Protecting Against Compromised Controllers in Software Defined Networks Using an Efficient Byzantine Fault Preventing Control Plane

Joseph Gardiner

School of Computing and Communications

Lancaster University

August 2022

A thesis submitted to Lancaster University for the degree of Doctor of Philosophy in the Faculty of Science and Technology

Abstract

Software Defined Networking (SDN) is a modern approach to computer networks that involves the separation of the control and forwarding planes. Using this approach, control is achieved through the use of an SDN controller, which enables the delivery of far more intelligent, efficient and resilient networks.

Whilst the use of an SDN controller offers many potential benefits, the centralisation of network control introduces a single point of failure - if the SDN controller develops a fault, or is under attack, then the network can be severely disrupted. From a security perspective, the SDN controller represents a tempting target for an attacker - if the attacker can gain control over the controller then they can act as a malicious insider, gaining control over the operation of the whole network. The actions of a compromised SDN controller can be seen as an occurrence of byzantine (or arbitrary) faults. By introducing a byzantine fault tolerant (BFT) element to the control plane, insider attacks can be prevented.

This thesis explores the impact of a compromised SDN controller, and provides a defence called SDBFT: Software Defined Byzantine Fault prevenTing control. I reduce fault tolerance to fault preventing, which means fault detecting with recovery. SDBFT prevents a compromised SDN controller from performing malicious actions in a network. Within this thesis, I first analyse and demonstrate a number of attacks that can be performed from a compromised controller, including an exploration of the impact of such attacks on a real-world scenario involving Industrial Control Systems (ICS). I then propose, implement and evaluate the SDBFT system, using novel algorithms

that are able to protect against faulty controllers. I demonstrate through extensive experimentation that the SDBFT system far outperforms approaches built upon a traditional BFT model, and only represents a modest reduction in controller performance compared to the traditional SDN architecture.

Acknowledgements

I would first like to thanks my PhD supervisors, Dr. Peter Garraghan and Dr. Nicholas Race, who took over supervision duties on the departure of my former supervisor, Dr. Shishir Nagaraja. I am especially grateful to Shishir for the initial encouragement to pursue a PhD and the extensive support he has offered me during the initial years of my PhD, and beyond.

I would also like to thank the colleagues who have supported me during this process, in particular those who gave me encouragement to complete this thesis even during tough times.

Finally, I would like to thank the family and friends who have given me support during this process, and in particular my parents, Pauline and Michael, who have supported me every step of the way.

Declaration

This thesis is my own work and no portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification at this or any other institute of learning. It is the result of my own work with the exception of manuscript revision by the supervisors Peter Garraghan, Nicholas Race and Shishir Nagaraja.

Contributing Publications

Joseph Gardiner, Adam Eiffert, Peter Garraghan, Nicholas Race, Shishir Nagaraja, and Awais Rashid. 2021. **Controller-in-the-Middle: Attacks on Software Defined Networks in Industrial Control Systems** In Proceedings of the 2nd Workshop on CPS&IoT Security and Privacy (CPSIoTSec '21), ACM, New York, NY, USA, 63–68. DOI:10.1145/3462633.3483979

Contents

List of Figures **xi**

List of Tables **xiii**

List of Acronyms **xv**

1 Introduction **1**

1.1 Background 1

1.1.1 The Rise of SDN 1

1.1.2 The Problem With SDN 2

1.1.3 Dealing With The Problems 5

1.2 Motivation 7

1.3 Aims 9

1.4 Contributions 9

1.4.1 Exploration of Attack Capabilities From a Compromised SDN Controller 10

1.4.2 Design of a Consensus-Based Distributed Controller Architecture to Prevent Malicious Insiders 11

1.5 Thesis Structure 12

2 Background **14**

2.1 Introduction 14

2.2 A History of Networking 15

2.2.1 The Evolution of the Network 15

2.2.2 Network Routing 16
2.3 Programmable Networks 18
2.4 Software Defined Networking 19
2.4.1 SDN Controller Architecture 20
2.4.2 SDN Deployment Architectures 22
2.4.3 Applications 23
2.4.4 Security of SDN 23
2.5 The OpenFlow Protocol 24
2.5.1 OpenFlow Packets 25
2.5.2 OpenFlow Handshake 26
2.5.3 Flow Tables 27
2.5.4 Other SDN Protocols 29
2.6 SDN Operation 30
2.6.1 Reactive 30
2.6.2 Proactive 30
2.6.3 Hybrid 31
2.7 SDN Controllers 32
2.7.1 Controller Placement 34
2.8 Dependability and Faults 35
2.8.1 Dependability 35
2.8.2 Faults 38
2.9 SDN Fault Tolerance 40
2.9.1 Native OpenFlow Support 40
2.9.2 ONOS 41
2.10 Consensus 41
2.11 Byzantine Fault Tolerance 44
2.11.1 Byzantine Faults 45
2.11.2 Byzantine Fault Tolerant Algorithms 47
2.12 Conclusion 56

3 Literature Review and Related Work 57
3.1 Introduction 57
3.2 Security of SDN 57
3.3 Detecting Compromised SDN Controllers 59
3.3.1 Limitations 61
3.4 Mitigating Attacks in SDN 61
3.4.1 Securing the Controller 61
3.4.2 Preventing Controller Poisoning 63
3.4.3 Protection Against Malicious Applications 64
3.4.4 Securing the Control-Data Plane (Southbound) Interface . 65
3.5 Multiple Controller SDN control 65
3.6 Primary-Backup Fault tolerant SDN control 67
3.7 Byzantine Fault Tolerant SDN Control 70
3.7.1 Consensus amongst administrators 73
3.8 Discussion 74
3.9 Conclusion 76

4 Insider Attacks in Software Defined Networks 78
4.1 Introduction 78
4.2 Attacker Model 79
4.2.1 Attacker 79
4.2.2 Attack Vector 79
4.2.3 Attacker Goals 81
4.3 Attacks 82
4.3.1 Denial of Service Attacks 83
4.3.2 Eavesdropping attacks 84
4.3.3 Data Tampering Attacks 85
4.3.4 Service Degradation 86
4.3.5 Other Attacks 86
4.4 Attack Demonstration 87
4.4.1 Setup 87
4.4.2 Results 92
4.4.3 Discussion 100

4.5 Real World Impact — Industrial Control Systems 103
4.5.1 Industrial Control Systems 103
4.5.2 Attacker 107
4.5.3 Setup 109
4.5.4 Attacks 112
4.5.5 Discussion 118
4.6 Conclusion 119

5 Designing An Efficient Consensus Approach for SDN Control 121
5.1 Introduction 121
5.2 System Overview 121
5.2.1 Requirements 123
5.2.2 Notation 125
5.3 Quorums 126
5.3.1 Quorum Size 126
5.4 SDBFT protocol 127
5.4.1 Assumptions 128
5.4.2 Normal Operation 130
5.4.3 Failure Operation 132
5.5 Signatures 137
5.5.1 Controller Verification 139
5.5.2 Limitations 139
5.6 Controller Assignment 140
5.6.1 Requirements 141
5.6.2 Simple Algorithm 142
5.6.3 Simple Algorithm Performance 145
5.6.4 Existing Approaches 147
5.7 Controller Consistency 148
5.7.1 Publisher-Subscriber Protocol 150
5.7.2 Existing Approaches 151
5.8 Limitations of Approach 152
5.8.1 Proactive Control 153
5.8.2 Controller Diversity 154

5.9 Conclusion 155

6 Implementing the SDBFT Protocol 156

6.1 Introduction 156
6.2 SDBFT Implementation Overview 156
6.3 Proxy Implementation 159
6.3.1 Configuration 159
6.3.2 Communication 160
6.3.3 Message Acknowledgements 160
6.3.4 Signatures 161
6.3.5 OpenFlow Message Handling 162
6.4 Controller modification 166
6.4.1 Message serialising/deserialising 168
6.4.2 Xid setting 168
6.4.3 Acknowledgement handling 168
6.4.4 Synchronisation 169
6.5 Implementation of Comparative System 170
6.5.1 Configuration 171
6.5.2 Protocol 171
6.5.3 Proxies 172
6.6 Conclusion 177

7 Experimental Setup 178

7.1 Testbeds 178
7.1.1 OpenVSwitch (OVS) Virtual Environment 178
7.1.2 Mininet 180
7.1.3 Physical Switch 182
7.2 Simple TCP Proxy 184
7.3 Measurement Tools 184
7.3.1 Ping 185
7.3.2 Cbench 185
7.4 Floodlight Configuration 187
7.4.1 Applications 188

7.5 Conclusion 189

8 Evaluating The SDBFT Controller Architecture 190

8.1 Introduction 190
8.1.1 Method of Analysis 191
8.2 Baselines 193
8.2.1 Setup 194
8.2.2 Results 196
8.2.3 Discussion 203
8.3 Multi hop path test 205
8.3.1 Setup 206
8.3.2 Results 206
8.3.3 Discussion 210
8.4 Failure operation 212
8.4.1 Setup 212
8.4.2 Results 213
8.4.3 Discussion 215
8.5 High Throughput Benchmark 217
8.5.1 Setup 217
8.5.2 Results 218
8.5.3 Discussion 223
8.6 Testing on physical switch 223
8.6.1 Setup 223
8.6.2 Results 224
8.6.3 Discussion 228
8.7 Deployment of Physical Proxy 228
8.7.1 Setup 228
8.7.2 Results 229
8.7.3 Discussion 230
8.8 Network Traffic Load 231
8.8.1 Setup 231
8.8.2 Results 232
8.8.3 Discussion 233

8.9 Conclusion . . . 234

9 Conclusion **236**

9.1 Thesis Contributions . . . 237

9.1.1 Exploration of Attack Capabilities From a Compromised SDN Controller . . . 237

9.1.2 Design of a Consensus-Based Distributed Controller Architecture to Prevent Malicious Insiders . . . 238

9.1.3 Research Impact . . . 241

9.1.4 Summary . . . 241

9.2 Future Work . . . 242

9.2.1 Proactive Control . . . 242

9.2.2 Controller Verification . . . 243

9.2.3 Anonymous Information Sharing . . . 243

9.2.4 Native Implementation of SDBFT . . . 244

9.3 Reproducibility . . . 244

9.4 Concluding Remarks . . . 245

Appendix A Implementation **246**

A.1 SDBFT Proxy Configuration . . . 246

Appendix B Evaluation Setup **248**

B.1 OVS Test Launch Script . . . 248

B.2 Mininet Python Configuration Example . . . 249

B.3 Bash Launch Script . . . 251

Appendix C Evaluation **253**

C.1 Baseline Results . . . 254

C.2 Multi-hop Path Test Results . . . 260

References **263**

List of Figures

2.1 SDN Architecture 20
2.2 OpenFlow Message Header 26
2.3 Openflow Handshake 27
2.4 Reactive Switch Control 31
2.5 SDN Controller Components 32
2.6 Floodlight Web Administrative Tnterface 34
2.7 Laprie Dependability Tree 36
2.8 Byzantine Siege 45
2.9 Byzantine Generals Problem 47
2.10 PBFT Algorithm 51
2.11 BFT-SMaRt Protocol 55

4.1 Floodlight Web Administrative Interface 80
4.2 Attack Taxonomy 83
4.3 Attack Demonstration Network 89
4.4 Amplified DOS Attack 96
4.5 Attack Demonstration Network — Redirect Attacls 97
4.6 Redirect Eavesdropping Attack 98
4.7 Redirect Traffic Attack Results 100
4.8 Purdue Reference Architecture 105
4.9 FactoryIO 110
4.10 ICS Devices 112
4.11 ICS Attack Demonstration Network Topologies 113

5.1 Traditional Controller Architecture 122

5.2 SDBFT Controller Architecture 122
5.3 Typical Switch to Controller Communication 129
5.4 Switch to Controller Consensus, Working State 129
5.5 Switch to Controller Consensus, Failure State 132
5.6 Signed Switch to Controller Communication 139
5.7 Controller Assignment Example 1 146
5.8 Controller Assignment Example 2 147

6.1 SDBFT Proxy Architecture 158
6.2 SDBFT Proxy Switch to Controller Message Handling 163
6.3 SDBFT Proxy Controller to Switch Message Handling 165
6.4 Controller System Model 167
6.5 BFT-SMaRt Proxy Architecture 173

7.1 Baseline Setup 180
7.2 Physical Network Setup Design 183
7.3 Physical Network Hardware 183

8.1 Baseline Network 194
8.2 Baseline Direct and Simple Proxy Results 196
8.3 Baseline Unsigned SDBFT and BFT-SMaRt Results 198
8.4 Baseline Signed SHA512withRSA SDBFT and BFT-SMaRt Results 200
8.5 Baseline Signed SHA256withRSA SDBFT and BFT-SMaRt Results 202
8.6 Multi-hop Direct and Simple Proxy Results 208
8.7 Multi-Hop Unsigned SDBFT and BFT-SMaRt Results 209
8.8 Multi-Hop Signed SDBFT and BFT-SMaRt Results 211
8.9 Fault Recovery Results, Single Fault 214
8.10 Fault Recovery Results, Three Faults 216
8.11 Physical Switch Topologies 224
8.12 Single Physical Switch Results 225
8.13 Three Physical Switch Results 227
8.14 Hardware Proxy Topology 229
8.15 Hardware Proxy Results 229

List of Tables

2.1 Common OpenFlow packets 28
2.2 A Selection of Commonly Used SDN Controllers 33

3.1 Existing literature in Byzantine Fault Tolerant SDN control . . . 75

4.1 Flow Rule Blocking Results 114

5.1 Notation 126

7.1 Summary of testbeds 179

8.1 Summary of experiments 192
8.2 Effect Size Classification 193
8.3 Fault Recover Results 215
8.4 CBench Output, Single Virtual Switch 219
8.5 CBench Output, Sixteen Virtual Switch 221
8.6 Physical Switch Results 226
8.7 Network Traffic Load Results 232

C.1 Baseline results without using signatures 254
C.2 Baseline Results Increasing Controller Counts 255
C.3 Baseline Results BFT-SMaRt versus SDBFT 255
C.4 Baseline Results Using SHA512 with RSA Signatures 256
C.5 Baseline Results Using SHA256 with RSA Signatures 257
C.6 Signed Baseline Results SHA512 with RSA Versus Unsigned . . . 258
C.7 Signed Baseline Results SHA256 with RSA Versus Unsigned . . . 259

C.8 Multi-hop Path Test Unsigned Results 260
C.9 Multi-hop Path Test Signed Results 261
C.10 Multi-hop Path Test Unsigned BFT-SMaRt Versus SDBFT 261
C.11 Multi-hop Path Test Signed BFT-SMaRt Versus SDBFT 262

List of Acronyms

BGP Border Gateway Protocol

CDPI Control-Data Plane Interface

DHT Distributed Hash Table

EGP Exterior Gateway Protocol

EIGRP Enhanced IGRP

ICS Industrial Control Systems

IDS Intrusion Detection System

IGRP Interior Gateway Routing Protocol

IPS Intrusion Prevention System

IP Internet Protocol

IS-IS Intermediate System to Intermediate System

ISPs Internet Service Providers

JVM Java Virtual Machine

NIB Network Information Base

NOS Network Operating System

OSPF Open Shortest Path First

RIP Routing Information Protocol

SDBFT Software-Defined Byzantine Fault Tolerant control

SDN Software Defined Network

TCP Transmission Control Protocol

UDP User Datagram Protocol

Chapter 1

Introduction

This thesis presents the design of a novel consensus-based byzantine fault-tolerant distributed architecture for a controller within an Software Defined Network (SDN), to prevent against insider attacks through compromised controllers. In this chapter, I provide a brief introduction to SDN and the issues in terms of security of the SDN paradigm that motivate the need for a fault-tolerant controller architecture, including the limitations of current approaches. I then outline my aims and key contributions and provide an overview of the structure of the rest of this thesis.

1.1 Background

1.1.1 The Rise of SDN

In recent years, the notion of programmable networks has become an area of increasing interest and research. In traditional networks the control and data planes resided on the same device, with control being relatively static and deterministic — the new model has shifted to moving the control plane into software, allowing for real-time control of network flows.

The most well-known programmable network model is Software Defined Networking (SDN), wherein switches are purely forwarding devices which operate

using a rule-based flow table which is populated through the use of an SDN controller. The controller can either pro-actively install flow rules to control future flows, or react to new flows seen on the switch and make routing decisions on the fly. An SDN controller can run multiple applications, providing specific functionality beyond normal routing, including load balancing, firewalls and traffic monitoring. Typically, a switch will receive a packet and will attempt to match it with a rule in its flow table. If matched, it will apply the action defined by the rule. If there is no match, the switch will forward it to the controller, which will then return a new flow rule (or set of rules) which will define some action, such as to forward the flow to a particular port or drop all packets. SDN most commonly refers to the specific OpenFlow [148] protocol for switch-controller communication used to establish SDN networks, providing the interface between the switch and controller, as well as the specification for how the switch operates. Whilst OpenFlow is already in use at major organisations, other protocols do exist.

Google uses a self-built high-speed SDN network for handling inter data centre communications [104], while the NSA uses a locked down version of the Ryu controller to control their intra-net[1]. SDN networks are also increasingly commonly paired with network function virtualisation (NFV) technologies to provide the routing mechanism within datacentres and cloud environments, where a dynamic network is required to keep up with a rapidly changing topology.

1.1.2 The Problem With SDN

Typically in an SDN, a switch is controlled by a single controller, with one controller responsible for many switches. In a small network environment, one controller can easily control the entire network. Controllers are often large, complex applications running on standard hardware and operating systems (for example most Openflow controllers run as java or Python programs in user space on Linux). Whereas; before routing decisions were made on the switch, meaning it would be difficult, though not impossible, for an attacker to have an influence over routing decisions, the control plane, and the decision making for routing,

[1] https://www.networkworld.com/article/2937787/nsa-uses-openflow-for-tracking-its-network.html

now operates on a general purpose computer which may well be connected to the internet or other internal enterprise systems. This single point of failure becomes a high-value target for an attacker [60, 207, 210, 206].

Modern operating systems are large, complex systems, and with that comes the risk of vulnerabilities and exploits. Zero-day exploits can remain undetected by defenders for months after being discovered by cyber criminals, and sold on the black market [27, 149]. Older, well known vulnerabilities, while fixed by vendors, can remain an issue for years following discovery on unpatched machines, such as the critical EternalBlue exploit that still exists years after its discovery [178]. These exploits can give attackers full administrative control over a machine, often with very little evidence of their presence. SDN controllers run as processes on traditional operating systems, so it is therefore feasible for an attacker to gain control of a machine running an SDN controller (e.g. by exploiting a vulnerability in the operating system or other software installed on the system), or at the very least gain the ability to intercept switch-controller communications on the controller host, allowing them some degree of control over the network. Similarly, with the increasing complexity of SDN controllers comes an increasing risk of a vulnerability in the controller, or one of its dependencies. For example, in December 2021 a major vulnerability, Log4Shell, was found in the log4j java logger library[1], which is utilised by many enterprise Java applications, including SDN controllers such as the Floodlight controller used in this work. This vulnerability allowed the attacker to connect to external services and potentially download, and run, arbitrary code and affected hundreds, if not thousands, of software projects that utilised the log4j library.

There are a wide range of actions an attacker could carry out if they were in control of the SDN controller and routing decisions. The obvious examples include denial-of-service attacks, wherein the attacker drops the network flows of a particular target host, or all hosts. In a similar fashion, the attacker can break the network through the introduction of illegal topologies, such as loops. These sort of attacks are relatively easy to detect and prevent utilising existing approaches which look for such topological errors in SDN control commands [111, 112, 7]. A more persistent attacker could perform more intelligent attacks that

[1] CVE-2021-44228

do not cause obvious issues with the network, but affect it in other ways. For example, an attacker could add a few extra hops on the paths of a particular flow to introduce higher latency. While this may not seem like a major issue, in a scenario such as a financial institution engaging in ultra-low latency trading, where transactions need to happen very quickly (in the order of microseconds), a delay of even a few milliseconds over the expected could result in major issues such as trades being carried out after prices have changed [10]. An attacker who has control over flows could facilitate person-in-the middle attacks by routing a target flow through a machine that they own without the sending and receiving parties realising.

In the past, the primary threat in cyber-security was individuals carrying out relatively simple attacks for the fun and to earn respect amongst their peers. There are still have low-impact threats — script-kiddies — inexperienced and less-skilled attackers who use off-the-shelf malware or simple, widely available tools to carry out simple attacks such as distributed denial-of-service (DDoS), or credential harvesting. These types don't usually have a specific target; if they can't get into a system easily they will move on. Now there is, however, a growing threat from advanced attackers, namely criminal gangs and nation state attackers. These new threats are well resourced, highly skilled, persistent and targeted, which has led to the term Advanced Persistent Threat (APT). These attackers work with the goal of stealing information, and disrupting critical infrastructure. Two well known examples of this include attacks against western defence companies from the east (it is suspected the Chinese J-20 jet fighter is based on stolen plans of the USA's F-22 and F-35 fighters[1]), or the (supposedly) American and Israeli attack on Iranian nuclear enrichment plants in the form of Stuxnet [110]. Such attackers can gather large amounts of intelligence about their targets, discover their own zero day exploits, which are unknown even to the software developers, and write single-use pieces of malware which can evade detection. While a script-kiddie is unlikely to attack an SDN network, it is well within reason that an APT level attacker would put the time and effort into

[1] `https://www.cnbc.com/2017/11/08/chinese-theft-of-sensitive-us-military-technology-still-huge-problem.html`

attacking the SDN control plane in order to assist in their other operations due to the control that such a network would allow.

1.1.3 Dealing With The Problems

One approach to dealing with this problem is to detect attacks once they have occurred, then react and repair the damage later, through the deployment of a specialised Intrusion Detection System (IDS) or Intrusion Prevention System (IPS). This is common within the host-based malware detection field, with the caveat you often cannot detect the malware until it has already infected the machine and started performing malicious actions. In a networking environment, a large delay in detecting a malicious action may be a major problem — too large a delay and the attack can be completed before it is detected. Because many network flows are short lived, damage can be done even if there is a delay of just a few seconds between an attack starting and a response occurring. For example, this could be enough time for an attacker to redirect an important request to a malicious server to disrupt a critical user request. This detection approach requires an accurate data stream from the controller in order to apply detection, which could be difficult to guarantee if the attacker has full control. For example, the controller could report taking one action, but actually perform another.

