. Qi, J. Wang, and B. Chen, "OpenSCaaS: An open service chain as a service platform toward the integration of SDN and NFV," *Network, IEEE*, vol. 29, no. 3, pp. 30–35, 2015.

[71] D. S. Johnson, *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[72] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee, "No more middlebox: Integrate processing into network," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 459–460, 2011.

[73] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible open middleboxes with commodity servers," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pp. 49–60, ACM, 2012.

[74] 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*, pp. 13–18, ACM, 2014.

[75] J. Shah, "Implementation and performance analysis of firewall on Open vSwitch.," tech. rep., Technische Universitt Mnchen, 2015.

[76] "OpenFlow switch specification," Tech. Rep. ONF TS-025, Open Networking Foundation, Palo Alto, CA, 2015.

[77] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 254–265, ACM, 2011.

[78] N. Freed, "RFC-2979: Behaviour of and requirements for Internet firewalls," *Network Working Group, Internet Engineering Task Force (IETF)*, 2000.

[79] S. Karen and H. Paul, "Guidelines on firewalls and firewall policy," *NIST Recommendations, SP*, pp. 800–41, 2008.

[80] E. D. Zwicky, S. Cooper, and D. B. Chapman, *Building Internet firewalls*. " O'Reilly Media, Inc.", 2000.

[81] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet security: Repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[82] S. M. Bellovin and W. R. Cheswick, "Network firewalls," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 50–57, 1994.

[83] R. Bragg, K. Rhodes-Ousley, and M. Strassberg, "The complete reference: Network security," 2004.

[84] J. Wang, Y. Wang, H. Hu, Q. Sun, H. Shi, and L. Zeng, "Towards a security-enhanced firewall application for OpenFlow networks," in *Cyberspace Safety and Security*, pp. 92–103, Springer, 2013.

[85] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FlowGuard: Building robust firewalls for software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 97–102, ACM, 2014.

[86] M. Suh, S. H. Park, B. Lee, and S. Yang, "Building firewall over the software-defined network controller," in *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, pp. 744–748, IEEE, 2014.

[87] J. G. V. Pena and W. E. Yu, "Development of a distributed firewall using software defined networking technology," in *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, pp. 449–452, IEEE, 2014.

[88] 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*, pp. 525–528, IEEE, 2014.

[89] K. Kaur, K. Kumar, J. Singh, and N. S. Ghumman, "Programmable firewall using software defined networking," in *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pp. 2125–2129, IEEE, 2015.

[90] K. Kaur and J. Singh, "Building stateful firewall over software defined networking," in *Information Systems Design and Intelligent Applications*, pp. 159–168, Springer, 2016.

[91] A. Shukhman, P. Polezhaev, Y. Ushakov, L. Legashev, V. Tarasov, and N. Bakhareva, "Development of network security tools for enterprise software-defined networks," in *Proceedings of the 8th International Conference on Security of Information and Networks*, pp. 224–228, ACM, 2015.

[92] C. Yoon, T. Park, S. Lee, H. Kang, S. Shin, and Z. Zhang, "Enabling security functions with SDN: A feasibility study," *Computer Networks*, vol. 85, pp. 19–35, 2015.

[93] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 374–385, ACM, 2011.

[94] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 19–24, ACM, 2012.

[95] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[96] J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented systems," *Technical Reprot, University of Klagenfurt, Austria*, 1994.

[97] C. Baier, J.-P. Katoen, *et al.*, *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[98] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[99] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[100] E. Murphy-Hill and D. Grossman, "How programming languages will co-evolve with software engineering: A bright decade ahead," in *Proceedings of the on Future of Software Engineering*, pp. 145–154, ACM, 2014.

[101] J.-R. Abrial and J.-R. Abrial, *The B-book: Assigning programs to meanings*. Cambridge University Press, 2005.

[102] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. REX Workshop, Mook, The Netherlands, May 29-June 2, 1989. Proceedings*, vol. 430. Springer Science & Business Media, 1990.

[103] J.-R. Abrial, *Modeling in Event-B: System and software engineering*. Cambridge University Press, 2010.

[104] K. Stobie, "Model based testing in practice at microsoft," *Electronic Notes in Theoretical Computer Science*, vol. 111, pp. 5–12, 2005.

[105] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proceedings of the 27th international conference on Software engineering*, pp. 392–401, ACM, 2005.

[106] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. De Groot, and E. Lear, "RFC1918: Address alloaction for private internets," tech. rep., IANA Network Working Group, 1996.

[107] V. Cardellini, M. Colajanni, and S. Y. Philip, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, p. 28, 1999.

[108] R. Wang, D. Butnariu, J. Rexford, *et al.*, "OpenFlow-based server load balancing gone wild," Hot-ICE, 2011.

[109] D. Joseph and I. Stoica, "Modeling middleboxes," *Network, IEEE*, vol. 22, no. 5, pp. 20–25, 2008.

[110] M. Butler and S. Hallerstede, "The Rodin formal modelling tool," in *BCS-FACS Christmas 2007 Meeting-Formal Methods In Industry, London.*, 2007.