The second approach is to build protection into the controller architecture which prevents attacks from occurring at all. This is akin to the network intrusion prevention system (IPS) model, where a detector sits on the network gateway and blocks any flows which exhibit malicious, or unusual, behaviour. In an SDN setting, this would require some system that sits between the switch and controller, or on the switch itself, in order to analyse controller outputs — making decisions about whether or not to accept the result. This would have to occur outside of the controller, and its host, due to the risk of compromise. Applying IPS-like autonomous detection can be a costly operation (including the potential for false positives), and could reduce controller performance to the point that the network suffers.

A compromised SDN controller is effectively a malicious insider within the network control architecture. For a discussion on how an SDN controller can be compromised see Chapter 4. One way to look at insider attacks within SDN is to consider the attacks as introducing faults to the network, with a compromised controller being seen as a faulty service, causing byzantine (arbitrary) failures within the network. A compromised controller can return a response to a switch request which does not match the expected outcome, causing traffic to be routed in an incorrect manner. A controller architecture that is resilient to insider attacks could then be considered fault-tolerant if it is able to operate normally despite compromised nodes. Within the distributed systems space there has been a large amount of work designing protocols that enable systems to operate in the presence of faults (see Section 2.11), ensuring that even if a portion of the nodes in a distributed system are faulty the system operates without fault (in the best case handling f faulty nodes with $2f + 1$, or more commonly $3f + 1$, nodes). In the simplest fault tolerant protocol a client will make a request to a group of servers, which then respond with a majority vote mechanism deciding on the correct response. There has been work to provide fault tolerance within SDN, however this is usually limited to fail-stop failures of a single controller (where the controller goes offline either temporarily or permanently), and control simply transitions to a backup controller which has access to a distributed datastore containing switch state information shared with the primary and so can efficiently gain control of the switch with existing knowledge of its state (see Sections 3.6 and 3.7).

What if we move to a distributed control architecture, where a switch communicates with multiple controllers at the same time and applies a fault tolerant algorithm to prevent faults from occurring? This has the potential to prevent almost all attacks resulting from compromised controllers, although introduces some possible issues:

- Adding extra communication steps increases the latency of routing decisions being made, degrading network performance.
- Introducing multiple controllers increases the required computing cost for the control plane.

- The network design becomes far more complex. For example, how are controllers assigned to switches?
- How do we ensure controllers are in sync, such that they can reliably agree on actions for specific flows?
- How are faulty or compromised controllers dealt with?

Typically, byzantine fault tolerant (BFT) algorithms require at least $3f + 1$ (with some optimised algorithms reducing this to $2f+1$) nodes in order to provide fault tolerance, where f is the maximum number of faulty nodes. The protocols usually require at least 4 rounds of communications, including multiple broadcasts. For example, if I take perhaps the most well known algorithm, Practical Byzantine Fault Tolerance, or PBFT [41], this requires 5 rounds of communication, including two rounds where all nodes broadcast to all other nodes. PBFT also requires a node to take the role of *leader*, which is the primary node that the client contacts, and then replicates the request to other members. If this leader is not operating properly, there is a large time penalty in choosing a new leader which could become unacceptable. A compromised leader could also modify, or even drop, client requests if requests are not signed by the client. The key requirement of the existing algorithms is fault tolerance — as long as there are fewer than f faulty nodes the algorithm will always handle the faults as part of the protocol and complete fully. This is good for fault tolerance, but bad for performance due to the latency and communication overhead of such algorithms. If I assume a scenario where the servers are non-faulty for the majority of the time, then the extra communication steps to handle faults that do not occur represent a large cost in terms of performance which is unacceptable in a low latency scenario such as SDN control.

1.2 Motivation

Previous work around fault tolerance within SDN architectures has largely focused around handling fail-stop failures, where the primary controller no longer responds to requests from switches. The typical approach is to utilise a backup

controller which takes over control when required. This approach is suitable for scenarios where a controller fails completely, either through genuine fault or through a Denial-of-Service (DoS) attack, however it can not necessarily handle byzantine faults introduced either through genuine fault, or through a more advanced attacker, such as malicious routing decisions. Further, as I show in Chapter 4, a particularly skilled attacker could inject malicious flow rules into the network which are subtle enough to not adversely impact upon network performance, meaning that detection is difficult.

One approach is to make use of a byzantine fault tolerant (BFT) architecture which can handle arbitrary faults both from genuine faults and those caused by an attacker [136, 137, 76, 158]. In these approaches, $3f+1$ are required to handle f faulty controllers — as long as no more than f controllers are faulty then the fault is handled and the switch is correctly updated. However, this comes with a large additional overhead as BFT algorithms are typically costly, requiring large amounts of replication to provide $3f + 1$ controllers per switch and operating over multiple rounds of communication. This additional overhead translates to additional latency when reacting to new flows. Whilst on a single hop path this extra latency could be negligible, over a typical many-hop path this latency becomes an issue.

From a network engineer's perspective, who would want the network to perform optimally, the trade-off of large amounts of extra latency for extra security may not be worth the additional complexity. It is well known that security systems that have a noticeable impact on users tend to not be used (as is often the case with the tradeoff of *usability* vs *security* [35, 123, 165, 241]). This led me to explore an approach that can still provide levels of protection that, in the case of no fault, operates with a minimal impact on network performance, and whilst under attack performs at a worst case equivalent to the traditional BFT approach.

1.3 Aims

In this work I design a control architecture for SDN in which a switch communicates with multiple controllers at the same time, with a simple, and efficient, byzantine fault-tolerant algorithm to provide agreement amongst controllers. I relax the requirements of a traditional byzantine fault-tolerant algorithm to handle faults within the protocol itself — I instead aim to identify when a fault has occurred and then enter a failover protocol to handle it, rather than handle directly in the initial run of the protocol. In this design, a switch directly communicates with a quorum of controllers (without the use of a leader), and then directly receives the responses back from all controllers. If all the controllers return the same result, the switch accepts this result and updates the flow table. If there is any disagreement in the responses (including any arbitrary difference), then the switch enters a recovery node and rejects the response, picking a new quorum of controllers from the pool. This represents a trade-off over traditional fault-tolerant protocols, as in the non-faulty case control requires far less communication than PBFT (two rounds instead of five) and is hence more efficient, but a slower recovery from fault than if it was directly handled within the initial run of the protocol. This approach has the additional benefit of requiring only $2f+1$ controllers, with $f+1$ used during normal operation with a further f backup controllers required when a fault is detected. An efficient BFT algorithm usually requires at least $3f+1$ nodes to provide fault tolerance. For example, a five node system will tolerate one faulty node. In this system, a five node system will be able to identify that a fault has occurred even if four of five controllers are faulty.

1.4 Contributions

In this section I outline the contributions of this thesis.

1.4.1 Exploration of Attack Capabilities From a Compromised SDN Controller

I provide an analysis of the capabilities of an attacker who has gained some degree of control of an SDN controller. I begin by evaluating the attacker model, including the method of controller compromise and the goals of such an attacker. I then define a set of attacks that can be launched from a compromised SDN controller, with a focus on attacks which cause modifications to the data plane through the use of malicious flow rules. This is a combination of attacks from the literature, as well as a number of novel attacks that to the best of my knowledge are explored for the first time in this thesis. In many works which focus on the security of controllers, the exploration of attacks that can be launched is limited and so I aim to provide a focus on the potential impact of a compromised SDN controller.

1.4.1.1 Practical Demonstration of Attacks

I implement the described attacks through a set of malicious applications for the Floodlight SDN controller [187], and show their impact on a simulated network using Mininet [130]. The impact of these attacks range from simple denial of service, to less obvious attacks which introduce additional latency into network communication, or support person-in-the-middle attacks. Whilst existing works have implemented a subset of the individual attacks, to the best of my knowledge this is the first piece of work to implement and evaluate the impact of multiple attacks which modify the data plane from a compromised control plane.

1.4.1.2 Impact of Attacks on Industrial Control Systems

I examine the impact of the described attacks in a real-world setting of Industrial Control Systems (ICS). Within ICS, the safety of systems is paramount and an increasing use of real-time ethernet-based communication requires resilient networking. SDN is being increasingly proposed for use in industrial networks, in

part due to the benefits to security and resilience afforded through reactive network control, however there has been minimal discussion of the negative security aspects of SDN in such environments. I further develop my attacker model for the ICS case, and test the impact of my attacks on a combined physical and simulated testbed utilising real-world ICS devices and protocols, a physical SDN switch and a simulated physical process. I show that even simple attacks which introduce a small amount of additional latency can fully disable real-time industrial network protocols, and cause disruption to others.

To the best of my knowledge, this is the first piece of work that explores the impact of a compromised SDN controller within the context of industrial control systems. This work has been published as "Controller-in-the-Middle: Attacks on Software Defined Networks in Industrial Control Systems" [87].

1.4.2 Design of a Consensus-Based Distributed Controller Architecture to Prevent Malicious Insiders

I propose a novel consensus-based distributed architecture for byzantine fault tolerant SDN control, which I refer to as SDBFT (Software Defined Byzantine Fault prevenTing control). SDBFT is a lightweight protocol for applying consensus within SDN controllers in order to identify, and handle, byzantine faults, whilst requiring only $f + 1$ primary controllers to handle f faulty or malicious controllers (reverting to $2f + 1$ through the addition of f backup controllers on the occurrence of a fault), compared to the $3f+1$ used by comparable approaches which utilise a traditional byzantine fault tolerant approach. The primary difference between SDBFT and the traditional BFT approach is that when using BFT, all $3f + 1$ controllers are used to handle all requests, whilst when using SDBFT only the $f + 1$ primary quorums handle requests, whilst the remaining f backup capacity can be reserved on other controllers. Whereas traditional BFT algorithms can prevent faults from happening as part of the core protocol, with little difference in operation between the faulty and non-faulty state, I relax fault tolerance to fault-identification with recovery, which allows for a far more efficient protocol, with fewer communication steps to be used in the non-faulty scenario.

This is achieved through the use of a system where $f + 1$ primary controllers are used if there is no disagreement amongst controllers, with the introduction of an additional f backup controllers when disagreement occurs. The SDBFT architecture is able to provide byzantine fault tolerance for SDN control with a low additional packet processing time, and far fewer messages than comparative systems using a traditional BFT approach, making it more viable for use in real-world networks with a trade-off of a slight delay when handling faults. Whereas existing works focus on one aspect of the problem, such as controller assignment, this thesis represents thee first approach which covers all aspects of the potential deployment including controller assignment, signature use and controller consistency.

1.4.2.1 Implementing and Evaluating the SDBFT Architecture

I implement SDBFT using the Floodlight controller, and a switch proxy to replicate the switch-side logic. I evaluate the performance of SDBFT using three testbeds, including a Mininet environment, an OpenVSwitch virtual network and a physical testbed using commercial SDN switches. I perform a number of tests to measure the performance and resilience of SDBFT, compared to a non modified controller and a traditional BFT based approach which closely resembles the related work. In particular this includes an evaluation of the performance during the occurrence of fault, which is missing in existing work. This is the most substantial evaluation of a fault-tolerant (or preventing) control plane to demonstrate the practical viability of such approach. I show that the SDBFT architecture provides fault tolerance with far less overhead than a traditional BFT approach.

1.5 Thesis Structure

The thesis is structured into nine chapters. In Chapter 2, I provide a background on SDN and fault tolerance. I describe the history of SDN, describe common architectures and provide details on the most common SDN protocol — OpenFlow. I give an introduction to consensus and describe byzantine fault tolerance.

In Chapter 3, I provide a literature review on the security of SDN, and cover various techniques for providing security and fault tolerance within SDN controller architectures.

In Chapter 4, I explore attacks against software-defined networks that can be launched from a compromised SDN controller. This includes discussing the goals and attack vectors of the attacker and an exploration of multiple attacks, some from the literature and some new. I then look at a specific real-world use case of SDN in the form of industrial control systems, and measure the impact of a subset of the attacks against a number of industrial protocols.

Chapter 5 describes the design of my consensus-based fault-tolerant SDN control architecture, SDBFT, and I discuss my implementation of this architecture in Chapter 6.

In Chapter 7, I describe my experimental setup for measuring the performance and resilience of SDBFT, describing three simulated, virtual-networking and physical switch based testbeds, along with the tools I utilise.

Using the testbeds developed in Chapter 7, in Chapter 8 I then evaluate the SDBFT protocol utilising a number of tests, including baseline performance experiments, high-throughput benchmarks and fault recovery time.

Finally in Chapter 9 I discuss future work and summarise my key contributions and impact of my work.

Chapter 2

Background

2.1 Introduction

In this chapter I provide a brief history of networking and the development of programmable networks. I then give an overview of Software Defined Networking (SDN) and its operation, with a focus on the OpenFlow protocol. I then introduce the concepts of dependability and byzantine fault tolerance.

In Section 2.2 I provide historical context with a history of the development of networks, and the development of traditional routing algorithms. In Section 2.3 I then explore the concept of programmable networks which aims to provide far greater control of networks than traditional network paradigms and routing algorithms. I then examine the focus of this work, SDN, in Section 2.4, going into detail on the architectures, applications and uses of SDN, and then in Section 2.5 I go into detail on the most widely-known SDN protocol, OpenFlow.

Section 2.6 explores the different modes of operation of SDN control, which is followed by an overview of SDN controllers in Section 2.7. I also provide an overview of the currently implemented fault-tolerance in SDN controllers in Section 2.9.

Finally, Sections 2.8 to 2.11 introduce the concepts of dependability and faults, consensus and byzantine fault tolerance.

2.2 A History of Networking

2.2.1 The Evolution of the Network

The concept of a computer network in which two computers communicate with each other was first proposed by J. C. R. Licklider in his 1960 paper "Man-Computer Symbiosis" [138]:

> *"It seems reasonable to envision, for a time 10 or 15 years hence, a "thinking center" that will incorporate the functions of present-day libraries together with anticipated advances in information storage and retrieval and the symbiotic functions suggested earlier in this paper. The picture readily enlarges itself into a network of such centers, connected to one another by wide-band communication lines and to individual users by leased-wire services. In such a system, the speed of the computers would be balanced, and the cost of the gigantic memories and the sophisticated programs would be divided by the number of users."*

Shortly afterwards in October 1962 Licklider became the first head of the computer research program at the Advanced Research Projects Agency (ARPA, now known as the Defence Advanced Research Projects agency, or DARPA), where he pushed research into his networking concept [134]. During this time, Leonard Kleinrock at MIT introduced the concept of packet switching (where individual packets are addressed and routable) [113], which would lead fellow MIT researcher Lawrence G Roberts and Thomas Merril to first computer network in 1965, connecting two computers over a low speed dial-up telephone line.

In September 1969, ARPA launched ARPANET, the network that would eventually become what we now know as the internet [134]. In October 1969 the first message was sent between computers at the Stanford Research Institute (SRI) and the University of California, Los Angeles (UCLA). By the end of 1969, the

University of California, Santa Barbara (UCSB) and the University of Utah were also connected to the ARPANET, growing it to four site.

As the ARPANET grew, multiple other networks were developed across the world (such as the Merit and CYCLADES networks), however these different networks were not compatible with each other, and so in 1974 Bob Kham of DARPA and Vint Cerf of Stanford University published the "Transmission Control Program", a protocol that provided all of the transport and forwarding functionality for the internet (coincidentally, this was the first time the word "internet" was used) [44]. In 1978, version three of the Transmission Control Program separated the singular protocol protocol into two protocols - the Internet Protocol (IP) for addressing and the forwarding of single packets, and the Transmission Control Protocol (TCP) for providing additional features, including the ability to handle lost packets. The User Datagram Protocol (UDP) was also made available at this time. Around the same time, Ethernet was developed as a physical layer protocol by researchers at Xerox PARC, primarily led by Robert Metcalfe [154], and standardised as IEEE 802.3 in 1983.

2.2.2 Network Routing

One of the first widely used routing protocols was the Routing Information Protocol (RIP), originally developed by Charles Hendrick in 1982 and standardised in 1988 [95]. RIP is an example of a distance-vector routing protocol, in which routes are determined based on distance, in this case the number of hops. Distance-vector based protocols are based upon the concept of routing tables, which contains information about the routes to networks, that are shared amongst routers along with other network information. Routing is performed by identifying the next router which would have the shortest path to the target, and forwarding the packet onto it. RIP attempts to prevent network loops occurring by limiting the number of hopes a packet can take to 15. This can reduce the chance of loops, but has the downside that it limits the protocol to smaller networks as packets cannot travel more than 15 hops.

Another distance vector routing algorithm emerged in 1986 in the form of the proprietary Interior Gateway Routing Protocol (IGRP) from Cisco [52], later

followed by the Enhanced IGRP (EIGRP) protocol in 1993 [51]. IGRP removes the 15 hop limit experienced by RIP, allowing routed up to 255 hops in length, which allows for much larger networks that RIP. EIGRP builds on IGRP to support classless IP addressing, and therefore subnets. Further, EIGRP reduces the amount of information exchanged between routers by only transmitting updates to routing tables.

Link-state routing protocols are the alternate to distance-vector routing protocols, first described in the 1970s for use in ARPANET [151, 150]. In link-state routing protocols, each node (switch or router) builds up a map of the network showing the connectivity between nodes. This graph can then be used to calculate the shortest path to the destination, for example through the use of Dijkstra's algorithm, which is able to find the shortest path between two nodes in a graph with weighted edges [67]. Whereas in distance-vector routing algorithms routers exchange routing tables with each other, in link-state protocols switches and routers only exchange information on their neighbours. Two prominent examples of link-state routing protocols are the Open Shortest Path First (OSPF) and Intermediate System to Intermediate System (IS-IS) protocols.

OSPF, originally designed in the 1980s, is a link-state routing protocol which uses a form of Dijkstra's routing algorithm for finding the shortest path to destinations [161]. As well as the distance between routers, the protocol also considers other factors including link throughput and availability. Version 2 of the protocol added support for IPv4 [162], with support for IPv6 also added later [81]. OSPF is still widely used today, in particular within enterprise networks. IS-IS is another link-state routing protocol that emerged around the same time as OSPF, and is very similar, also relying on Dijkstra's algorithm [177]. IS-IS is commonly used as the routing protocol in the networks managed by Internet Service Providers (ISPs).

The Exterior Gateway Protocol (EGP) [157], later followed by the Border Gateway Protocol (BGP) [141], are examples of exterior gateway protocols. Whereas all of the previously described protocols are interior gateway protocols, designed for routing within a network, exterior gateway protocols are designed to provide routing between networks (or autonomous systems). BGP is an example

of a path-vector routing algorithm, in which path information is shared between routers. An important aspect of BGP is that is relies heavily on network-administrator configured routed. BGP is the backbone of the modern internet, with failures in BGP, in particular those caused by misconfiguration causing major outages on substantial portions of the internet [167, 85].

2.3 Programmable Networks

Software Defined Networking (SDN) is just one example of the concept of programmable networks, in which the control plane, which represents the intelligence used for making routing decisions (using the algorithms described above) and the data planes, which is responsible for the actual forwarding of packets on the network, are separated (previously both functions existed within a switch itself). Before the emergence of SDN, and in particular the OpenFlow protocol, there was a rich history in the development of programmable networks [80, 11].

Before SDN, a similar but disjoint concept was developed in *active networks*, the first work coming from Tennenhouse and Wetherall in 1996 [224]. The concept of active networks is that programs can be injected into the nodes of the network [225]. In the Tennenhouse and Wetherhall approach this is done by replacing traditional network packets with "capsules" — small programs that are executed at each switch as they pass through them. This custom code brought programmability to the network through the packets themselves, for example including code which dictates the next hop the packet will take. The alternate to the capsule model is the programmable router/switch model, where the code was loaded onto switches and routers through an out-of-band channel [25]. Whilst the active networks concept, in particular the capsule model, did not gain much traction past the 1990s, the programmable switch approach is still in use, with the P4 language being an example under active development [29].

One of the earliest works exploring the separation of the control and data planes was Tempest, an architecture for providing the capability for multiple asynchronous transfer mode (ATM) networks to be hosted on the same physical

switches [153]. As part of this work, the idea of software controllers which would allow control of packet forwarding was considered.

ForCES (Forwarding and Control Element Separation) is an early interface for communication between the control and data planes, standardised by the Internet Engineering Task Force (IETF) in 2004 [238]. Whilst ForCES allows for the separation of the control and data planes as is the core principle of SDN, the difference to the modern SDN architecture is that both planes still exist on a single device, on separate devices in very close proximity. This was used to develop SoftRouter, which used the ForCES API to facilitate the communication between routers which maintained a forwarding table with a separate controller. ForCES struggled with vendor adaption and was eventually abandoned [80]. Shortly afterwards the Routing Control Platform further explored the concept of logically centralised control, however this used the existing BGP protocol for modifying the forwarding tables of routers [80].

Ethane (which is based upon the previously proposed Sane [40]) is the first approach which follows the modern SDN approach of a centralised controller and flow tables on switches [39]. Ethane makes use of a logically centralised controller for administering level access control, with a focus on enterprise networks. The development of Ethane directly influenced and led to the development of the OpenFlow protocol, primarily due to the deployment of Ethane at Stanford University.

2.4 Software Defined Networking

The first work describing SDN, and in particular the OpenFlow protocol, was published by McKeown et al. at Stanford University in 2008 [148]. This work details the design of the SDN switch, controller and OpenFlow protocol. The driving motivation of this work was experimentation — the SDN architecture would allow researchers to run experimental networks alongside production networks on the same hardware. Many early works on SDN focus on the separation of research and production networks [214, 215]. After being deployed on multiple

campus networks, the OpenFlow architecture was adopted within industry, becoming popular for use within datacentres [89, 80]. Whilst most commonly used within individual networks, specifically enterprise and data centre networks [169], there have also been deployments of software-defined wide-area networks (SD-WANSs) [240, 156, 139]. A well known example of a SDWAN is B4, which is a private WAN owned by Google and used to manage the connection between their datacentres [104]. B4 utilises logically centralised applications to control routing, with a control plane built upon the Onix distributed SDN controller [115].

The OpenFlow protocol is now standardised and promoted by the Open Networking Foundation [173]. I give a deeper overview of the OpenFlow protocol in Section 2.5. For the remainder of this work, I focus on the OpenFlow-based architecture for SDN.

2.4.1 SDN Controller Architecture

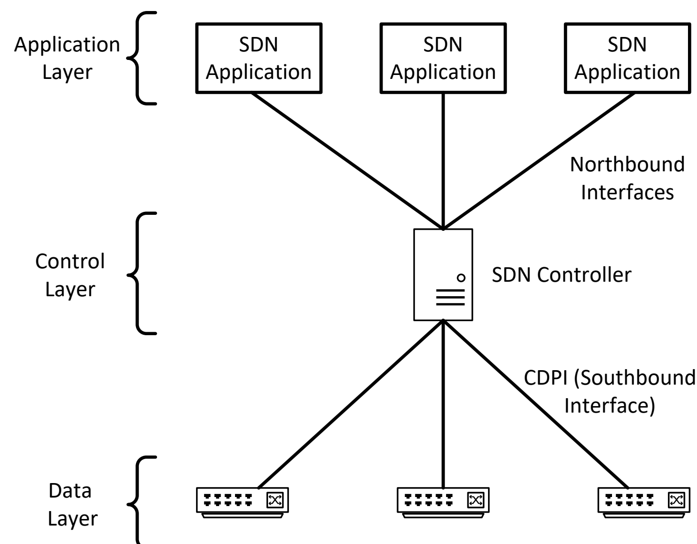


Figure 2.1: SDN Architecture

In the SDN model, the network is split into three layers: the data plane, control plane and applications [79, 120, 146].

Data Plane The interconnected forwarding devices, either physical or virtual, within the network (switches). These are the devices that perform the actual forwarding within the network. The core components of data plane devices are a set of forwarding engines, and a Control-Data Plane Interface (CDPI) agent for handling communication with the control plane.

Control Plane The control plane, usually referred to as the SDN controller or Network Operating System (NOS) is the logically centralised component responsible for communicating with data plane devices over the CDPI, including translating commands from applications into flow rules useable by the date plane, and providing a network view to the applications. A single controller will communicate with multiple data plane devices, and the control plane may be made up of multiple distributed, but logically centralised, controller instances.

Applications (Management Plane) The application layer consists of a set of applications for performing network functions, such as forwarding, load balancing and firewalls. Applications communicate with the controller using the northbound interface, including providing instruction to the controller about actions that need to be taken within the network and receiving network state information from the controller to use in application logic.

Communications between the layers is provided through the control-data plane interface (CDPI, also commonly called the southbound interface) and the controller-application interface (commonly referred to as the northbound interface or API).

Control-data plane interface (CDPI)/Southbound interface The CDPI allows communication between the switches of the data plane, and the SDN controller. The CDPI will specify how the switches and controllers communicate, and will also dictate the instruction set of the forwarding device. The CDPI provides functionality including the sending of switch events from the switch to the controller, providing control of forwarding operations and collecting statistics from the data plane. The most common CDPI is OpenFlow (as discussed in Section 2.5).

Controller-application interface/ Northbound interface The northbound interface is used by applications to communicate with the SDN controller. The northbound interface provides a layer of abstraction for the applications over the low level instructions used by the data plane, with the controller converting the high level instructions from the application. Whilst there is no de-facto standard for the northbound interface (as OpenFlow is to the CDPI), the northbound interface is most commonly implemented as a RESTful API.

There has been further attempts to provide a level of abstraction over SDN in order to make the programming of networks easier. An example of this is Frenetic, which is a high-level network programming language which can be used for programming distributed collections of switches, which provides a layer of abstraction over the SDN network below (in the provided example implemented with the NOX controller) [84]. The language supports querying the network and dictating traffic forwarding policies, which are translated into low-level flow rules onto switches by a custom run-time system on top of the SDN controller.

2.4.2 SDN Deployment Architectures

An SDN deployment can take one of three architectures: centralised, distributed and hierarchical. Each style of deployment has benefits and downsides which I will briefly discuss.

Centralised In the centralised architecture, the entire network is controlled by a single, centralised controller which maintains a complete view of the network. Whilst this has the advantage that it is the easiest deployment, it provides a severe limitation in terms of scalability, as well as provides a single point of failure.

Distributed In the distributed approach, multiple controllers each control a local partition of the network, with only knowledge of their local network view. Approaches can be taken to synchronise information between controllers to expand upon this view. Whilst this provides greater scalability, the limited network view of an individual controller limits large-scale coordination across

the network. I discuss a number of proposed distributed controller architectures in Section 3.5.

Hierarchical In the hierarchical architecture, distributed controllers control a local partition of the network, however a logically centralised root controller with a global network view is used for making certain routing decisions.

2.4.3 Applications

One of the core components of the Software Defined Network is the applications that run on or communicate with the controller in order to provide extended functionality to the controller to perform specific functions. These can either be part of the core SDN controller, or run as external services communicating with the controller over an API. An SDN controller will typically come pre-loaded with a set of applications which perform core functionality, such as routing, as well as further applications for more advanced features. For example, the Floodlight SDN controller comes with pre-built applications for forwarding, load balancing, firewall and static flow pushers, amongst others [188].

As well as the default applications, the majority of the open SDN controllers support user-designed apps, with most controllers providing programming guidelines for developers. One of the advantages of SDN applications is that they can be used in a modular fashion and shared amongst users. As expected, this led to the emergence of SDN application stores, with HP being the first to launch an SDN application store in 2014 [99][1].

2.4.4 Security of SDN

Whilst centralising network control provides many benefits, this introduces a single point of failure, as well as a single target for attackers to focus on to gain both control and monitoring over a target network [60].

[1] At the time of writing, I was unable to find access to the HP app store, with the only discoverable links being dead. It is unclear if it is still in operation as I was also unable to find any references indicating its closure.

Kreutz et al. defines seven threat vectors for software defined networks [119]. These include direct attacks on switches and controllers, as well as the various communication channels between components of the SDN. Whilst some of these, such as the compromise of switches, is not specific to SDN, the impact of such threats can potentially be augmented though the use of SDN.

There are numerous approaches that can be taken to compromise or otherwise negatively effect an SDN controller. These can include launching poisoning attacks against the network view held by the SDN controller [98, 168, 64, 230], installing malicious SDN applications [235, 205] or directly compromising the controller [194]. I discuss these approaches more in Chapter 4. Further, there has been a large amount of discussion of the impact of denial-of-service (DoS) type attacks against SDN controllers and switches [103, 69, 234, 108, 114, 244, 74, 70, 72, 71]. These DoS attacks can both send a large volume of requests to the controller, exhausting resources, or alternatively send a large number of new flows through a switch causing the flow table to be filled, requiring rules to be removed and increasing the load on the controller.

The use of SDN also introduces further opportunities for attackers to fingerprint networks to learn how which devices are connected and how the the network operates. Cui et al. demonstrate a fingerprinting attack against SDN to learn the packet-forwarding logic of the network, which can be performed both actively and passively [59]. Conti et al. leverage a scenario where the attacker can read the flow table on a switch in order to learn extensive detail about the configuration of the network, including the security policies in place [54]. Multiple approaches could be used by the attacker to learn the state of a flow table, including utilising a listening port on the switch [19], through default credentials on a switch [243], observing switch-controller communications [184] or directly compromising a controller.

2.5 The OpenFlow Protocol

When referring to SDN, most commonly the protocol in use for communicating between a switch and a controller is the OpenFlow protocol, managed by the

Open Networking Foundation[1], and originally developed by McKeown et al. at Stanford University [148]. The initial version of the protocol, Version 1.1, was released in 2011 [171]. The most recent publicly available version is Version 1.5.1, released in 2015 [175]. Version 1.6 of the protocol was released to ONF members in 2016. The most widely supported version of the protocol is version 1.3, released in 2012 [174]. As well as providing a specification for switch-controller communication, the OpenFlow standard also specifies the implementation of the switch in order to operate in OpenFlow networks, including the handling of the protocol and construction of flow tables.

The OpenFlow protocol operates over a TCP connection, and supports the use of Transport Layer Security (TLS) to secure the channel between the switch and controller, using port 6653 for communication.

2.5.1 OpenFlow Packets

Table 2.1 lists commonly used OpenFlow packets that are sent between a switch and controller. As can be seen, there are four main groups of packet types — those used in setting up a connection between a switch and controller, those used in switch control operations, messages to relay flow statistics from the switch and messages used for other specific purposes.

2.5.1.1 OpenFlow Packet Header

All OpenFlow packets contain the OpenFlow header, as shown in Figure 2.2. The header specifies the OpenFlow version, the message type (as seen in Table 2.1, represented by a numerical value), the total message length and a transaction id (`xID`), used for pairing replies to requests. The `xID` field is optional, and is often not implemented for all packet types on switches and controllers, defaulting to a value of 0.

Fields specific to the OpenFlow packet type will then follow the header. Note that some packet types, such as a `FeatureReq` message, will only consist of the

[1] `https://opennetworking.org`

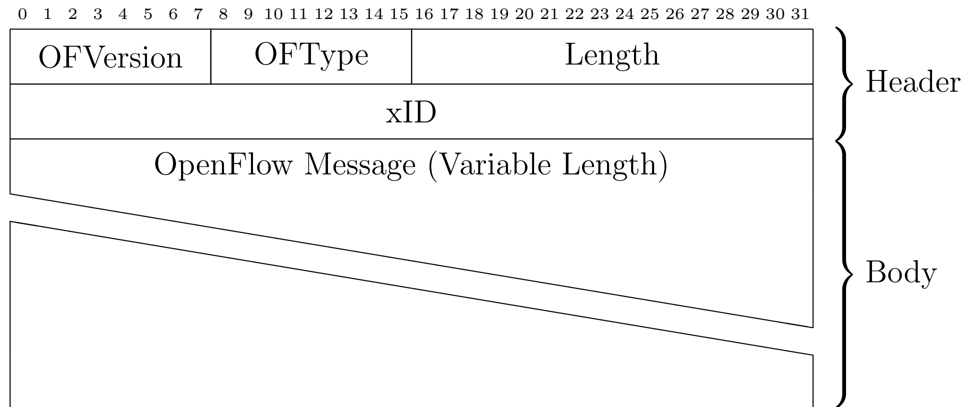


Figure 2.2: OpenFlow message header (Openflow 1.3 [174])

OpenFlow header, with the switch able to respond based on the message type field in the header alone.

2.5.2 OpenFlow Handshake

When a switch is launched, it will connect to the controller. Typically this connection is configured by IP and port number, often the default OpenFlow port of 6653. The switch will then establish a TCP connection to the controller.

Once the TCP connection is established, a simple handshake protocol, as instructed by the OpenFlow specification, is performed, as shown in Figure 2.3. The handshake begins with the sending of an OpenFlow `Hello` message, which can be initially sent by either the switch or controller. The receiver will respond with a second Hello message to confirm they are alive. The controller will then send a `FeaturesReq` to the switch, which is used to gather information on the switch. The switch will reply with a `FeaturesRes` message containing the appropriate information. This can include the number of `PacketIn` messages that can be buffered by the switch when sending requests to the controller, and the number of flow tables supported by the switch. Once the `FeaturesRes` has been received by the controller, the required portion of the handshake is complete, and the switch can start sending `PacketIn` requests to the controller.

There are further optional messages that can be sent between the switch and controller after the required handshake has been completed, which further con-

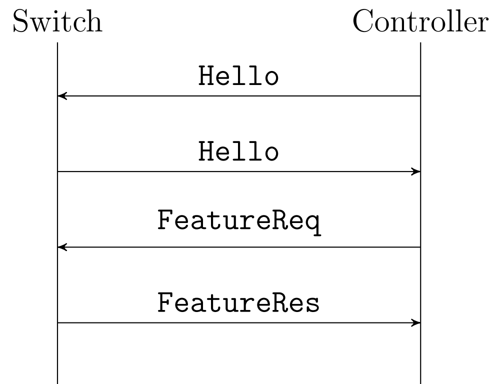


Figure 2.3: OpenFlow Handshake (Openflow 1.3 [174])

figure the switch. The `SetConfig` can be used to set configuration parameters on the switch, and a `GetConfigReq` and be used to query configuration parameters (returned through the matching `GetConfigRes`). The controller can use a `RoleReq` to set the current role of itself (Master, Slave or Equal, see Section 2.9.1). The switch must reply with a `RoleRes`, which follows the same structure as the request and confirms the current role of the controller.

2.5.3 Flow Tables

One of the core components of the SDN switch is the implementation of a set of flow tables. In the simplest case, a switch will maintain both a layer 3 and a layer 2 flow table [91]. The flow table consists of a set of rules which include match fields (for matching to a flow), action fields (dictating what to do with the packet) and a set of statistics representing the rule (such as number of packets). Flow rules can be set to expire after either a hard timeout, or after a certain amount of time with no traffic utilising the rule.

When implemented onto a physical switch, flow tables are often implemented in Content-Addressable Memories (CAMs), for layer 2 flow tables, and Ternary Content-Addressable Memories (TCAMs) for layer 3 tables [201, 91]. The memories have a fixed size, which introduces a limit on the size of flow tables. If the flow tables are full, then flow rules must be removed to introduce new rules into the tables. There are various strategies for doing this, including simple approaches.

Table 2.1: List of common OpenFlow packets sent between the controller (C) and switch (S) in OpenFlow 1.3

Group	PacketName	Direction	Description
Handshake	Hello	S→C,C→S	Sent at beginning of communication. Used for version negotiation
	FeatureReq	C→S	Requests an enumeration of the switchs' abilities
	FeatureRes	S→C	Response to FeatureReq
	GetConfigReq	C→S	Query fragment handling properties of packet handling pipeline
	GetConfigResp	S→C	Acknowledgement of GetConfigReq
	SetConfig	C→S	Set fragment handling properties of packet handling pipeline
	RoleReq	C→S	Request assignment of role (see Section 2.9.1)
	RoleRes	S→C	Returns role currently assigned to controller
Switch Operations	PacketIn	S→C	Used to send a packet from switch to controller
	PacketOut	C→S	Used to inject packet from controller into data plane
	FlowMod	C→S	Allows controller to modify state of switch
	FlowRemoved	S→C	Sent when entry in flow table is removed.
	PortStatus	S→C	Sent on change of status of a port
	GroupMod	C→S	Used to modify group tables
	PortMod	C→S	Used to modify state of OpenFlow port
	TableMod	C→S	Used to control what happens to packet when not matched in the flow table specified (or all tables)
Stats	MultipartReq	C→S	Used to request information (stats) about individual flows
	MultipartRes	S→C	Response to MultipartReq
Other	BarrierReq	C→S	Used for synchronisation. All state control operations sent from controller to switch must be performed before BarrierRes is sent
	BarrierRes	S→C	Response to BarrierReq
	Error	S→C,C→S	Indicates failure of an operation
	EchoReq	S→C,C→S	Simple packet for exchanging information about latency, bandwidth and liveness.
	EchoRes	S→C,C→S	Response to EchoReq

such as first-in-first-out. This limitation can also be utilised by attackers in order to perform denial-of-service attacks on the network, as these tables can be easily filled by generating large volumes of traffic with unique source and destination addresses, filling the table and requiring a large amount of controller interaction to handle flows [234, 233].

2.5.4 Other SDN Protocols

Whilst the majority of SDN implementations utilise the OpenFlow protocol, there are other protocols in use and being developed.

One example is the Interface to Routing System (I2RS) being developed by the Internet Engineering Task Force (IEFT) [14]. I2RS utilises existing routing algorithms between forwarding devices, including OSPF and BGP, however provides a southbound API allowing applications to modify routing tables. Also developed by the IETF is NETCONF, which allows for the remote configuration of switches [77].

OpFlex is an open source protocol developed by Cisco as a direct competitor to OpenFlow [219, 209]. A primary difference between OpFlex and OpenFlow is that whilst OpenFlow centralises all routing decisions to the controller, OpFlex instead uses the controller to enforce policies which are sent to network devices, which use local decision making to apply them.

The OpenVSwitch Management Database (OVSDB) protocol is equivalent to an SDN southbound API, specifically for configuring OpenVSwitch virtual network devices [182]. It is designed to be used in conjunction with OpenFlow to manage the virtual switches themselves including functions such as creating bridges and adding ports.

2.6 SDN Operation

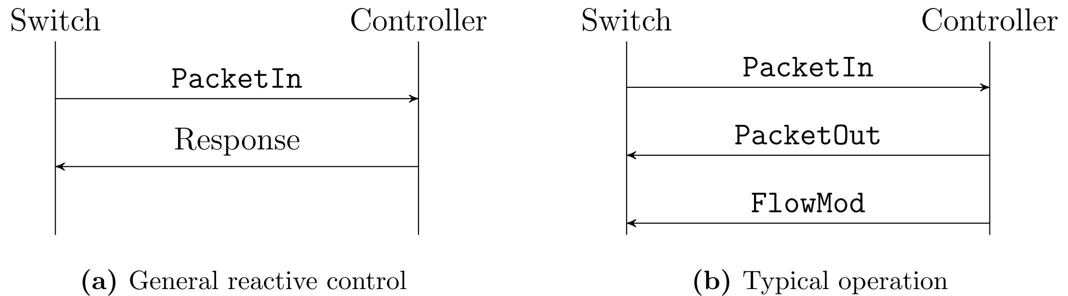
2.6.1 Reactive

In the reactive approach, on receiving a new flow the switch first performs a lookup against the flow tables. If a match is found, then the action dictated by the flow table is applied. If no match is found, then the switch generates a `PacketIn` OpenFlow message, and sends to the controller. The controller will then process the packet in (more specifically, the applications installed on the controller), and will return a response instructing the switch how to proceed as shown in Figure 2.4a. Typically, this will consist of a `FlowMod` OpenFlow message which instructs the switch to add a new entry into the flow table, modify an existing entry or delete an existing entry, as shown in Figure 2.4b. This is accompanied by a `PacketOut` message which instructs the switch to forward the buffered packet. The switch table rules will usually be set with a short time-to-live value so that it frees space in the flow table later (the default is typically 5 seconds without a packet matching the flow rule). Note that multiple OpenFlow messages can be included in a single network packet.

Reactive control has the benefit that it provides finer grained control of the network "on-the-fly", providing the ability to handle new flows as they arrive, without having to pre-empt future flows. However, reactive flow control is expensive both in terms of controller side computation in processing switch requests, and in network communication overheads between the switch and controller. The switch has to wrap the packet in a `PacketIn` message, and send to the controller and then wait for a reply, potentially buffering new packets in the flow while waiting. In scenarios where delays can cause issues, such as financial markets, this can be unacceptable.

2.6.2 Proactive

In the proactive approach, the flow tables of the switch are populated by the controller in advance, meaning that the majority of new flows can be handled by the switch without querying the controller. This is largely done through the use



(a) General reactive control

(b) Typical operation

Figure 2.4: Reactive switch control

of more generalised flow table entries, for example matching on a longer prefix. This is closer to the routing table model used in traditional networks.

This has the benefit that all packets are forwarded at line rate as the switch no longer needs to query the controller with all new flows. The major downside is that in a purely proactive model some of the finer grained control gained through the use of SDN is lost — in order to modify the network state the controller needs to first fetch flow statistics from the switch which introduces an extra latency.

2.6.3 Hybrid

Rather than utilise a purely proactive or reactive approach, the alternate is to use a combination of the two. For known flows that do not require fine-grained control, the flow tables are pro-actively filled allowing for high throughput on the switch. The remainder of the flows, which may require fine-grained control, are processed reactively and sent to the controller. For example, flows relating to known devices on an enterprise network (such as web servers, email servers etc) can be handled by pre-installed rules in the flow tables of network switches, but flows relating to a previously unknown device can be handled reactively by the controller (including applying any relevant security policies).

2.7 SDN Controllers

The SDN controller is one of the most critical aspects in SDN. There are many SDN controllers available, both open source and commercial [179]. SDN controllers are written in common programming languages, primarily Java, Python and C, and are designed to run on general purpose operating systems (Windows, Linux, MacOS). Table 2.2 provides a list of commonly used SDN controllers.