[111] R.-J. Back and F. Kurki-Suonio, "Distributed cooperation with action systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 4, pp. 513–554, 1988.

[112] B. W. Boehm, "Software engineering economics," *Prentice-Hall Advances in Computing Science and Technology Series, Englewood Cliffs: Prentice-Hall*, vol. 1, 1981.

[113] M. Canini, D. Kostic, J. Rexford, and D. Venzano, "Automating the testing of OpenFlow applications," in *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering (WRiPE)*, no. EPFL-CONF-167777, 2011.

[114] T. S. Hoang, D. Basin, H. Kuruma, and J.-R. Abrial, "Development of a network topology discovery algorithm," 2009.

[115] M. Kang, E.-Y. Kang, D.-Y. Hwang, B.-J. Kim, K.-H. Nam, M.-K. Shin, and J.-Y. Choi, "Formal modeling and verification of SDN-OpenFlow," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pp. 481–482, IEEE, 2013.

[116] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking SDN controllers.," in *FMCAD*, pp. 145–148, Citeseer, 2013.

[117] D. L. Dvorak *et al.*, "Nasa study on flight software complexity," *NASA office of chief engineer*, 2009.

[118] NASA Engineering and Safety Center, "Technical support to the NHTSA on the reported Toyota Motor Corporation unintended acceleration investigation.," tech. rep., NASA, 2011.

[119] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[120] J.-L. Lions *et al.*, "Ariane 5 Flight 501 Failure, report by the enquiry board," tech. rep., CNES, Paris, 1996.

[121] H. Robinson, "Obstacles and opportunities for model-based testing in an industrial software environment," in *Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pp. 118–127, 2003.

[122] K. Stobie, "Too darned big to test," *Queue*, vol. 3, no. 1, pp. 30–37, 2005.

[123] A. Hartman and K. Nagin, "The AGEDIS tools for model based testing," *Lecture Notes in Computer Science*, vol. 3297, pp. 277–280, 2005.

[124] W. Greiskamp, "Model-Based Testing: Theory and practice," in *Proceedings of the 4th Workshop on Model-based Testing in Practice (MoTiP), Dallas, USA*, 2012. `http://www.slideshare.net/wgrieskamp/keynote-motip-issre-2012`.

[125] M. Stevens and R. Norman, "Industry expectations of soft skills in IT graduates," in *18th Australasian Computing Education Conference (ACE 2016)*, 2016.

[126] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," *Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, Technical Report SCE-10-04*, 2010.

[127] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pp. 31–36, ACM, 2007.

[128] P. A. Geroski, "Models of technology diffusion," *Research policy*, vol. 29, no. 4, pp. 603–625, 2000.

[129] A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyás, W. Tavernier, and S. Sahhaf, "ESCAPE: Extensible service chain prototyping environment using Mininet, Click, Netconf and POX," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 125–126, 2015.

[130] R. Braden, "Rfc 1122: Requirements for internet hosts – communication layers," *Internet Engineering Task Force, SRI International*, 1989.

[131] M. Wasserman and P. Seite, "Rfc 6419: Current practices for multiple-interface hosts," *Internet Engineering Task Force, SRI International*, 2011.

[132] M. De Vivo, E. Carrasco, G. Isern, and G. O. de Vivo, "A review of port scanning techniques," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, pp. 41–48, 1999.

[133] W. Eddy, "RFC4987 - TCP SYN flooding attacks and common mitigations," *http://tools. ietf. org/html/rfc4987*, 2007.

[134] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX network programming*, vol. 1. Addison-Wesley Professional, 2004.

[135] S. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *Reliability, IEEE Transactions on*, vol. PP, no. 99, pp. 1–12, 2015.

[136] I. Alsmadi and D. Xu, "Security of software defined networks: A survey," *Computers & Security*, 2015.

[137] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[138] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[139] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[140] A. Hakiri, A. Gokhale, P. Berthou, D. C. Schmidt, and T. Gayraud, "Software-defined networking: Challenges and research opportunities for future Internet," *Computer Networks*, vol. 75, pp. 453–471, 2014.

[141] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and OpenFlow: From concept to implementation," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 4, pp. 2181–2206, 2014.

[142] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges," *Computer Networks*, vol. 72, pp. 74–98, 2014.

[143] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 4, pp. 1955–1980, 2014.

[144] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[145] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using OpenFlow: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 1, pp. 493–512, 2014.

[146] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti, *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[147] P. Ranjan, P. Pande, R. Oswal, Z. Qurani, and R. Bedi, "A survey of past, present and future of software defined networking," *International Journal*, vol. 2, no. 4, 2014.

[148] S. Rowshanrad, S. Namvarasl, V. Abdi, M. Hajizadeh, and M. Keshtgary, "A survey on SDN, the future of networking," *Journal of Advanced Computer Science & Technology*, vol. 3, no. 2, pp. 232–248, 2014.

[149] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 27–51, 2014.

[150] T. A. Limoncelli, "OpenFlow: A radical new idea in networking," *Queue*, vol. 10, no. 6, p. 40, 2012.