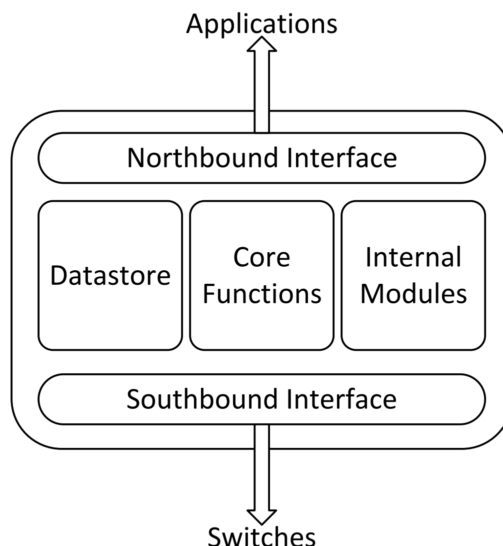


Figure 2.5: SDN Controller Components

Figure 2.5 demonstrates the primary components of a typical SDN controller. In order to communicate with forwarding devices, a southbound interface is implemented which maintains a connection with devices and handles the sending and receiving of, usually OpenFlow, messages. Similarly, a northbound interface is implemented, typically a RESTful interface, to communicate with external applications. The controller then contains a set of core functions, which are responsible the core functionality of the controller. This can include device (switch) management, maintaining the network topology and basic routing. Importantly, this will include a component for forwarding switch requests to internal modules or external applications, and sending the responses from these modules back to

Table 2.2: A Selection of Commonly Used SDN Controllers

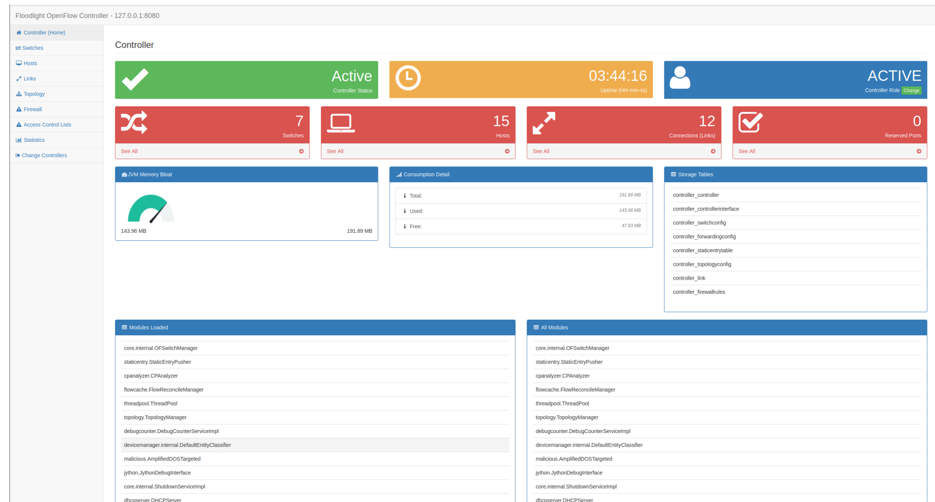
Controller	Language	Open Source	Developer/ Maintainer	Distributed
OpenDaylight [152]	Java	Y	Linux Foundation	N
Floodlight [187]	Java	Y	Big Switch Networks [1]	N
NOX [90]	Python/C++	Y	Nicira Networks	N
POX [186]	Python	Y	Nicira Networks	N
Ryu [197]	Python	Y	Independent	N
ONOS [20]	Java	Y	Linux Foundation, Open Networking Foundation	Y
Beacon [78]	Java	Y	Stanford University	N
Onix [115]	C, Python	N	Nicira Networks, Google, NEC, ICSI	Y
OVS-controller [176]	C	Y	Independent	N

[1] Big Switch Networks no longer involved, though developers still contribute

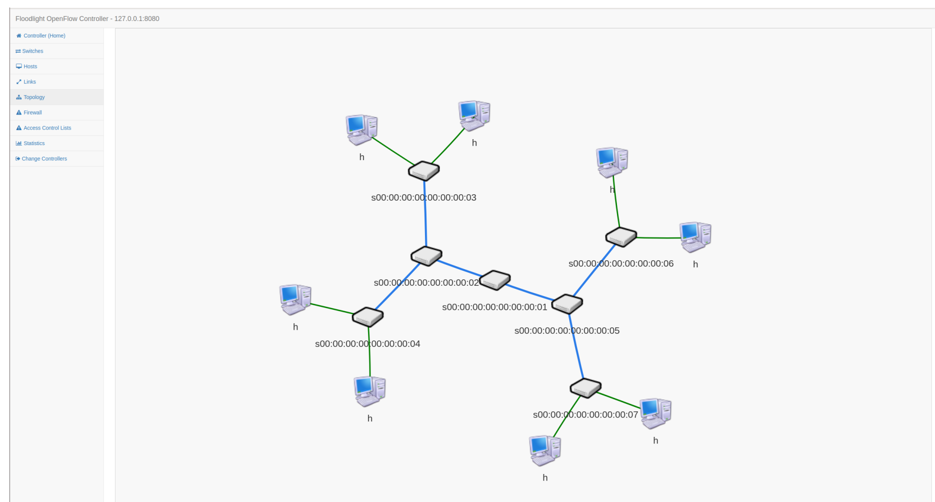
the switches. This functionality will rely on a datastore used to store network state information, including the currently observed topology, to be used by applications when making decisions, such as routing. This may be limited to a local datastore in a fully centralised deployment, or may represent a distributed, yet logically centralised, datastore is a distributed controller architecture is used.

A controller will also typically contain a set of internal modules which operate as applications to perform network functions. These can include functionality such as forwarding/routing, firewalls and load balancers. A controller will often be distributed with a set of these modules, but may also allow for the development of further modules by users.

Some SDN controllers provide an interface for management of the controller itself, commonly through the use of a web-based interface over HTTP. An example of this for the Floodlight controller can be seen in Figure 2.6



(a) Home Page



(b) Network Topology View

Figure 2.6: Floodlight web administrative interface

2.7.1 Controller Placement

An important consideration within SDN networks, in particular those with multiple controllers, is the placement of controller in relation to switches [94, 61, 105]. In particular, a major impact on the performance of an SDN controller is the latency of the switch-controller channel [183]. The two main factors that impact how quickly a controller can process a request are the computational time on the

controller in processing the request, and how long does it take for the request, and associated reply, to travel between the two entities. Therefore, there is a need to ensure that the switch-controller latency is reduced where possible. As well as reducing latency, the placement of controllers can have an impact on the security and robustness of the SDN control plane [163, 239, 93]. For example, a controller located at a large distance from the switch over a high latency communication channel will result in larger flow setup times.

2.8 Dependability and Faults

In this section I introduce the concept of dependability in computer systems. I then go on to discuss different types of faults that can occur within computer systems, and the failures that they can cause.

2.8.1 Dependability

Dependability is the ability of a system to, at any given time, provide the service for which it is designed [132]. Dependability generally refers to a set of attributes (as discussed in Section 2.8.1.1 below) which can be used to assess how dependable a system is.

Dependability and security are closely related [107, 131]. The dependability of a system is can be directly linked to how secure a system is. For example, if a system is vulnerable to denial-of-service type attacks which can take the system offline, and such an attack is carried out, then there is a direct impact on the availability of the system.

The Laprie dependability tree, as seen in Figure 2.7, demonstrates the attributes of dependability, as well as branches representing the *means* of achieving those attributes and the *impairments* to dependability.

2.8.1.1 Attributes

The attributes of dependability are the properties which can be measured in order to assess the level of dependability within a system. According to Avizienis et

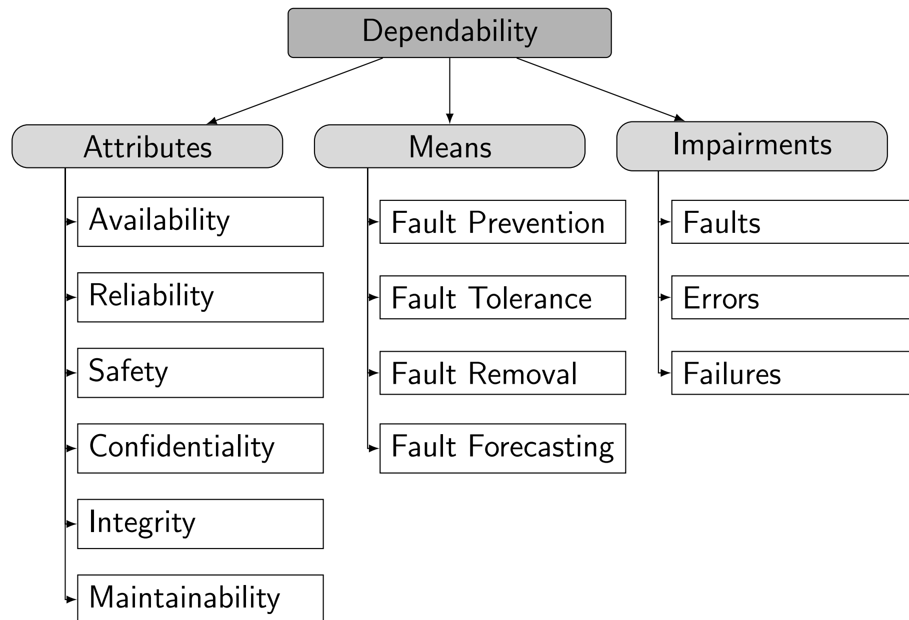


Figure 2.7: Laprie Dependability Tree [132]

al., dependability covers the following attributes [15]:

Availability The system is operating, or ready to operate, correctly. Usually given as a percentage of uptime. For example, typically a server is highly available if it meets the *five nines* threshold of 99.999% uptime, representing less than 5.26 minutes of downtime per year. Note that availability indicates uptime only (liveness), not correctness.

Reliability The ability of the system to provide the continuity of correct service. Whereas availability representing the total uptime of a system, reliability measures the frequency of failures. Often represented as the mean time between failures (MTBF), where

$$MTBF = \frac{\text{Total Uptime} - \text{Total Downtime}}{\text{Number of Failures}}$$

Integrity The system, and in particular data, is unaltered, either by a fault or outside (attacker) influence.

Safety The system is operating without harmful effects on the user or environment.

Maintainability Represents how easy the system is to repair.

With the addition of the **Confidentiality** attribute, which covers the disclosure of confidential information, **Security** can also be considered an attribute of dependability being a summation of the CIA triad of confidentiality, integrity and availability [107, 131, 202]. In some particular scenarios, safety may also be a factor of security, such as in the example of cyber-physical systems as discussed in Chapter 4, Section 4.5.

2.8.1.2 Means

The means represent a set of methods and techniques that can be utilised in order to aid in the development of a dependable system and to increase dependability.

Fault Prevention Measures are taken to prevent the introduction of faults at the implementation phase, for example through secure development methodologies

Fault Tolerance Measures are taken such that the system is able to operate correctly and provide the required service, even with the presence of faults.

Fault Removal Discovered faults are removed. Can be applied during development through testing and other verification methods, or during software use through maintenance (patching, updates etc).

Fault Forecasting The occurrence and impact of future faults is predicted.

In this work I focus on the concept of fault tolerance.

2.8.1.3 Impairments (Threats)

Impairments (often called threats) are the things that cause an impact to the dependability of a system. The primary thing that can cause a loss of dependability are faults, which can in turn cause errors and failures. I discuss these further in Section 2.8.2 below.

Fault A defect (bug) in a system. Whilst a fault may exist in a system, it has to be triggered during program execution in order to cause an error (and failure).

Error An error represents a deviation in the behaviour of a system from its intended behaviour. An error can be caused by a fault, and can cause a failure. Errors are generally not observable outside of the system unless they cause a failure, but can be detected by monitoring the running process (such as through the use of debuggers).

Failure A failure is when the delivered service deviates from the system specification. A failure is observable to a user (human or service) of the system.

2.8.2 Faults

It is first important to distinguish between the concepts of faults, errors and failures. A fault is a hardware defect or bug (programming mistake) within a system. An error is the result of a fault that occurs within the operation of a system [116, Ch. 1, p. 1-2]. Whilst a fault will always be present within the system, it will only cause an error when the execution of the program triggers a fault. A failure is the state in which the system deviates from its specification [132]. Whilst a failure is caused by an error, not all errors will result in failure. For example, a error which causes an exception to be thrown can be caught and handled to prevent a failure occurring As per the example provided by Goren and Krishna [116, Ch. 1, p. 2], a programming fault that uses an absolute version of a function (that always returns a positive value) could cause an error if passed a negative value, where the result of the function should be a negative value. An error caused by a fault may also cause further errors within a system if an erroneous result of the error is passed into other components within the system, resulting in multiple errors caused by a single fault.

In terms of security, a fault can be exploited by an attacker in order to cause an error or failure. For example, in the case of a buffer overflow attack, in which a programming error allows a user input to overflow a buffer, the attacker can cause a controlled error to cause the program execution to deviate from it's normal operation and execute their own code.

2.8.2.1 Types of Failure

A failure can manifest in a number of ways. According to Cristian, the classification of failures in a distributed system are [58]:

Omission Error The server fails to respond to (one or more) incoming requests.

Crash Failure The server halts and fails to respond to all requests.

Timing Failure The server responds to a request either too early, or too late.

Response Failure The server responds incorrectly, either by returning an incorrect output, or the state transition that occurs is incorrect.

Arbitrary Failure The server may produce an arbitrary response at arbitrary times. This also encompasses all other types of failure.

These types of failures can have direct impact on the dependability attributes discussed in Section 2.8.1. For example, omission and crash failures (also commonly known as fail-stop failures) which result in the system going offline or not responding to requests has a direct impact on the availability and reliability of the system. Similarly, response or arbitrary failures can result in a loss of integrity in the system.

The failures can also be caused with malicious intent by an attacker. By causing an omission, crash or timing failure, the attacker effectively performs a denial-of-service attack (whilst timing attacks are not obviously denial of service, if performance is degraded enough then a user will stop using the service, or alternatively in real-time systems additional delay can cause system failure). If the server were to become compromised, then the attacker can cause response failures to respond to requests as they require, or cause the server to send arbitrary responses. These types of attacker-induced failures are the focus of this work.

2.9 SDN Fault Tolerance

2.9.1 Native OpenFlow Support

As of version 1.2 of the OpenFlow specification [172], OpenFlow supports the assigning of one of three roles to controllers relating to how they operate with each switch:

Master A `master` controller has full read-write access to the switch. It can process asynchronous messages from the controller, such as packet-in messages, and can send commands, such as flow-mod, to the switch. Where multiple controllers exist, only one controller can be the master for each switch, with all other controllers being assigned `slave` roles.

Slave A controller assigned the `slave` role has read-only access to the switch, and can only receive port-status messages.

Equal The default role, permitting full read-write access to the switch. Unlike the `master` role, there can be multiple controllers with the `equal` role, each with the same level of access. The switch performs no arbitration between controllers.

In Version 1.1 of the OpenFlow protocol [171], controllers could not be assigned roles. If more than one controller was used within a network, a switch could only connect to a single one.

Utilising a group of controllers provides tolerance to fail-stop failures of controllers. If a single `master`, multiple `slave` setup is used, if the master controller fails, one of the slave controllers will become the next master. If a group of `equal` controllers is used, the switch communicates with all controllers, meaning that if one fails it will simply continue to operate under the remaining controllers. Controllers operating in `equal` mode need to be synchronised to ensure a consistent view of the network state. As of Version 1.5.1 [175] the OpenFlow specification does not dictate how the switch should partition control amongst different `equal` controllers. The protocol does not specify any consensus or agreement protocols

amongst controllers in the equal state, meaning that the protocol itself provides no protection against rogue controllers.

2.9.2 ONOS

ONOS (Open Network Operating System) [20] is an SDN controller maintained by the Open Networking Foundation [170]. ONOS is open source and built using Java, and is one of the most full-featured controllers available. The controller natively supports distribution, allowing controllers to run in a cluster. The motivation behind this is both for fault-tolerance in the case of controller failure, and also for scalability.

Distribution within ONOS operates using a single controller, single switch model. For any given switch, there is one primary `master` controller which handles all control for a particular switch. The primary controller is chosen by the pool of controllers using a leadership election. If the primary controller fails, the switch connects to one of the backup controllers.

ONOS is logically centralised, through the use of a distributed datastores, with network state information disseminated as events.

2.10 Consensus

In consensus, or agreement, protocols, a set of processes will agree on a common value from a set of proposed values [38]. In the broadest sense, two events occur. Each process will *propose* a value, and all processes will then *decide* on a common value. Any correct process will decide upon the same value. A consensus protocol must meet a set of properties covering termination, validity, integrity and agreement (which vary according to the particular protocol). Some consensus protocol designs have built in fault tolerance.

Paxos [126] is one of the earliest and most widely known consensus protocols, and comes in many variants. In basic Paxos, processes can be proposers, learners, acceptors and leaders. At a high level, a proposer (the leader) sends a prepare proposal to the acceptors. Acceptors can return a promise to accept no further

proposals to the proposer. If a proposer receives promises from a quorum of acceptors, it will send an accept request, containing its final proposal. An acceptor will then accept the proposal (and its value), and send an accepted message to the proposer and every learner. The key point in Paxos is that all proposals are attached to a sequence number, which determines the order in which proposals are accepted. For a more detailed explanation of the basic protocol I refer to Lamport [127].

Moise [160] proposes a variant of Paxos that employs the optimisations described in FastPaxos [28] and Fast Paxos [128] (note the addition of a space). FastPaxos [28] provides an optimisation over Paxos in which the leader remains stable. This is achieved by removing the initial prepare phase. Fast Paxos, proposed by Lamport [128], uses the case where all proposers propose the same initial value. The leader send a special signal to acceptors to accept the next proposal they receive. Moise [160] makes use of both optimisations. The FastPaxos optimisation is run in every instance, while the Fast Paxos optimisation is only used when suitable (the decision needs to be taken at runtime by the leader).

The previously described Paxos variants only provide tolerance to fail-stop failures (crashes), requiring $2f + 1$ processes, where f is the expected number of faults. This is an improvement of earlier works that only tolerate non-byzantine faults requiring $3f + 1$ processes [181]. Lamport et al. [129] first discuss the problems of consensus with Byzantine failures. Subsequent works have expanded on Paxos to provide Byzantine fault tolerant properties. Martin et al. [145] propose Fast Byzantine (FaB) Paxos. FaB Paxos requires $5f + 1$ processes to provide fault tolerance.

The k-set consensus protocols, as introduced by Chaudhuri [47], are an extension of the consensus protocol in which each processor decides on a single value, such that the set of all decided values across all processes is of size at most k. The number of choices is directly related to the number of allowable faults. The protocol is designed for use in a totally asynchronous system. The author shows that, due to the uncertainty condition that the set of values that could be chosen is not pre determined, the final set determined over the run, that the limit is a (k-1)-resilient protocol. The total number of processors must be greater than 2(k-1). In the protocol, each processor maintains a vector of size n, where the ith

entry corresponds to the initial value of processor i if it knows it, ϕ otherwise (so initially the vector held by each processor only contains its own initial value). In each round, each processor broadcasts its vector, and receives the vectors of all the other processors, which it will use to populate its own. As $k-1$ processors may be faulty, each processor only waits for $n-k-1$ vectors in each round. Once a node receives $n-k-1$ vectors that are identical to its own, it decides a value based on that vector. It then broadcasts that vector, labelled as a decider vector.

De Prisco et al. [62] extend k-set consensus protocols to the case where Byzantine failures may be present. In particular, they evaluate with 6 different validity conditions. Three different protocols are proposed, that provide Byzantine tolerance in the message passing model, with differing validity conditions.

Dolev et al. [68] provide an agreement protocol that can reach agreement with Byzantine faults without using authentication using $3f+1$ processes, in $2f+3$ rounds. The protocol assumes that the set of possible values is $\{0,1\}$ with 0 used as a default for faulty processes, and the setting is synchronous in that a process knows when a round starts and ends. In the protocol, processes transmit their value, and other processes indicate their support for processes for which it receives values. In any round, if a process confirms that enough processes have committed to 1, it also does. After $2f+3$ rounds, if 1 is committed then it agrees on 1, else 0. Bracha and Toueg [34] present an asynchronous protocol that can tolerate fail-stop failures with $\lceil (f+1)/2 \rceil$ correct processes, and Byzantine faults with $\lceil (2n+2)/3 \rceil$ correct processes.

Hurfin and Raynal [102] propose a asynchronous consensus protocol that incorporates a weak failure detector built out of a finite state automaton, a voting mechanism and the assumption that processes may change their mind during a round. Unreliable failure detectors were first proposed by Chandra and Toueg [45], however this original work uses a centralised setting. All unreliable failure detector algorithms are based on a rotating coordinator paradigm, using asynchronous rounds. In each round, a pre-determined process (the coordinator) proposes a decision value, and then the protocol dictates how the processes cooperate. In [45], the coordinator receives each processes estimate of the decision value, computes a new estimate and distributes. If it receives an acknowledgement from a majority of processes, it takes its estimate as the overall output

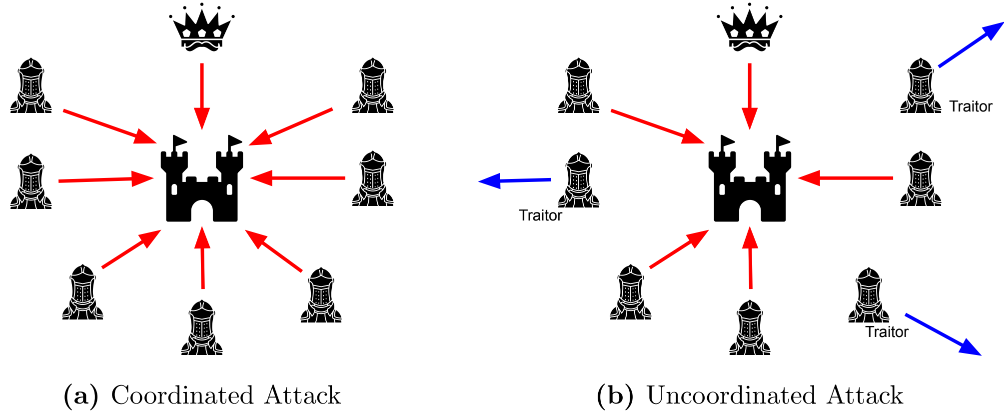
and informs all processes. Processes can send back a negative acknowledgement if they believe the coordinator to be faulty. Hurfin and Raynal follow a similar approach, where processes vote to move onto the next round (and a new coordinator). The voting behaviour of processes is determined by a simple finite state automaton. Consensus can be reached in 2 communication steps. However, as this protocol allows processes to change their values arbitrarily, the protocol is not Byzantine fault tolerant.

Fitzi and Garay [82] propose a consensus protocol that solves the δ-differential consensus problem. The δ-differential problem differs to strong consensus [166], where the decided upon value is proposed by a single process, no matter what is proposed by the other processes, by deciding upon a value that is proposed by the majority of the participants (assuming the input values are chosen from some fixed domain) using a shared-coin protocol. This protocol has limitations in the decentralised detection setting where inputs are probabilities/confidence values as there may be no majority value from the inputs as it is likely that every process provides a different input.

2.11 Byzantine Fault Tolerance

In Section 2.10 above I discuss a number of algorithms for providing consensus amongst a set of participants. Whilst these approaches can be effective in allowing a set of participants to agree on a value, even in the present of faulty or malicious participants, these protocols do not translate well to the case of secure SDN control, and similar scenarios, where a switch (client) needs to communicate with a set of controllers (servers). Whilst the consensus protocols would allow the controllers to agree on a single response to a switch request, they would not provide protection for returning the result safely to the client. This led to the development of byzantine fault tolerant algorithms.

In this section, I introduce the concept of byzantine faults and discuss a number of algorithms which provide byzantine fault tolerance.



(a) Coordinated Attack **(b)** Uncoordinated Attack

Figure 2.8: Byzantine Siege

2.11.1 Byzantine Faults

Generally, a byzantine fault can be described as a fault in which different symptoms are presented to different observers [75]. The concept of byzantine faults was first proposed by Lamport et al. in "The Byzantine Generals Problem" [129]. The original work provides an example in which multiple units of the Byzantine (as in the Byzantine empire) army are camped outside an enemy city, with each unit commanded by its own general. Generals are able to communicate via messenger. The goal of the generals is to agree on a plan of attack in order to siege the city, with the caveat that some of the generals are traitors and aim to prevent the generals from reaching agreement. In the basic problem, the plan can either be to attack, or retreat. If all generals agree to attack (as in Figure 2.8a), then the siege will be won. If any decide to retreat, then the siege will be lost (as in Figure 2.8b). The risks are that a message may be lost or modified by the messenger, or alternatively a malicious general may send messages aimed to mislead the other generals. The generals need an algorithm which both ensures that all *loyal* generals decide upon the same plan of action, and a small number of traitors cannot cause the loyal generals to adopt a bad plan.

The problem is formulated as every loyal general must obtain the same information $v(1), ..., v(n)$, where $v(i)$ represents the value (decision) of the ith general, which they communicate to each other. The first condition is that each loyal

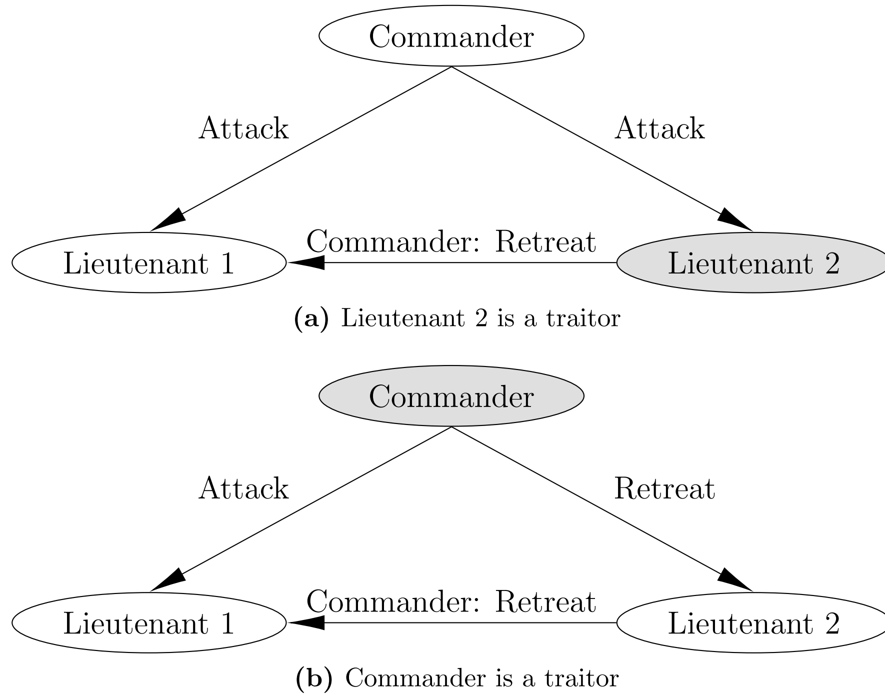
general must obtain the same set of values $v(1), ..., v(n)$. This could lead to a scenario where a traitorous general sends different values to different generals, which means that a general cannot use the value of $v(i)$ directly from the ith general. The second condition is that the value $v(i)$ sent by a loyal general, must be used by all other loyal generals.

Within the original work, the problem is restricted to a scenario in which a single general sends his value (order) to a set of lieutenants. This is the specified "Byzantine General Problem" in which a) all loyal lieutenants must obey the same order and b) if the commanding general is loyal, they all loyal lieutenants should obey his order. Both the commanding general, and the lieutenants, can be traitors.

Lamport et al. describe this as the three generals solution. If we consider the scenario in which the loyal commander tells both lieutenants to attack, however Lieutenant 2, who is a traitor, tells Lieutenant 1 that the commander told him to retreat (as seen in Figure 2.9a). As the loyal Lieutenant 1 must obey the command from the commander, then he must attack. Figure 2.9b shows an alternate scenario where the commander is a traitor, and gives each lieutenant a different order. It is impossible for Lieutenant 1 to know who is the traitor here, as they do not know what order the commander gave to Lieutenant 2. In this scenario, as both lieutenants are loyal they both follow the order that they received from the commander, and the siege fails.

This leads to the claim that in the scenario where there are three generals, and one is a traitor, there is no solution to the byzantine generals problem, meaning that to handle f traitors, a minimum of $3f + 1$ generals are required. A full proof of this is provided in [181]. This is why the byzantine-fault tolerant algorithms below generally require $3f + 1$ replicas in order to operate.

In distributed computing, a byzantine fault represents the scenario in which a component may fail, but can appear to be both failing and functioning to different servers. Typically, byzantine faults are considered in a client server model in which a client sends a request to a set of replicas (servers) who must agree on the ordering of requests, process the request, generate a response and return to the client, with the client taking the majority response. A commonly used example of this is a network file store where a client can make read and



(a) Lieutenant 2 is a traitor

(b) Commander is a traitor

Figure 2.9: Byzantine Generals Problem

write requests to the servers. In these algorithms the servers, and in some cases also the client, are potentially faulty.

2.11.2 Byzantine Fault Tolerant Algorithms

A typical byzantine fault tolerant (BFT) algorithm is designed to operate with a set of replicas (servers) providing a service (a commonly used example is a network file store), along with a client that will make requests to the replicas, which will process the request and generate a response, and expect some reply. A BFT algorithm will typically consist of two phases — agreement and execution. The agreement stage is typically used to ensure all replicas are in a consistent state, and most often relates to message ordering. This step usually requires multiple rounds of communication requiring multiple multicast message, and so represents a communication overhead, along with an encryption overhead if digital signatures are used. The execution stage is where the replicas process a request and return the result to the clients. This stage generally represents a computational

overhead, though this can vary with the exact application in use.

Lamport et al., when initially proposing the notion of byzantine faults as the "Byzantine Generals Problem" (see Section 2.11.1 above), propose two solutions to providing byzantine fault tolerance [129]. This covers two scenarios — one in which the generals communicate orally, and one in which signed messages are used.

In the oral messages approach, it is assumed that every message is delivered correctly, the receiver of a message knows who sent it (meaning a traitor cannot inject messages into the system), and the absence of a message can be detected. The algorithm $OM(f)$ aims to find a majority value amongst the generals. The algorithm has two parts. In the case where $f = 0$, $OM(0)$:

1. The commander sends his value to every lieutenant.
2. Each lieutenant uses the value received from the commander, or assumes an order of *retreat* if no value is received.

For values $f > 0$, the algorithm $OM(f)$:

1. The commander sends his value to every lieutenant
2. For each i, v_i is the value received by Lieutenant i, or *retreat* if no value is received. Lieutenant i then becomes the commander in $OM(f-1)$, sending v_i to the other lieutenants.
3. For each i, j where $i \neq j$, v_j is the value Lieutenant i received from Lieutenant j in step (2), or *retreat* if no value was received. Lieutenant i uses the value $majority(v_i, ..., v_{n-1})$, where $majority()$ is either:
   (a) The majority value in the set or received values
   (b) The median value of the set of received values, assuming the possible values come from an ordered set.

The problem with this approach is that participants can lie. This is solved in the approach in which signed messages are used, with the assumption that a loyal general's signature cannot be forged, and anyone can verify the signature of any general. The algorithm a function $choice(V)$ that when applied to the set

of orders V, will obtain a single order. If V contains a single element, then that is the accepted order, otherwise if the set it empty, then *retreat* is the accepted order. The notation $v : i$ is used to indicate order v has been signed by general i, which can be chained to show multiple signatures (e.g. $v : i : j$ indicates message $v : i$ has also been signed by general j). The algorithm $SM(f)$ is as follows, assuming initially $V_i = \emptyset$:

1. Commander signs and send his value to every lieutenant
2. For each i:
   (a) If Lieutenant i receives message $v : 0$ from the commander, and has received no other values, then:
      i. $V_i = v$
      ii. Send $v : 0 : i$ to every other lieutenant
   (b) If Lieutenant i receives a message $v : 0 : j_1 : ... : j_k$ and v is not in V_i then:
      i. add v to V_i
      ii. If $k < m$, send message $v : 0 : j_1 : ... : j_k, i$ to every lieutenant other than $j_1 : ... : j_k$
3. If Lieutenant i receives no more messages, he accepts the order $choice(V_i)$

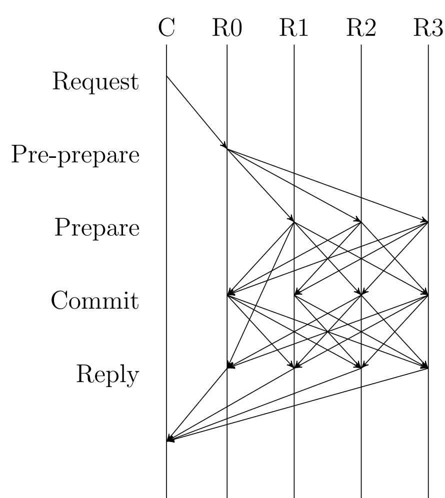
A proof of the correctness of both the $OM(m)$ and $SM(m)$ algorithms is provided in the original paper [129]. Since this initial approach there has been a number of attempts to improve on these algorithms. The best known algorithm, and the basis for many future works, for providing byzantine fault tolerance is the "Practical Byzantine Fault Tolerance" (PBFT) algorithm as proposed by Castro and Liskov [41]. PBFT, as is the case with most BFT protocols, is a form of state machine replication [125, 203], in which the service is represented by a state machine that is replicated across a series of distributed nodes (replicas). In order to handle f faults, $3f + 1$ replicas are required. It is assumed that replicas are deterministic, meaning that any correct replica should return the same result to the same request. More replicas may be used, however as the authors note the greater the number of replicas the greater the impact on performance. On each run of the protocol, the replicas adopt a succession of *views*. In each view, one of

the replicas is designated as the *primary*, with the primary changing with each new view. A failure of the primary results in a change to the next view (this is similar to the approach as used by Paxos [126]). In the basic protocol:

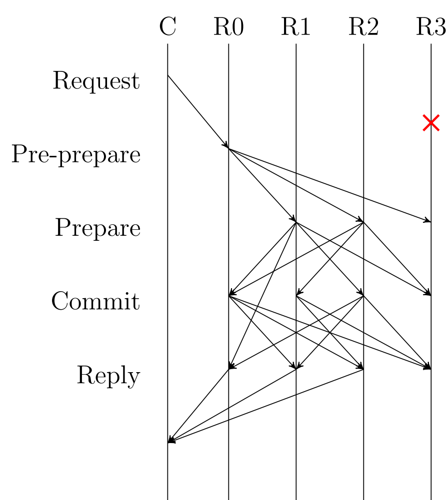
1. The client send a request to the primary
2. The primary multicasts the request to the other replicas
3. Replicas prepare a response, and send a reply directly to the client
4. The client waits for $f + 1$ responses from the replicas with the same result, which is then taken as the correct response.

The multicast step (step 2) is broken down into three phases, meaning the protocol is broken down to the *request,pre-prepare*, *prepare*, *commit* and *reply* phases, as shown in Figure 2.10a. The three-phase multicast ensures that requests are totally ordered. Messages between replicas are signed using public-key signatures to ensure message integrity and authenticity. In the *pre-prepare* phase, the primary assigns a sequence number to the request and forwards to all of the replicas. On accepting a *pre-prepare* message (based on a set of conditions), the replicas each broadcast a *prepare* message to all replicas. Once a replica has received $2f$ *prepare* messages form other replicas, it multicasts a *commit* message to the other replicas (which contains a digest of the original client request). Once a replica has received $2f + 1$ commits from other replicas, then the replica will process the client request and send the reply directly to the client (the *reply* phase). In the case of a single node failing, as shown in Figure 2.10b, the protocol can complete despite replica 3 (R3) becoming faulty and not participating in the protocol.

The protocol is evaluated using an implementation of a network file store (NFS) and compared with a standard NFS implementation. In testing, they show that the BFT-based version only generates a 3% overhead over the standard implementation, however this also includes the NFS operations. When performing micro-benchmarks with null operations on the datastore, the overhead of using replication can be as high as 309%. The overhead largely comes from the cryptographic operation of signing messages, as well as the cost of the additional communication steps. The authors propose some improvements to improve efficiency, such as sending a tentative response to the client after the *prepare* phase,



(a) Normal Operation



(b) Faulty Replica 3

Figure 2.10: PBFT Algorithm [41]

on the assumption that if the client receives $2f + 1$ matching replicas at this stage then it can be assumed the replicas will move to the *commit* stage. Message volume can also be reduced by only a single replica (specified by the client) sending a full response, with the others returning a digest of their response which can then be compared. The PBFT protocol was later expanded to incorporate a recovery protocol in order to automatically bring faulty replicas back online and

up to state [42].

Since PBFT, there have been many approaches which aim to improve upon the performance of the PBFT approach, by either improving the efficiency of the agreement stage of the protocol, or by decoupling the agreement and execution phases.

Yin et al. propose an approach in which the agreement and execution functionality of replicas are separated in order to reduce replication costs [242]. Compared to the traditional BFT approach in which replicas perform ordering, and then execute requests, requiring $3f + 1$ replicas, an ordering cluster of $3f + 1$ replicas order requests, which are then sent to an execution cluster where replicas process the request and return a reply to the clients. The execution cluster only needs to return a simple majority of $f + 1$ matching replies, and so only $2f + 1$ replicas are required to execute the request. When testing with an implementation based on the BASE library (an implementation of PBFT which allows for non-deterministic servers) [43], the protocol is shown to have higher latency than the reference BASE protocol, however is shown to have a lower per-request processing cost when compared to BASE, allowing for higher throughput. Kotla and Dahlin build upon this approach of agreement-execution separation by introducing a paralleliser layer between the agreement and execution laters, which permits the concurrent execution of requests (in part by relaxing the ordering requirement for requests which do not read or write to the same variables) [117]. The limitation of this approach is that the paralleliser requires knowledge of the system processing requests in order to identify non-overlapping requests. When compared to BASE, the proposed approach, CBASE, is able to handle 4700 requests/second when using 128 threads, compared to the 50 operations/second handled by BASE (which is limited to a single thread).

Zyzzyva is a state-machine replication BFT protocol that aims to improve performance by removing the three-phase ordering protocol of other algorithms, with replicas instead speculating on the correct order to process messages [118]. In the protocol, the primary attaches a sequence number to a request, and forwards to replicas, who then process the request and send a reply and a representation of their state history to the client. If the client receives $3f + 1$ matching replies and histories, they accept the reply. If they receive $2f+1 < n < 3f+1$ matching

replies, they send a *commit* message to the replicas to ensure the replicas update their states correctly. If fewer than $2f + 1$ responses match, then the client sends the request to all replicas, which request a new sequence number from the primary. If the client subsequently identifies inconsistent sequencing from the primary, it triggers the next view state and a new primary. When compared to the PBFT, Q/U and HQ protocols, the protocol is shown to achieve 2.7, 3 and 9 times greater throughput respectively, as well as significantly lower latency.

There are BFT approaches that combine BFT protocols with the concept of quorums. Assuming a universe of servers U ($|U| = n$), the quorum system is the non-empty set of subsets of U, where each pair of subsets intersect. Each of these subsets is a quorum. When performing a read or write operation, a client contacts a quorum of servers. There have been multiple approaches utilising quorums for protecting against non-byzantine faults [4, 49, 1, 86, 96, 164]. Malkhi and Reiter propose "Byzantine Quorum Systems", in which the overlap of the quorums provides protection against byzantine server failure, as quorums overlap by at least $2f + 1$ servers [143]. If a value is written to quorum $Q1$, and subsequently read from quorum $Q2$, then the correct value can be read from the intersection $Q1 \cap Q2$, on the assumption that if quorums overlap with $2f + 1$ replicas, then if f servers are faults $f + 1$ should return the correct result.

Abd-El-Malek et al. attempt to address the limitation in many BFT approaches in that as the number of tolerated faults increases (an increase in f), the performance of the system rapidly degrades [2]. The protocol, Query/Update (Q/U), is a quorum-based protocol which assumes a failure model that can handle both byzantine and crash faults [226, 5], with the performance degrading more gradually as f increases. In the protocol, a *query* represents a read operation, and an *update* represents a write operation. $5f + 1$ replicas are required, and clients maintain a cache of replica histories. Under normal operation, the protocol operates as a single phase – the client sends a request to the replicas, receives responses (along with replica histories) and if a quorum of $4f + 1$ replicas agree, it accepts the result. If $2f + 1 < n < 4f + 1$ matching responses are agreed, then a conflict resolution mechanism is triggered in which the client brings the replicas up to a consistent state and resubmits the request. When compared to

the BASE PBFT implementation, increasing f from 1 to 5 results in just a 36% drop in throughput with Q/U, compared to 83% for BFT.

Cowling et al. build on the idea of Q/U with the Hybrid Quorum (HQ) protocol [57]. This quorum-based protocol aims to improve on the limiting factor of Q/U in requiring $5f+1$ replicas, instead requiring a quorum of $2f+1$ replicas in normal operation, and $3f+1$ when under contention. Like Q/U, HQ is designed to scale well with an increase in f. The system assumes a read/write model for client operations. In normal operation, a 2-phase write protocol is used. In the first phase, a client contacts a quorum of replicas requesting a certificate indicating that they can make a write operation at a specified timestamp. If a set of matching certificates is returned, the client moves onto phase 2, in which they send their request, along with the certificate, to the quorum of replicas to perform the write. If there is contention, i.e. multiple clients attempting to write at the same timestamp, then the protocol falls back to the PBFT protocol to order the multiple requests.

Typically, as can be clearly seen in the previously mentioned works, the described protocols require a minimum of $3f+1$ replicas in order to handle f faults. Whilst this does provide fault tolerance, it represents a large cost in replication (with a minimum of 4 replicas required to handle a singe fault). There has been some work which has attempted to reduce this minimum to $2f+1$ replicas. An example of this is the MinBFT and MinZyzzyva protocols as described by Veronese et al., which are adaptions of the PBFT and Zyzzyva protocols respectively [231]. Part of this reduction is achieved through the use of tamperproof components, a Trusted Timely Computing Base (TTCB) and a Attested Append-Only Memory, which have both previously been used to provide $2f+1$ byzantine fault tolerance [50, 55]. The tamperproof component acts as an oracle, which allows the reduction in the number of required replicas [55]. The tamperproof module is a unique service identifier generator (USIG), which is present on each server and facilitates the secure generation of signed sequence numbers. The MinBFT algorithm is functionally very similar to PBFT, except the *prepare* phase has been removed, and the client sends the initial request to all replicas, not just the primary (though the primary is still responsible for assigning a sequence number and forwarding to the replicas). Similarly, MinZyzzyva follows the Zyzzyva protocol,

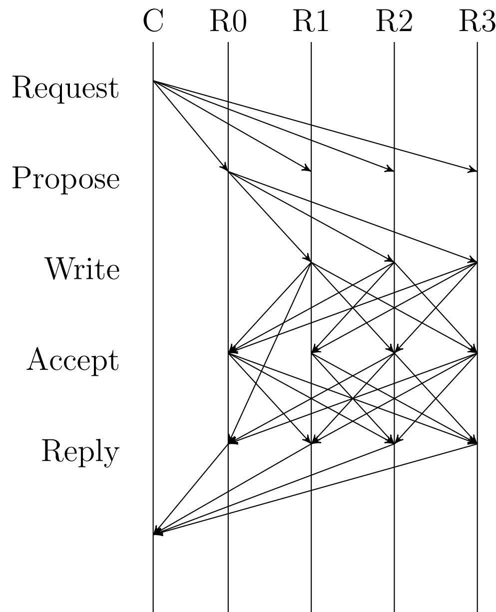


Figure 2.11: BFT-SMaRt protocol, normal operation

utilising the same stages, however again the client sends the initial request to all replicas, and not just the primary.

The BFT-SMaRt protocol, as proposed by Bessani et al. [22], is a java-based BFT implementation that has been used to provide a byzantine fault tolerant SDN controller. BFT-SMaRt builds upon the authors previous work in BFT state machine replication [220, 23, 37], with the BFT protocol specifically modelled on the MoD-SMaRt protocol found in [220]. BFT-SMaRt is very similar to the PBFT protocol, with the core protocol being a 3-phase process consisting of *propose*, *write* and *accept*, which can be seen in Figure 2.11. As with PBFT, and most BFT protocols, BFT-SMaRt requires $3f + 1$ replicas to handle f faults. When consensus is not reached, or there is inconsistency, a state transfer protocol is used in order recover replicas to a synchronous state [23]. As BFT-SMaRt is modular, it is able to support a greater number of clients than PBFT and achieve a higher throughput — PBFT reaches it's maximum throughput of 78765 requests/second with 100 client compared to 83801 with 1000 clients for BFT-SMaRt, with the single-threadiness of PBFT being the limiting factor (increasing past 100 clients causes a degradation in performance).

2.12 Conclusion

In this chapter I have provided a background on the key concepts of networking, programmable networks and SDN, with a focus on the OpenFlow protocol. I have also given an introduction to dependability and faults, consensus protocols and byzantine fault tolerance.

In the next chapter I provide a more focussed look at the existing literature on SDN security, with a particular focus on works which aim to add security to the SDN control plane to prevent controller (or application) compromise. I also examine existing work which attempts to provide byzantine fault tolerance in SDN control.

Chapter 3

Literature Review and Related Work

3.1 Introduction

In this chapter I provide an overview of the literature into the security of Software Defined Networks (SDN). I begin with an overview of papers discussing the general security of SDN, and then focus on security of the SDN controller. I first discuss works which aim to detect compromised SDN controllers, and then cover works which introduce security controls into SDN controllers in order to prevent compromise. I then examine fault-tolerant SDN control architectures, including those that follow a primary-backup model and those that apply byzantine fault-tolerant protocols.

3.2 Security of SDN

The security issues surrounding SDN were made apparent in the early days of SDN [207, 210]. Early work that proceeded SDN although explores the concept of separated control and data planes, Sane [40] and Ethane [39], already highlighted potential security issues through the use of centralised control. There have been

numerous works focussing on the security of SDN, covering all layers of the SDN architecture [206].

Kloti et al. apply the STRIDE security analysis framework ([97]) and attack trees ([198]) to the OpenFlow protocol [114]. When using STRIDE, a data flow diagram of the process is constructed and used to identify vulnerabilities (and their impacts of the system) of the types Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Through analysing the OpenFlow protocol they are able to identify information disclosure, denial of service and tempering vulnerabilities. For example, an attacker connected to an SDN switch through several other clients can infer which clients have communicated with a server connected to a second switch by analysing the flow setup times (a client that has communicated with the server will have a lower flow setup time as a flow rule will already exist in the switches flow table), leading to information disclosure.

Attacks on SDN has a large amount of overlap with the types of attacks that can occur within conventional layer 2 and 3 networks, as shown by Abdou et al. [3]. Many attacks that can be applied to a traditional network can be emulated within SDN, though with a different approach, for example ARP poisoning in a traditional network can be replaced by host profile poisoning [98]. Abdou et al. argue that defending an SDN from such attacks is more difficult, for example as the SDN control plane cannot be protect by edge based filtering which is a standard approach in conventional networks.

There are works which explore the security of specific use cases for software-defined networks. For example, Costa and Costa [56] demonstrate issues with FlowVisor [215], which provides network isolation using SDN, when a malicious controller is in use. The malicious controller can break isolation between network slices, for example by installing flow rules which can modify the VLAN IDs of flows. I demonstrate attacks that can be launched from a malicious SDN controller in Chapter 4, and in an associated paper [87].

Whilst in this work I focus on the impact of compromised SDN controllers, there has also been work on examining the impact of compromised SDN switches [12, 211].

3.3 Detecting Compromised SDN Controllers

One approach to limiting the impact of compromised SDN controllers is to detect when malicious flow rules are installed into a network. Detected malicious flow rules can then be removed, and the compromised controller repaired. The following works discuss approaches which look to identifying problematic, though not necessarily malicious, flow rules in SDN networks.

Veriflow is an example of such a system [111, 112]. Veriflow is a verification mechanism for the data plane which aims to identify network-wide invariants, such as loops, black holes and access control violations, utilising a checker which sits on the CDPI and intercepts flow rules as the are travelling form the controller to the switch. As the system sits on the CDPI and processes commands before they reach the switch, the system operates in near real-time in order to minimise additional latency. Veriflow works in three steps. First, equivalence classes are generated, with each equivalence class representing a set of packets which are forwarded in the same way. Secondly, a set of forwarding graphs are generated for the equivalence classes, modelling the forwarding behaviour of the network. Finally, these graphs are traversed in order to identify invariants and identify problematic flow rules.

Similarly, FlowChecker is a system which makes use of Binary Decision Diagrams (BDDs) to identify conflicting rules within a single switches flow table [7]. FlowChecker is an extension of previous work, ConfigChecker, which uses a similar approach for verifying network configuration [8]. Compared to Veriflow, which sits on the CDPI, FlowChecker behaves as a service which the controllers (and applications) use to verify flow rules. This in itself a limitation of the work in the scenario where controllers may be malicious and not perform verification.

3.3.0.1 Blockchain

There have been a number of blockchain-based approaches proposed to provide protection against compromised SDN controllers [33, 140, 65].

Boukria et al. propose the use of a trusted third party node within the network, along with the use of blockchain, to prevent against “False Flow Rule

Injection (FFJI)" attacks, in which an attacker performs a person-in-the-middle attack against the switch-controller channel to inject false flow rules onto the switch [33]. The controller, on sending a flow rule to the switch, stores a hash of the flow rule as a block on the blockchain, of which the controller and third party are members. On receiving a flow rule from the controller, the data plane (switch) forward the received rule onto the third party, which compares the rule received by the switch to the one stored on the blockchain. The system can then run in one of two modes. In the detection approach, the switch installs the flow rule anyway and the third party raises an alarm if there is a mismatch. In the prevention approach, the switch will wait for confirmation from the third party before installing the flow rule. This work is limited to only securing the connection between the switch and controller — if the controller is compromised then the malicious flow rule will be installed on the block chain and will be verified.

Lokesh and Rajagopalan propose a more complex method where three blockchain instances are used to represent switches, controllers and applications [140]. All transactions across the three layers are validated at each stage, with only fully validated flow rules being permitted onto the switch. The focus of the work is on malicious third-parties, in particular on the northbound API, and malicious third party applications, however it provides little protection against compromised, but authorised, applications or controllers.

Derhab et al. discuss BMC-SDN, a blockchain-based architecture for the multiple controller SDN network [65]. In this architecture, a primary controller is responsible for a network domain, along with multiple backup controllers. Backup controllers receive updates from the switch and process them, but do not directly control the domain. On sending an instruction to the switch, the primary will create a block and send it to the backup controllers, who will verify the update. If consensus is reached, then the update is written to the blockchain. Controllers are rated based upon a reputation mechanism, with controllers punished for pushing malicious updates, or being in the minority in validating updates. A controller with low enough reputation is flagged, ignored and reported to network administrators. The primary limitation in this work is that controllers push their own updates, and there is no validation that the report generated by a controller accurately represents the flow rule installed onto the switch.

3.3.1 Limitations

The primary limitation of efforts to detect faulty, or malicious, controllers through analysing the flow rules they install is that these approaches rely on knowing in advance what constitutes a problematic flow rule, which is usually a clear network violation such as a loop. If the compromised controller is able to install malicious flow rules which have an impact on traffic, they could potentially evade detection. Similarly, the attacker could perform their own analysis in order to test if their malicious flows will be detected. This type of approach is already performed in the field of adversarial-machine learning [88].

3.4 Mitigating Attacks in SDN

3.4.1 Securing the Controller

Scott-Hayward identifies three attributes for secure and resilient SDN controllers: secure controller design, secure controller interfaces and controller security services [204], and evaluates a number of SDN controllers against these attributes. Secure controller design includes application isolation, policy (rule) conflict resolution, multiple controller and application instances (for resilience) and secure storage. Secure controller interfaces covers secure control layer communication (for example use of TLS) and the security of any graphical interfaces and REST APIs. Finally, the controller security services includes IDS/IPS integration, authentication an authorisation, resource monitoring and logging/auditing services. None of the analysed controllers covered all of these features, partly due to the controllers being designed with a different focus (distribution vs resilience vs security).

ROSEMARY focusses on securing the network operating system from buggy or malicious applications by implementing application containment through launching applications in independent "micro-NOS" instances [216]. A micro-NOS is an individual instance of ROSEMARY itself, which isolates the application from

the underlying NOS kernel, as well as other applications. ROSEMARY supports the monitoring of resource usage by applications, and is capable of limiting resource usage, and can also limit the access of applications to privileged components, such as specific system calls. A motivating example within the work is a malicious application removing link information from the controllers datastore, which ROSEMARY is successfully able to prevent. ROSEMARY is shown to perform comparably with the BEACON and NOX SDN controllers, and with a much greater throughput than Floodlight, despite the overhead of the use of micro-NOS.

LegoSDN is a system to provide protection against application crashes, which could be caused by a malicious action such as a denial-of-service, which takes a sandboxing approach for SDN application [46]. Application crashes can cause inconsistencies in the data plane, for example if an application is installing a three switch path for a flow and crashes after installing flow rules on the first two switches, then the path will be incomplete. LegoSDN also applies a transactional approach to interactions with applications - a transaction is generated when a request is sent to an application, and is complete once the full response from the application has been received. A transaction buffer is maintained so that if the application crashes, it can be restarted and the request re-sent to being the application to the correct state. This is supported by functionality to create snapshots of application states. On application crash, LegoSDN can also revert any partial changes to the data plane, for example be removing any flow rules installed by the failed transaction. When tested using a modified version of the Floodlight controller, LegoSDN is shown to be able to recover from application failure in less the a second, much faster than a full controller reboot (which also causes a loss of network control). Rebooting an application is more efficient, only taking a few milliseconds, however a rebooted application will fail again if the switch request was deterministically faulty. LegoSDN incorporates an event transformer which can translate a switch request into multiple requests to correct against deterministic faults.

FortNOX is an extension to the NOX SDN controller which incorporates role-based authorisation and security constraint enforcement for SDN control [185]. The goal of FortNOX is to prevent flow rules from being installed within a switch,

which conflict with the pre-existing rule set, for example those of security appliances. The example conflict, dubbed dynamic-flow-tunnelling, describes an attack where HTTP access from $a \rightarrow b$ is blocked by a firewall. In the attack, flow rules with the `set` command change the source and destination IPs to modify the packet going through the firewall to appear to go to host c, however the packet is actually directed to host b, bypassing the firewall rule. FortNOX, applications are authorised with priority to install flow rules into switches, through the use of digital signatures. Proposed flow rules are checked for conflicts with the existing flow sets by reducing them to alias reduced rules (ARRs), which can be compared against existing flows. When conflict arises, flow rules are installed based on priority of the application that generated the rule.

Jo et al. propose a more complete solution for security network operating systems in the form of NOSArmor, which utilises several security building blocks (SBBs) to protect different network assets, which includes a rule conflict mediator derived from FortNOX [106]. Other SBBs include a host location tracker, link verifier and resource manager. Each SBB is implemented and its effectiveness tested. The performance impact of each SBB is shown, measured using the cBench tool [196]. Incorporating the SBBs does have a measurable impact on controller throughput (in responses/second) when compared to the base controller (Barista), with up to three of the SBBs tested at any one time.

3.4.2 Preventing Controller Poisoning

Deng et al. propose an approach for preventing controller poisoning through malicious PacketIn messages, in which the attacker sends packets through a switch with spoofed packet headers (including IP and MAC addresses) to greatly distort the controllers view of the network [64]. In this approach, hosts are mapped to physical switch ports by MAC addresses. If a new packet is observed for a mac address on a switch port and there is no stored mapping, the flow is legitimate. If there is an existing mapping for a host and a new packet is seen on a different switch port, the PacketIn is labelled as malicious and dropped.

3.4.3 Protection Against Malicious Applications

Wen et el. initially proposed a set of permissions for SDN applications, in particular the scenario of the use of third-party apps [235]. This approach is designated PermOF. In the proposed system, applications are granted access token allowing certain permissions, such as subscribing to certain notifications (such as PacketIn messages or topology updates) and the ability to add or remove flow rules from a switches flow table. This work was followed by SDNShield, a complete permission control system for SDN allowing administrators to apply fine-grained permission to applications, as well as allowing developers to handle rejected permissions gracefully [236].

OperationCheckpoint is an extension of the Floodlight controller which aims to secure the northbound API through the use of application permissions [205]. A set of permissions covering read (covering read operations that access controller information such as the network topology), write (such as installing/modifying flow rules) and notification (such as subscribing to `PacketIN` events) operations within Floodlight are defined, and a LinkedHashMap structure is used to store the permissions granted for applications. This covers both remote applications using the RESTful API, as well as local controller modules which function as applications.

SDN-Guard is a proposed architecture for preventing SDN rootkits [223]. SDN-Guard sits as a proxy between the switch and controller, and builds a model of the network view of the switch by monitoring flow rules on the switch-controller connection, and the network view of the controller by communicating the northbound interface of the controller. The focus is on rootkits which install malicious flow rules into the network but make attempts to hide their actions by modifying the controllers network view, causing a difference in the controllers network view and the actual state of the network. SDN-Guard utilised a detection component which can identify malicious flow rules using the network views, and then perform actions to remove them from the network (such as deleting malicious flow rules or re-inserting maliciously deleted rules).

3.4.4 Securing the Control-Data Plane (Southbound) Interface

The control-data plane interface (CDPI) is also a potential route for an attacker to gain control over the network, both to inject control commands through person-in-the-middle attacks, as well as eavesdrop on switch-controller communication to learn details about the network. The Open Networking Foundation‘s only recommendation for securing this channel is to utilise TLS on this connection, as specified in the OpenFlow switch specifications. This itself provides challenges due to the complexity and security challenges of secure key distribution for TLS [232, 155], as well as the additional extra latency and reduction in performance [165, 122].

To overcome the limitations of TLS, there have been works which address the security of the CDPI in a more lightweight fashion. Kreutz et al. describe the KISS protocol for security the control channel, utilising the light-weight NaCL cryptographic library [122, 21]. The core components of KISS are a shared secret key for encryption messages between the switch and controller, as well as an integrated device verification value (iDVV) for verifying devices.

3.5 Multiple Controller SDN control

Replication is a commonly suggested as a way to provide resilience in SDN [119]. Kreutz et al argue for replication, in particular with the diversity of control elements (for example utilising different SDN controllers) in order to protect against wide-scale compromise through common software vulnerabilities [119].

There have been many proposed and implemented approaches to introducing multiple controller SDN architectures [121]. There are multiple benefits to utilising multiple controllers, including scalability and robustness to failure. Of course, with distributing the SDN control plane comes challenges, including greater complexity and potentially reduced performance, as demonstrated by Levin et al. who show the trade off in state synchronisation versus performance of a distributed SDN load balancing application [135].

HyperFlow is the first distributed control planes for OpenFlow [229]. HyperFlow is a distributed but logically centralised architecture which makes use of a publish/subscribe model for providing consistency (built using WheelFS [221]. Due to WheelFS being resilient to partitioning, HyperFlow also is (as controllers are able to make decisions without contacting other controllers as part of the decision making process).

Controllers store updates within a distributed storage medium, and read updates from this system in order to maintain a global network view. As the controller does not contact other controllers to make routing decisions, there is no additional latency. The main difficulty is the controllers becoming inconsistent in periods of high load (if they cannot keep up with network updates). This is limited by the read rate of the storage system (writes can be concurrent so this is less of a bottleneck).

Whilst many SDN controllers support multiple controllers, ONOS was one of the first and is one of the most widely known distributed controllers [20]. ONOS replicated controllers with a goal of providing scalability. Multiple controllers run on servers. Each switch has a master controller, which is solely responsible for its programming, and also connects to a set of backup controllers that can take over as primary in the case of failure. The primary controller is chosen using a leadership election, built upon ZooKeeper [101].

ONOS maintains a global network view using a Blueprints graph implementation[1] over RAMCloud distributed key-value store. The individual controller instances are built using the Floodlight controller [187].

Jury is a consensus-based controller architecture in which multiple controller instances are used to verify the actions of a primary controller [142]. Jury is a module installed within controller instances and a separate validator, which receives updates from the controllers. There is an assumption that all controller instances are deterministic, and will return the same output for the same input query. This is partially backed by the use of a logically centralised datastore shared between the controller instances. Jury replicates switch requests amongst the secondary controllers, and all responses are sent to the validator to check if the response of the primary matches the majority of the backups. The system

[1] `https://github.com/tinkerpop/blueprints`

will not actively block controller actions, but can very quickly raise an alert if a fault is detected.

SDN-RDCD is a distributed SDN control architecture which detects compromised SDN devices, both switches and controllers [245]. The Openflow controller roles of master and slave are augmented with a third state as auditor. Each switch is controlled by a master controller, as well as one to several audit controllers. A switch sends `PacketIn` messages to both the master and the auditor(s). On receiving a request, the master also forwards this request to the auditor. The master sends its response to the switch and the auditor(s), and the switch then forwards this response, and the network state update, to the auditor. The auditor(s) can then use this set of information to identify if the switch or controller is compromised based on inconsistencies, and alert an administrator. This can also identify attacks such as a person-in-the-middle on the control network. If all auditors are compromised, the system fails, however just one non-compromised auditor is required to identify inconsistencies and alert an administrator. The approach is shown to be able to quickly identify compromised devices (within milliseconds), and has a low mCPU and memory overhead when compared to the base controller (ONOS).

3.6 Primary-Backup Fault tolerant SDN control

The majority of work to provide fault tolerance in the SDN control architecture utilises a primary-backup model. In these approaches, a single controller is responsible for a switch, with a set of backup controllers available to take over control of the switch in the case of failure of the primary controller. This most commonly refers to a fail-stop failure, where the primary controller becomes offline, for example through a software crash. These approaches commonly feature a logically centralised replicated datastore for ensuring consistent views of the network state across controllers. It is important to note that in this approach there may be multiple primary controllers operating within a network, however only a single controller acts as primary for any given switch. These approaches

often utilise the `MASTER` and `SLAVE` controller roles as specified by OpenFlow (see Section 2.9.1).

One of the earliest architectures that utilises a primary-backup model is SMaRtLight, as proposed by Botelho et al. [30, 109]. SMaRtlight assume a single primary controller is responsible for all of the switches within the network, with a number of replicas on different servers acting as backups, ready to take over as primary if required. To ensure controllers have an up-to-date view of the network a state shared datastore is used, built on top of replicated state machines [203, 31], implemented through the use of Paxos [126].The system is tolerant to fail-stop faults, where the controller stops working completely.

The primary controller is chosen by the datastore using a simple approach in which each controller, primary or backup, periodically request a lease from the datastore for a specified time. If there is no primary or the lease is expired, the invoker is made the new primary. If the invoker is the lease owner, they renew the lease. To reduce the amount of read operations to the datastore, controllers also maintain a local cache. The datastore responds to a request with the id of the current leaseholder and the time remaining on the lease. If a backup controller is made the leaseholder, then they must change their role to `OFPCR_ROLE_MASTER` on all switches.

The controller is built upon the Floodlight controller [187] and throughput is tested using cBench. The environment uses 2 controllers and three datastores tolerating a single fault on each layer. The system is evaluated with 1–64 switches, and a fixed number of hosts (1000). The system assume different caching levels for different applications (10%, 50% and 90%) which dictates how many read operations have to be made to the datastore. With 90% of operations absorbed by the cache, throughput is 367k flows/sec (compared to floodlights 2.5m). With only 10% of operations absorbed by the cache, the performance is reduced to 55k flows/sec. When a controller is made to be faulty, a backup becomes the new primary within 1 second, with the network returning to normal throughput after 4 seconds (this time includes the new primary updating its status on the 10 switches).

Fonesca et al. present a similar system for controller fault tolerance based on the primary-backup model, built upon the Nox controller [83]. The primary

difference between this approach and SMaRtLight is that in this approach, the primary controller shares updates directly with the backup controllers. On receiving a switch request, the primary will look for a matching entry in its storage table, and if one exists will respond to the switch with the flow rule. If there is no entry, it creates one and send a state update to the secondary controller(s), waits for an acknowledgement and then provides the switch with an response. Failure occurs when the primary stops responding to requests (a fail-stop failure). This is identified both by the switch using a timer when waiting for controller responses, and the switch periodically sending echo requests to the controller to check for liveliness. If the primary fails, the switch will then contact the secondary, who will become the new primary. The new primary will send status updates to the previous primary in order to inform them of the change, and instructing the previous primary to become a secondary.

The system is tested using a simple mininet network. The primary experiment has one primary and one secondary, and measures the latency for controller responses (for the purpose of this experiment all packets in a flow are sent to the controller). In normal operation, the latency is around 22ms, which increases to almost 900ms when the primary fails, due to the changeover, before gradually decreasing once the network stabilises. A second experiment measure the effect of the number of replicas, as the primary has to contact these before replying to the switch. With no replicas, the response time is around 8ms. For one replica, this increases to 14ms, 2 replicas 40ms and 3 replicas 62ms. This provides some doubt on the scalability of the technique.

Ravana is a SDN controller tolerant to fail-stop crashes of both controllers and switches [109]. The key difference to the previously described approaches is the consistency guarantees provided by the system, which ensures state updates and switch request are processed in the correct order across all controllers. The system also has a goal of keeping the architecture transparent to application developers — the controller still appears as a single controller to controller applications. Ravana provides fault tolerance to f faults with $2f + 1$ replicas.

Ravana treats the entire event-processing cycle (from event delivery from the switch to command execution on the switch) as a transaction, and either all or none of the components of this transaction are completed. Ravana also guarantees

that transactions are totally ordered across replicas and each is executed exactly once across the system. The controllers are based on replicated state machines with a mechanism to ensure consistency. The master controller is chosen using a leadership election built using ZooKeeper [101].

Ravana uses a two-phase replication protocol to deal with switch-state consistency. Each phase involves adding event-processing information to a replicated in-memory log. The first stage ensures that every received event is reliably replicated, and the second conveys if the event-processing transaction has been completed. Further, the system introduces acknowledgements for messages sent between the switch and controller.

At any one time, the master replica is responsible for actually controlling a switch, while the replicas act as slaves, receiving messages from switches but not controlling the switches. On controller failure, a leader election runs on the slaves to choose a new master. It then finishes processing any logged events (without sending any commands to the switch), and informs the switch that it is the new master.

The controller is implemented on top of Ryu 3.8. Using the PyPy interpreter, vanilla Ryu and Ravana has throughputs of 67.6k and 40.4k responses per second respectively. Ravana is capable of processing most events on average in 12ms (with network latency removed). In the case of failure, Ravana can recover in 75ms (std dev of 9ms).

3.7 Byzantine Fault Tolerant SDN Control

In Section 3.6 I discuss works that apply a primary-backup model to SDN control which utilises multiple controllers, however only a single controller is responsible for a switch at any one time. In this section I discuss works in which multiple controllers are simultaneously responsible for a switch through the use of some form of consensus mechanism, such as a majority vote or the application of a byzantine fault-tolerant (BFT) algorithm.

In one of the earliest pieces of work to examine the effectiveness of a BFT architecture in SDN, Li et al. [136, 137] explore the issue of controller assignment

in a BFT architecture, taking into account the differing levels of security required by individual switches (a switch with higher security requirements is assigned a greater number of controller replicas to provide greater resilience). For this purpose they first define the controller assignment in a fault-tolerant SDN (CAFTS) problem. CAFTS represents the problem of assigning controllers to switches, satisfying the requirements of the BFT algorithm in use, minimising the latency between controllers assigned to a single switch (to aid in the performance of the BFT algorithm) and to maximise the utilisation of controller resources. Their requirements first assignment (RQFA) algorithm is shown to provide more efficient controller assignment than a randomised approach. The suggested fault tolerant algorithm is PBFT [41]. They perform a simple evaluation of BFT control using both local replicas, and replicas hosted within a public cloud (Google compute), for setting up flows of increasing path lengths (up to 10 switches). They find that the additional latency introduced by BFT is minimal (up to 13% for 10 switches when using replicas in the cloud), in particular for short paths of less than 4 switches, though they do not specify which BFT protocol is in use for these practical tests and how many replicas were utilised. Further, it is not clear from the presented results what level of additional latency is incurred when using local controllers as this is only represented in plotted results, however on visual inspection this seems greater than the 13% increase worst case quoted for the public cloud scenario.

ElDefrawy and Kaczmarek implement the BFT-SMaRt protocol ([22]) within an SDN controller architecture [76]. This work focusses on the performance impact of applying the BFT protocol. Testing was performed by producing modified versions of the OpenFlowJ and Beacon SDN controllers, and the use of a proxy between the switch and controllers to implement the switch logic. When evaluated using a Mininet [130] network of 64 hosts and 63 switches in a binary tree layout, on setting up a flow with a path length of 11 switches the BFT approach results in a 2× slowdown with the modified OpenFlowJ controller, and a 6× slowdown with the modified Beacon controller. For a single switch, the flow setup time when using BFT increased from 9.44ms to 31.7ms when using OpenFlowJ, and 0.5ms to 14.5ms when using Beacon, with the number of flows able to be processed per second also reduced by similar amounts.

Sakic et al. propose a solution for handling both byzantine and fail-stop failures in SDN control, utilising an approach in which a switch contacts a primary controller, but also a set of secondary controllers, and uses the multiple responses to identify inconsistencies [199]. In the approach, MORPH, a switch communicates with a minimum of $2F_M + F_A + 1$, where F_M is the tolerated number of byzantine-failed controllers, and F_A is the tolerated number of offline (fail-stop) controllers. The focus of the work is on the problem of controller reassignment in such a controller architecture — on detecting a malicious or unavailable controller, the system applies a reassignment function in order to automatically reassign the switch to a new set of controllers, excluding those suspected of failure. One aspect of the approach is that as faulty controllers are detected, the number of required primary or secondary controllers for a switch is reduced. It is explained that this is to minimise the control plane overhead, but without further reason. It is assumed that as controllers are removed, reducing the requirement per switch allows the control plane to continue functioning with less resources. As the paper does not consider malicious controllers, it is unclear if this controller removal could be abused to cause a denial-of-service attack on the MORPH system, Evaluation assumes that F_m and F_A are both 5, so 16 controllers are required for operation, along with 34 switches. As well as being shown to handle faults, the system is shown to improve performance, in particular in terms of control-plane synchronisation overheads, as faulty controllers are removed from the network. Further, as the number of required controllers is reduced with the detection of a faulty node, switch request processing time reduces as more failures are detected.

Mohan et al. explore a solution that utilises $f + 1$ primary controllers which interact directly with the switch, and a further f backup controllers which can be consulted in the case of a failure [158, 159]. The work focuses on the issue of controller assignment, where they find that in their approach up to 50% fewer controllers are required than in the traditional $3f + 1$ BFT approach, with each controller on average experiencing 50% less load. Whilst this work simulates the controller assignment problem, they do not evaluate the effectiveness or performance of the approach. The proposed solution is similar to the SDBFT approach, however the work focusses on the controller assignment problem and not the core

protocol itself. It should be noted that this work was released after development of the SDBFT protocol had been completed.

Qi et al. describe a simple fault-tolerant approach in which multiple (M) controllers are contacted by the switch (where $M \geq 3$), with the majority output from controllers taken [190]. The architecture incorporates a "scheduling plane" which sits in front of the switch and selects from the pool of controllers, processing responses from controllers and sharing topology information between the set of controllers. In order to provide robustness the set of controllers is chosen to be diverse, for example a switch will contact an instance of the POX, NOX and Floodlight controllers, however each diverse controller is expected to return the same result. This work is not, however, practically evaluated.

Across all of these examples, except for the works by Mohan et al. and Qi et al., a full BFT protocol is utilised, which has the drawback of the overheads of the BFT protocols including additional rounds of communication (and the associated network load), and additional latency. The longer it takes to process a switch request, the fewer requests can be handled by the control architecture. The works which utilise lighter BFT protocols (Mohan et al. and Qi et al.) have limited practical evaluation of their designs, with, for example, Mohan et al. focussing on the controller assignment problem rather than the core protocol.

3.7.1 Consensus amongst administrators

Whilst the previously mentioned works focus on consensus amongst software in the form of controllers, Matsumo et al. utilise consensus amongst network administrators in order to prevent malicious administrators from deploying malicious configurations to controllers [147]. The proposed solution is the Fleet controller — a logically centralised but distributed controller in which each administrator has their own instance of the controller, managed by an administration layer, and a switch intelligence layer (which is also part of the controller) which mediates communication between the switches and administration layer. Fleet can take one of two proposed approaches. In the single configuration approach a threshold of administrators must agree on a configuration, with $2k + 1$ administrators required to handle k malicious administrators. Switch configurations

are encrypted using a threshold encryption standard (in this case Shamir's secret sharing [212]), and the switch intelligence layer can successfully decrypt the configuration as long as the majority of responses match. In the second approach, dubbed multi-configuration, each administrator installs a routing configuration and installed into the network providing multiple routing planes, any of which can be used. As long as one is non-malicious then the network should be able to function. This has the challenge of how to choose which routing plane to use at any one time, for example by introducing flow metrics onto the switch to evaluate how well a particular routing plane is working.

3.8 Discussion

The previous work on preventing failures takes two main approaches. The first approach involves the addition of attack detection and prevention mechanisms to a single controller (as discussed in Sections 3.3 and 3.4). Many of these systems have limitations in that they often apply to auxiliary aspects of the control plane, such as applications, and would not function properly if the controller itself is compromised. Similarly, most approaches of this type stop protection at the controller level and do not examine the southbound interface, meaning that if an attacker was able to perform a person-in-the-middle attack and modify controller responses these would not be detected. The system proposed in this work, SDBFT, solves this problem by moving the verification step to the switch level and so provides protection if any aspect of the control architecture outside of the switch is compromised.

The second approach is to make use of multiple controllers, such as through the use of distributed control planes (Section 3.5), which usually result in a primary-backup approach where on detecting a fault with a primary controller, a backup controller can be utilised to maintain control of a switch (Section 3.6). These approaches have a major limitation in that the majority of these approaches only provide protection against fail-stop faults where a controller stops operating completely, and would not protect against an actively malicious or compromised controller.

Table 3.1: Existing literature in Byzantine Fault Tolerant SDN control. f = number of faulty nodes

Paper	**Approach**	**Paper Focus**	**Required Nodes**	**Practical Evaluation**	**Performance Overhead vs single controller (if provided)**
Li et al. [136, 137]	PBFT control plane	Controller assignment and load balancing	$3f+1$	Some limited evaluation, number of nodes not clear. Local and cloud controller placement tested. Evaluation focus on controller assignment	13% in local deployment, not provided for cloud deployment but much greater.
ElDefrawy and Kaczmarek [76]	BFT-SMaRt control plane	Deployment of control plane and performance overhead	$3f+1$	Implemented as modified versions of OpenFlowJ and Beacon controller. Tested using Mininet, 1 and 10 switch path lengths with 4 replicas	2x slowdown (best result)
Sakic et al. [199]	Backup verified control plane (single primary with multiple backups for verification	Controller reassignment	$2F_M + F_A + 1$, where F_M = byzantine faults and F_A = fail stop faults	OpenVSwitch and Docker setup	N/A (Reassignment and switch reconfiguration tested only)
Mohan et al. [158, 159]	Multiple controller primary-backup consensus based control plane	Controller assignment and reassignment	$f+1$ with f backup $(2f+1)$	None (numerical only)	N/A
Qi et al. [190]	Majority vote multiple controllers	Controller scheduling and assignment	≥ 3	None	N/A
SDBFT	Multiple controller primary-backup consensus based control plane	Protocol design, security through signatures, controller synchronisation, controller assignment, performance overhead, practical evaluation	$f+1$ with f backup	Three test platforms (Mininet simulation, virtual environment, physical switches). Baseline performance testing, failure operation and controller load testing	4 controllers: 60% without signatures, 178% with signatures

More relevant to this work is are the distributed control plane approaches which apply a BFT, or similar, algorithm in order to prevent compromised controllers from performing malicious actions, as discussed in Section 3.7 above, and summarised in Table 3.1. These existing works usually require a minimum of $3f + 1$ controllers to handle f faulty (or compromised) controllers, over many rounds of communication. This presents a very large communication overhead, and also requires a large amount of replication of controllers due to the requirement of all $3f + 1$ controllers assigned to a switch to actively handle all requests from that switch. SDBFT reduces this to $f + 1$ with f backup nodes by relaxing the fault tolerant requirement to fault detecting with recovery. The previous work by Mohan et al. follows a similar approach, however the work focusses on the controller assignment problem rather than the core performance, and therefore practical usability, of the approach [158, 159]. Those that do provide practical evaluations generally do not test with malicious or faulty controllers in place, or give further considerations to wider security properties such as non-repudiation. Within this space, SDBFT represents the first approach which covers all aspects of the potential deployment including controller assignment, signature use and controller consistency backed up by an extensive practical evaluation.

The standard practice within the literature is to measure the flow setup time in milliseconds, measured through the use of a ping request and extracting the round trip time of the first packet. A baseline is generated using a traditional single controller setup. This can then be used to directly measure the additional overhead of the modified control plane by measuring the difference in the round trip time between the baseline and modified controller. Further, a large amount of the literature measures the controller bandwidth in flows handled per seconds, computer using a controller benchmarking tool such as cbench. This allows the measurement of controller performance whilst under load.

3.9 Conclusion

In this chapter I discussed various works which cover the security of SDN controllers. This includes works which aim to detect compromised controllers, secure

the control plane from compromise and introduce fault tolerance into the control plane.

In the next chapter I explore the impact of a compromised SDN controller by examining the network-level attacks that can be launched through a compromised controller, demonstrating these attacks in a simulated network environment. I also explore the impact of these attacks in the specialised environment of Industrial Control Systems (ICS).

Chapter 4

Insider Attacks in Software Defined Networks

4.1 Introduction

When designing a defensive system, it is important to examine which threat actors we are defending against, including what their end goals are. It is also important to understand the impact of the attacks which I am trying to defend against. This knowledge can help us to ensure that implemented defences are adequate to protect against the attacks which may occur. In previous works it it is often stated that if an SDN controller were to become compromised, then the attacker could cause serious harm to the underlying network ([119, 207, 210, 206]), but in what ways and to achieve what attackers' goals?

In this chapter, I first give an overview of the attacker who would compromise an Software Defined Network (SDN) controller, including their goals and attack vectors. I then discuss a number of potential attacks against a network that can be launched from a compromised SDN controller, and demonstrate a number of these on a small-scale virtual network. I then also provide an exploration of the impact of SDN based attacks on a real-world scenario in the form of industrial control systems, where network availability is paramount to ensuring the safe operation of the systems.

4.2 Attacker Model

In this section I discuss the types of attacker that may attempt to attack an SDN network. I then discuss how that may actually introduce malicious flow rules into the network, and then talk about the various goals they may have when attacking.

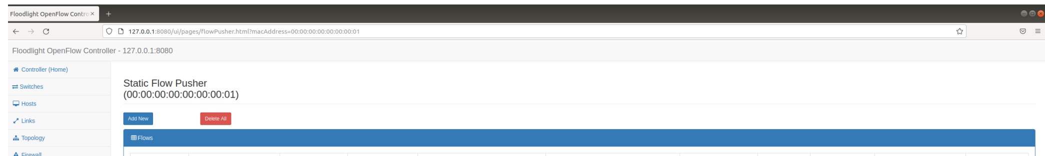
4.2.1 Attacker

I assume a targeted, well resourced attacker that is focused on a particular network. The attacker would compromise the SDN controller in a targeted fashion to disrupt the network below in the ways discussed in this chapter. Whilst the attacks described in this chapter are relatively simple to implement at a technical level, a high level of manual effort is required in order to perform anything past basic examples. I assume the attacker will be part of an organised crime gang, a nation state or similar which are targeting the network of a particular organisation for either financial or political motivation. These types of attackers often come under the term *Advanced Persistent Threat* (APT) groups.

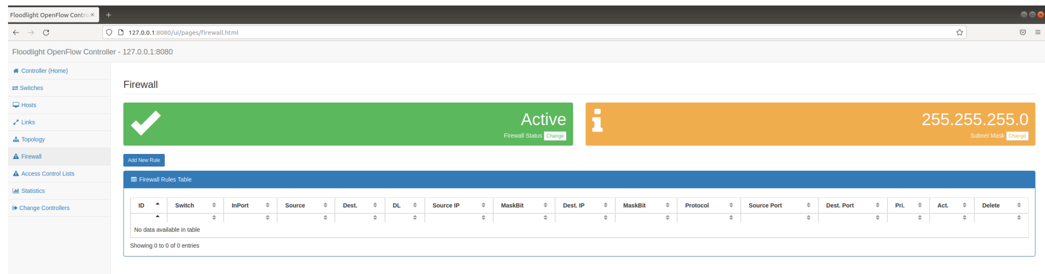
There are of course risks to the SDN controller host becoming compromised by wider-ranging malicious campaigns such as ransomware, however I assume that a controller affected by such malware will become non-operational producing a fail-stop failure, rather than malicious control within the network.

4.2.2 Attack Vector

There are a number of possible ways in which an attacker could compromise the integrity of a controller in order to send malicious commands to a switch [208]. As well as directly compromising the controller code, an attacker could also compromise third party applications that communicate with the controller, or the communication channels of the northbound or southbound interfaces. For the majority of the approaches it is assumed that the attacker has already gained some level of access to the target network. An attacker could compromise, or interfere with, the controller in one of the following ways:



(a) Static flow pusher



(b) Firewall configuration

Figure 4.1: Floodlight web administrative interface

- **Compromised administrative interface** Most SDN controllers provide functionality for remote administration of the SDN controller through APIs or administrative interfaces (such as web-based pages). If a host is compromised that is authenticated to these interfaces (such as an administrators terminal) then the attacker can gain access to the control provided by the administrative interface, and all functions provided by it. Typically this will include changing the set of installed applications or manually installing/modifying flow rules on switches [119]. Figure 4.1 shows the web interface for the floodlight controller, which is unauthenticated, and allows the installation of flow rules and configuration of the firewall.
- **Compromised Northbound Interface** The attacker compromises a northbound interface in order to interact with the controller, for example, by performing a person-in-the-middle attack on a RESTful interface, or by exploiting a lack of authentication on the interface.
- **Compromised Southbound Interface** The attacker compromises the OpenFlow connection between the switch and controller — for example, by performing a person-in-the-middle attack — and modifies requests and responses between the two, or injects packets into the channel. Whilst the use of an

encrypted channel through TLS is recommended within controller documentation, this is usually not the default when installing a controller (with a simple TCP connection being standard), and requires appropriate certificates to be installed and managed.

- **Malicious Application** A malicious application is installed on the controller by either a malicious administrator, through administrator error or through the compromise of an existing application. The application is limited to the functionality of applications provided by the controller, and not more. Matsumo et al. focus on providing defences against a malicious administrator [147].
- **Poisoning Controller Network View** The attacker uses crafted packets on the network to poison the topology datastore of the controller, causing routing errors [98, 168]. Ujcich et al. demonstrate an attack in which a malicious application poisons the shared control plane to cause a legitimate application with higher privilege to install its desired flow rules [230].
- **Compromised Controller** Through a software vulnerability, the attacker gains control over the controller. As an example, Röpke and Holz describe a proof-of-concept rootkit able to infect a network operating system (or controller), specifically OpenDayLight and HP SDN Wan [194].
- **Compromised Host** In the most severe case, the host on which the controller resides is compromised, resulting in full attacker control over the controller software, and traffic in/out of the host.

Within this work, I focus on the worst case compromised host/malicious controller and application use case in which the attacker gains arbitrary control over the routing decisions made by the controller, though the majority of the attacks would also be possible using the alternate attack vectors.

4.2.3 Attacker Goals

I define five main goals of the attacker who has gained control of the SDN controller:

- **Denial-of-service** The attacker wishes to prevent either a single host (targeted) or set of hosts (indiscriminate) from communicating by preventing all communication of the hosts.

- **Eavesdropping** The attacker wishes to collect the traffic of either a single host (targeted) or group of hosts (indiscriminate), without affecting the availability of the network service, in order to gather information.

- **Data tampering** The attacker wishes to change the contents of packets for a particular host in order to carry out a person-in-the-middle attack, or redirect a victim to an attacker controller service.

- **Service degradation** The attacker wishes to degrade the performance of the network for a single host (targeted) or all hosts (indiscriminate) in order to make the network unusable, or to introduce errors in external applications that rely on high speed communications. Similarly, the attacker can cause failure in SDN-based applications. For example, Costa and Costa [56] demonstrate issues with FlowVisor [215], which provides network isolation using SDN, which can break isolation between network slices.

- **Attack Augmentation** The attacker uses malicious flow rules to assist in further attacks, for example by allowing attack traffic through firewalls [185].

4.3 Attacks

I now provide an overview of the set of attacks that can be performed by a compromised controller, separated by type. Each of the following attacks could be implemented by either a malicious or compromised controller application, or by a controller itself being compromised. A taxonomy of these attacks can be seen in Figure 4.2. Attacks for which citations are provided are taken from the literature, whilst those without citation are novel to this work.

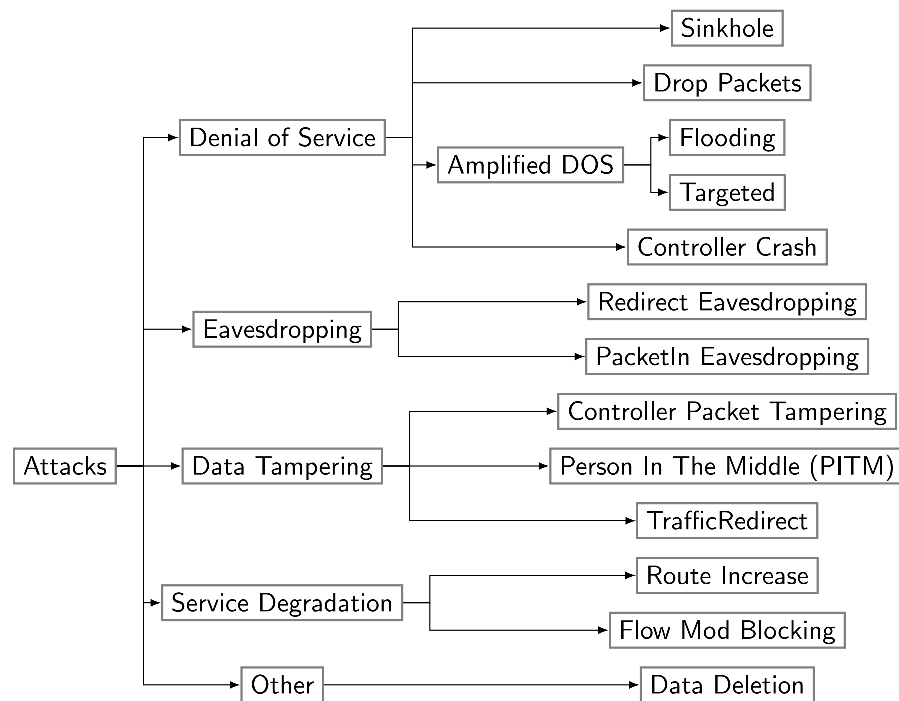


Figure 4.2: Attack Taxonomy

4.3.1 Denial of Service Attacks

The denial of service attacks attempt to prevent hosts from communicating. They can range from causing all traffic to be dropped, or being targeted to the level of preventing one host from accessing one service.

- **Sinkhole** In the sinkhole attack, the compromised controller directs targeted flows to a port that leads to a sinkhole, so the traffic will eventually be dropped. In a coordinated version of this attack, the sinkhole is located a few hops away from the target host and the compromise controller routes the packet down a path to the sinkhole to make it appear as a genuine routing error.
- **Drop packets** The simpler version of the sinkhole attack is to simply install flow rules that dictate the packet should be dropped. This can be targeted to only drop packets relating to a particular source/destination, or be generalised by installing flow rules that cause all packets to be dropped.

- **Amplified DOS (flooding)** In this attack, the attacker can insert flow rules that push packets out of all available ports on a switch by utilising multiple action fields within a flow rule. This will cause packets to be repeatedly duplicated at switches, flooding portions of the network. The more switches under the control of the attacker, the greater the effect of the attack. The overall effect will be a drop of performance for all hosts within the network, potentially to the point where the network becomes too congested.
- **Amplified DOS (targeted)** Similar to the previous attack, this involves pushing packets out of multiple ports on a switch in order to cause a performance hit at a target destination. However, in this approach I assume the attacker knows the network topology and ensures the packets are duplicated along particular paths only. This means that the rest of the network should be largely unaffected, but a target host could be taken offline.
- **Controller crash** In an extreme case, the controller can be taken offline by terminating the running instance. This attack was previously described by Shin et al. [216]. Note that this is the only described attack that is not directly preventable through the use of SDBFT, as no responses are sent from the controller to the switch, however SDBFT will enter failure mode if a non operational controller is detected. In a variation of this attack, Shin et al. also demonstrate an attack against the Floodlight controller in which the attacker increases the resource usage of the controller until it surpasses the available system memory, causing the java virtual machine to crash and exit.

4.3.2 Eavesdropping attacks

These attacks allow the attacker to intercept the traffic on the network that they would not usually have access to.

- **Redirect eavesdropping** In this attack, the attacker redirects traffic to another location where data can be collected. This can be done either by directing the flow through an alternate route (as in the route increase attack below), or through packet duplication (a copy of the packet is sent out a

second port to a collector), effectively configuring a mirror port on the switch. A version of this attack is presented in [100].

- **PacketIN eavesdropping** The attacker prevents a matching flow rule being installed on a switch for a target flow. Depending on the switch configuration packets are either buffered on the switch with only packet headers included in the PacketIN message, or the packet is not buffered and the full packet data is included. If full packets are sent to the controller, then a malicious application can log these packets and the contents can be parsed to extract information. This has the secondary impact of a timing attack, as requiring all packets to pass through the controller will introduce extra latency to flows, and is functionally equivalent to the flow mod blocking attack described below, therefore I do not demonstrate this attack specifically.

4.3.3 Data Tampering Attacks

The data tampering attacks allow the attacker to modify network packets, either directly on the controller itself, or by redirecting traffic to an alternate location on the network.

- **Controller packet tampering** An extension to the flow mod blocking and PacketIN eavesdropping attacks, in this attack the controller goes further and modifies packet contents whilst they are being processed, effectively performing a controller in the middle attack.
- **Person-in-the-middle (PITM) attack** The above eavesdropping, with slight modification, can be used to aid in person-in-the-middle (PiTM) attacks [12]. The attacker can redirect packets through a device that they control within the network, then return the packet back to the switch to be forwarded onto the intended target. The attacker can then read the packets to collect data, or modify them as required. This is effectively the same as the previous attack, although packets are not duplicated.
- **Traffic Redirect** In this attack, the attacker performs a variation on the PiTM attack, which allows them to build a malicious replica of a target server, and redirect client requests to the attackers replica instead of the target. This

could, for example, aid in phishing attacks. The client believes they are accessing the correct domain name, and indeed server IP, however all requests are redirected to the attacker controlled server.

4.3.4 Service Degradation

The attacker wishes to degrade the performance of the network for either a targeted host, or set of hosts. This is another form of denial-of-service attack, however unlike the previous set of attacks which completely block communication, these attacks reduce network performance to cause a loss of performance of running services. This could be be particularly problematic for systems which require real time, or near real time, communication.

- **Route increase** In this attack the attacker introduces an extra hop into the path of the packet wherever possible. This is done by pushing the packet to an alternate port on the switch than the one that would usually be chosen. It is assumed the at the next switch the packet route will be corrected ensuring that the packet actually arrives, just with an additional delay. The attacker can repeat this attack on multiple switches that they control in order to introduce further delay.
- **Flow mod blocking** Lee et al. [133] demonstrate an attack against the ONOS controller, which can be easily generalised to apply to the compromised controller scenario. In the example, the attacker uses a malicious application to change the ONOS properties to result in the `PACKET_OUT_ONLY` option to be set to true. This prevents the forwarding application from installing flow rules on the switch, only being able to respond to PacketIn messages with PacketOut messages. This results in all traffic being sent through the controller, which degrades network performance. In the example given, ping times increase from 1ms to 4ms.

4.3.5 Other Attacks

- **Data deletion** In this attack, as described by Shin et al. [216], an attacker-controlled application modifies entries within the controllers datastore in order

to affect the operation of other applications running on that same controller. An example provided is the removal of network link information from the datastore, causing a monitoring application to display the wrong information. This attack can be extended to apply to a distributed controller architecture, wherein multiple controller share single, yet distributed, datastore. In this scenario, a controller can modify dataset entries or misreport network state (such as the existence of links), causing other controllers to see invalid information. I do not replicate this attack in my environment, however I refer to the related work in which the attack is demonstrated against multiple Openflow controllers [216].

4.4 Attack Demonstration

To demonstrate the effectiveness of these attacks, I produce an implementation of each attack and test, depending on the attack goal, the impact on network performance, or if the outcome of the attack is achieved (in the case of data tampering attacks).

4.4.1 Setup

To demonstrate the attacks I create a malicious instance of the Java-based Floodlight controller. Attacks were implemented by writing a malicious application for each attack, mostly achieved by making modified copies of the default `Forwarding` routing application. As Floodlight allows applications to specify their own position in the packet processing queue, the malicious applications were configured to be executed before the default routing applications (where applicable). An example of this is shown in Listing 4.1. In this example, the two functions dictate that for `PacketIN` messages, the packets should first be processed by the `DeviceManager` application before the malicious application, and should not be processed by the default `Forwarding` application until after the malicious application has processed the packet. An application can prevent further applications

from processing the packet by returning `Command.STOP` from the message processing function (returning `Commmand.CONTINUE` allows the request to move onto the next application in the queue). Configuration for the malicious applications (setting target IP/MAC addresses) was done through the default Floodlight properties file, as is also the case for legitimate applications.

```
@Override
public boolean isCallbackOrderingPrereq(OFType type, String name) {
      return (type.equals(OFType.PACKET_IN) &&
  (name.equals("devicemanager")));
}

@Override
public boolean isCallbackOrderingPostreq(OFType type, String name) {
    return (type.equals(OFType.PACKET_IN) && (name.equals("forwarding")
      || name.equals("virtualizer")));
}
```

Listing 4.1: Malicious application ordering insertion code

Testing was performed using the Mininet platform [130]. Mininet is a simulated SDN development and testing platform deploys can deploy virtual SDN switches (OpenVSwitch) and simulated hosts with a given topology. This was chosen as it is an easy way to build a large scale network with a custom topology, and also to tear down and cleanup the network after the attack has occurred. As some of the attacks, such as the amplified DoS, require a large number of switches and routes, Mininet provides an easy way to achieve this scale. Further, Mininet can be easily configured to use sequential MAC addresses for switches and hosts, meaning that it is easy to predict which hosts and IP addresses map to which MAC addresses A simple topology of six switches, each with one connected host, was used for the majority of testing, which can be seen in Figure 4.3. This topology is not necessarily designed to replicate a real-world topology, but it allows me to clearly demonstrate the effectiveness of the attacks. In a real-world setting, the effect of some attacks may be limited, in particular in a sparser network with fewer available routes.

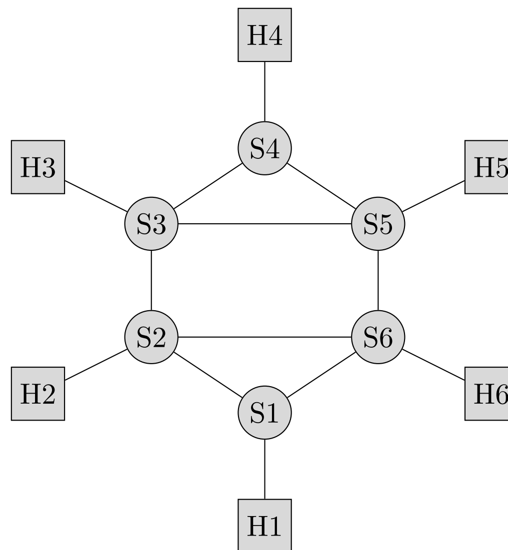


Figure 4.3: Simple network used for attack demonstrations with six switches and six hosts

4.4.1.1 Malicious Applications

I will now describe the specific actions taken by each malicious application.

- **Sinkhole** I do not explicitly demonstrate the Sinkhole attack, as I clearly demonstrate the rerouting of packets through the route increase, redirect eavesdropping and traffic redirect attacks, and the overall outcome of the attack is identical to the drop packets attack described below.
- **Drop packets**

  An application, *TargetDropper*, was produced which takes a list of IP addresses from the Floodlight properties file and instructs the switch to drop (through an action-less flow rule) packets for flows for which one of the target IPs is in the destination. This application is based on the default *Forwarding* application, and is set to be run before any other application. The target dropper will return `Command.STOP` if a target IP is seen to prevent packets being passed onto the `Forwarding` application. If the packet is not for a targetted IP, the application will allow the packet to pass onto the next application in sequence for normal processing. The `processPacketInMessage` method where this

decision is made can be seen in Listing 4.2. The `doDropFlow` function already exists within the forwarding application, and installs a flow rule directing the switch to drop matching packets. Similar logic is used for all malicious applications where a target is used.

```
@Override
public Command processPacketInMessage(IOFSwitch sw, OFPacketIn pi,
IRoutingDecision decision, FloodlightContext cntx) {
  Ethernet eth = IFloodlightProviderService.bcStore.get(cntx,
IFloodlightProviderService.CONTEXT_PI_PAYLOAD);
IPacket pkt = eth.getPayload();

if (pkt instanceof IPv4) {
  IPv4 ip_pkt = (IPv4) pkt;
  log.info("Packet found: " +
ip_pkt.getDestinationAddress().toString());
  if (targets.contains(ip_pkt.getDestinationAddress().toString())) {
    doDropFlow(sw, pi, decision, cntx);
    return Command.STOP;
  }
}
    return Command.CONTINUE;
}
```

Listing 4.2: TargetDropper main logic

- **Amplified DoS (flooding)** The *AmplifiedDOS* application is a modified version of the default Forwarding application. Any flows destined to target IP addresses are handled by the AmplifiedDOS application, all other flows are passed on to the standard Forwarding application for processing. The application installs a flow rule on every switch the packet traverses with multiple action fields, one for each active port on the switch. The multiple action fields direct the switch to forward the packet out of all available ports on the switch. Eventually every switch under the attackers control will forward the flow out

of all available ports, which causes an exponential growth in the number of packets being forwarded as packets are repeatedly duplicated. Whilst this is not a targeted attack, I specify a target flow to duplicate so other hosts can be used to test the effect.

- **Amplified Dos (targeted)** The *AmplifiedDOSTargeted* is similar in functionality to the *AmplifiedDOS* application, however when generating a flow rule it will only duplicate the flow to a subset of the ports of a particular switch which will direct the duplicate packets in the direction of the target host. For testing purposes this is hard-coded into the application. For the example network in Fig 4.3, if I assume H4 is the target and I have a flow from H1 to H4. The application will cause the flow to be forwarded to switch S2 and S6. The application will then instruct S2 to forward to S3 and S6, and switch S6 to forward to S2 and S5, and so on. In order to prevent exponential packet growth, the flow rules prevent packets being sent back out of the port they came in on. This results in duplicate packets being received by the target with less chance of flooding the network. For example, on the test network a flow from H1 to H4, with H4 as the target, the attack results in 8 packets being received by the H4 for every packet sent by H1.
- **Controller crash** This application terminates the current Java Virtual Machine (JVM) instance when a `PacketIn` message to a particular IP address is observed. The JVM is terminated by a simple call to `System.exit(0)`.
- **Redirect eavesdropping** The *Eavesdropping* application is almost identical to the default forwarding application, except that for any flow to or from the target addresses, an extra action will be added to the flow rule outputted by the application mirroring the traffic to the appropriate port where the data collection machine is sat.
- **Person-in-the-middle (PiTM)** The *PiTM* application is a modified version of the Forwarding application which is programmed to configure flow rules for the target flow to output on the first port on which the attack machine is connected. The attack machine has two connections to the switch, and will output packets on the alternate connection to the one they were received

on. The malicious application will forward any flows coming from the second attack port onto the original target. This is done in both directions for the target flow.

- **Traffic Redirect** The *Redirect* application, built upon the Forwarding application, intercepts packets to the target server IP address. For any packets destined for that address, the application sets the outbound port for the `FlowMod` and `PacketOUT` to the port on which the attacker controlled host is connected.
- **Route increase** The *RouteIncrease* application manually modifies the installed route between H1 and H4. By default, the route that is installed would be H1–S1–S2–S3–S4–H4, representing a path length of 5. Through modifying flow rules, the malicious application instead directs the flow on the H1–S1–S2–S6–S5–S3–S4–H4 path which visits all switches, increasing the path length to 7.
- **Flow Mod Blocking, PacketIN Eavesdropping and Controller Packet Tampering**

  The *RuleBlocker* application is used for preventing the controller from installing flow rules onto the switch. This application is almost identical to the standard *Forwarding* application, except that it will only allow target packets to be forwarded through `PacketOut` messages, and will not generate flow rules. This means that all packets in the flow have to pass through this application. Packets that are not in target flows are passed onto the next application for normal processing. The application can modify packet contents, allowing for packet tampering. The attack can be targeted or indiscriminate.

4.4.2 Results

For testing, I use host H4 as a target for attack. To measure a baseline performance, I make 10 pings of 4 packets each from H1 to H4, with a 6 second pause between each (the default flow rule inactivity timeout is 5 seconds), and take the mean round trip time (RTT) of the packets. A ping from H1 to H4 has an average RTT on the first packet (where flow setup occurs) of 7.95ms (SD 1.62), with

the subsequent packets, matched to a flow rule, having a mean RTT of 0.105ms (SD 0.042). These figures assume the controller already knows the location of each host, which would be the case in an established network. In cases where the destination is not know, the flow setup time is greater as an initial packet flood is performed to identify the location of the host. This results in a slightly larger flow setup time, with greater variation, however subsequent packets on the flow are unaffected. I also use the `pingall` command provided by mininet to test connectivity between all hosts. This performs a single packet ping between each pair of hosts, printing `X` if there is no connection between two hosts. An example of the output of `pingall` when not under attack is below:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6
h2 -> h1 h3 h4 h5 h6
h3 -> h1 h2 h4 h5 h6
h4 -> h1 h2 h3 h5 h6
h5 -> h1 h2 h3 h4 h6
h6 -> h1 h2 h3 h4 h5
*** Results: 0% dropped (30/30 received)
```

Listing 4.3: Mininet `pingall` example

4.4.2.1 Denial-of-Service

Controller Crash When a packet destined for H4 is seen, the malicious application kills the JVM, resulting in the controller going offline. As no backup has been configured, no communication is possible within the network. This can be seen in the `pingall` output below:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 X X X
h2 -> X X X X X
h3 -> X X X X X
h4 -> X X X X X
h5 -> X X X X X
h6 -> X X X X X
*** Results: 93% dropped (2/30 received)
```

Listing 4.4: Controller Crash `pingall` Output

Drop Packets I drop all packets destined for H4 using the `TargetDropper` malicious application. The output of `pingall` is below:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 X h5 h6
h2 -> h1 h3 X h5 h6
h3 -> h1 h2 X h5 h6
h4 -> X X X X X
h5 -> h1 h2 h3 X h6
h6 -> h1 h2 h3 X h5
*** Results: 33% dropped (20/30 received)
```

Listing 4.5: Drop Packets `pingall` Output

All packets destined for H4 were dropped, meaning each host received no ping response. Similarly, as responses from other hosts to H4 are also dropped, H4 is unable to receive responses from any other host (however, requests from H4 are received by other hosts as the blocking is unidirectional), meaning that the `pingall` output shows no connectivity from H4 to other hosts. I only focus on destination IP addresses in this test, but it would be trivial to match a source IP address as well to prevent communication between a specific pair of hosts whilst allowing other hosts to communicate normally.

Amplified DOS (Flooding) I trigger the malicious application whenever a packet destined for H4 is seen on any switch. As the packet is replicated at every switch, the number of packets grows exponentially and very quickly the network is overloaded. To demonstrate this more clearly, I send a single ping packet from H1 to H4, and then attempt to ping from H3 to H5:

```
mininet> h1 ping -c 1 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=1013 ms

--- 10.0.0.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1013.590/1013.590/1013.590/0.000 ms


mininet> h3 ping -c 4 h5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
From 10.0.0.3 icmp_seq=4 Destination Host Unreachable

--- 10.0.0.5 ping statistics ---
4 packets transmitted, 0 received, +1 errors, 100% packet loss, time 3081ms
```

Listing 4.6: Amplified DoS Attack

As can be seen, the first ping is successful (though experiences a heavy delay). Even after the initial ping has stopped, I see a very large number of packets entering the controller. On the ping from H3 to H5 (which are both not targeted by the attack), the connection fails as the network is overloaded and the controller is dealing with a very large number of requests. As an example, I perform a packet capture on a single port on one of the switches, seen in Figure 4.4. After just 10 seconds, the packet capture contains more than 390,000 duplicated packets from a single ping (all packets have matching ids and sequence numbers), with the rate increasing as more and more packets are duplicated.

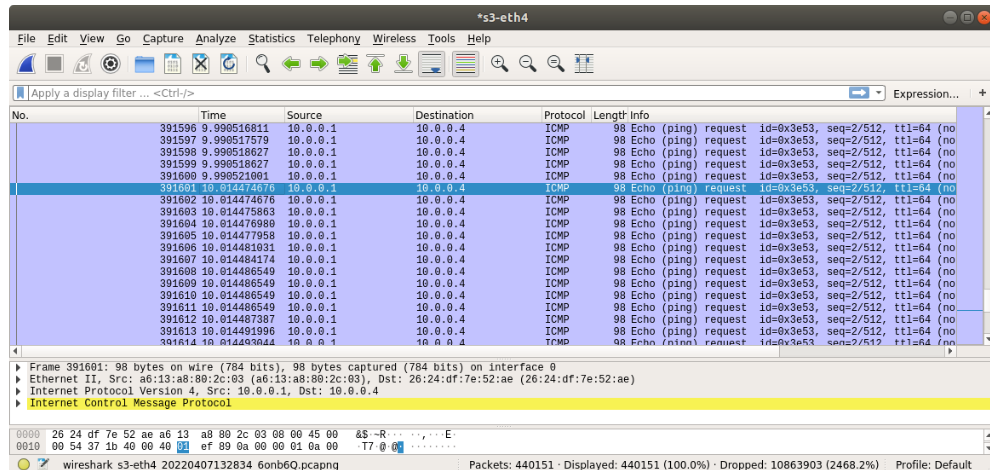


Figure 4.4: Amplified DOS Attack

Amplified DOS (Targeted) I start the targeted DOS attack against host H4. As can be seen by observing the network traffic using Wireshark configured on H4, on sending a ping from H1 to H4, for each packet sent by H1, 8 are received by H4, and H1 received 8 responses back. Whilst this is not enough to overload the host, it shows the attack is feasible. If a heavier (higher throughput) application when compared to ping is used, it is expected that H4 will be taken offline as the attack essentially emulates a distributed denial of service attack. Below is the ping output from pinging H1 to H4, after the malicious flow rules have been setup:

```
mininet> h1 ping -c 2 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.126 ms
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.186 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.190 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.194 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.197 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.200 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.203 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.206 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.197 ms
```

```
--- 10.0.0.4 ping statistics ---
2 packets transmitted, 2 received, +7 duplicates, 0% packet loss, time 1021ms
rtt min/avg/max/mdev = 0.126/0.188/0.206/0.028 ms
```

Listing 4.7: Amplified DoS Targeted Attack

The first ping request (`icmp_seq=1`), is duplicated 8 times by the network, and 8 responses are returned to the sender (which the ping utility correctly identifies as duplicates). Note that in the example 2 pings were sent, but only a single response is shown for the second ping (`icmp_seq=2`). This is because the ping tool will exit on receiving the first response to the final request, even though 8 responses are actually received. If the ping tool is only instructed to send a single packet, it will only count a single response, even though 8 are received.

4.4.2.2 Eavesdropping

Redirect Eavesdropping To demonstrate the redirect eavesdropping attack, I use a simpler topology, as seen in Figure 4.5. In this topology, H3 is the attacker controlled machine, and H1 and H2 are the communicating hosts. The malicious application, on seeing traffic to or from H1 (10.0.0.1), will add an additional action to the flow rule to output traffic to port 3, where H3 is connected. Figure 4.6 shows the Wireshark capture on the ethernet interface of H3, whilst a ping is made from H1 to H2 (10.0.0.2). The ICMP packets between H1 and H2 are also received on H3, which would normally not be able to observe these packets, even if capturing in promiscuous mode (in this case promiscuous mode is disabled).

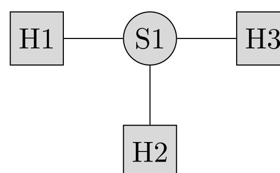


Figure 4.5: Simple network used for redirect eavesdropping and redirect traffic attacks

Figure 4.6: Redirect Eavesdropping Attack

4.4.2.3 Data Tampering

Controller Packet Tampering For a demonstration of this attack, please see Section 4.5.4.4.

Person-in-the-Middle For a demonstration of this attack, please see Section 4.5.4.5.

Traffic Redirect To demonstrate the traffic redirect attack, I again make use of the simpler topology seen in Figure 4.5. In this scenario, H1 (10.0.0.1) is the target server, H3 (10.0.0.3) is the client, and H2 is the attacker controlled machine. I run a simple web server on H1, running on port 998, using the Python 3 `http.server`. The folder in which Mininet runs contains 2 folders, `host1` and `host2`, which each contain a file called `host.txt`, which indicates on which host the web server is being run. I start a web server on H1, using the `host1` folder as the root. The `host.txt` file within this folder contains “This is host 1 (10.0.0.1)”. When not under attack, if H3 requests this file from IP address 10.0.0.1, they will receive a file from host 1. This can be seen in Figure 4.7a.

I then introduce an attacker machine, H2, which is configured to have the same IP and mac address as H1. H2 is also running a web server, also bound to IP 10.0.0.1 on port 998, but in this case serving the contents of folder `host2`. The malicious application, in seeing a packet destined for 10.0.0.1 (H1), will direct it out of port 2 where H2 is located, instead of port 1 where H1 is located. This means that, when H3 sends a request to H1, the request is instead redirected to

H2, and the contents of the `host.txt` file within the `host2` folder is returned. This can be seen in Figure 4.7b. As can be seen in the output, the returned file is that of H2.

4.4.2.4 Service Degradation

Flow mod blocking & PacketIN Eavesdropping On preventing flow rules from being installed for a target, I repeat the 10 repeated pings and take the mean packet times, excluding the first packets. The mean RTT for packets was 4.16ms (SD 0.95), which is a substantial increase on the normal case (0.105ms). Note that this attack only works in one direction, preventing flow rules from being installed for packets towards H4, but not the return path to H1, which means that the latency is a substantial increase, but does not extend to the 7.95ms mean bi-directional setup time.

Below is an example of pinging from H1 to H4 whilst H4 is the target of the attack:

```
mininet> h1 ping -c 5 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=11.9 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=4.81 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=5.88 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=3.50 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=4.35 ms

--- 10.0.0.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4009ms
rtt min/avg/max/mdev = 3.507/6.110/11.983/3.035 ms
```

Route Increase The effect of the route increase attack was first verified by monitoring packet captures on individual switches. By doing this, I confirm that the H1-H4 flow does indeed pass through the longer route. The RTT of packets increases slightly from 0.105ms (SD 0.042) to 0.114ms (SD 0.016). This is a

```
mininet> h1 cd host1/ ; python3 -m http.server --bind 10.0.0.1 998 &
mininet> h3 wget 10.0.0.1:998/host.txt -O data
--2021-07-25 05:33:51--  http://10.0.0.1:998/host.txt
Connecting to 10.0.0.1:998... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26 [text/plain]
Saving to: 'data'

data                100%[===================>]      26  --.-KB/s    in 0s

2021-07-25 05:33:51 (171 KB/s) - 'data' saved [26/26]

mininet> h3 more data
This is host 1 (10.0.0.1)
mininet>
```

(a) Request from H3 to H1 whilst not under attack

```
mininet> h1 cd host1/ ; python3 -m http.server --bind 10.0.0.1 998 &
mininet> py h2.setIP('10.0.0.1/8')
mininet> py h2.setMAC('00:00:00:00:00:01')
mininet> h2 cd host2/ ; python3 -m http.server --bind 10.0.0.1 998 &
mininet> h3 wget 10.0.0.1:998/host.txt -O data
--2021-07-25 05:36:23--  http://10.0.0.1:998/host.txt
Connecting to 10.0.0.1:998... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26 [text/plain]
Saving to: 'data'

data                100%[===================>]      26  --.-KB/s    in 0.001s

2021-07-25 05:36:23 (45.5 KB/s) - 'data' saved [26/26]

mininet> h3 more data
This is host 2 (10.0.0.2)
mininet>
```

(b) Request from H3 to H1 when under attack

Figure 4.7: Redirect traffic attack

smaller increase than expected, which I believe is down to the efficient nature of Mininet, in which all links feature minimal latency. To verify this, I ran a simple test of the packet RTT between H1 and H2 (a 3 hop path) and H1 and H3 (a 4 hop path). These achieve mean packet RTT of 0.089ms and 0.1 ms respectively, a very small increase. Whilst the attack has a minimal impact in this test scenario, in a real-world setting where inherently slower links exist the effect of this attack can become more noticeable.

4.4.3 Discussion

As can be seen in Section 4.4.2, I have been able to successfully demonstrate almost all of of the proposed attacks within a simple network environment, with

the remainder demonstrated against an industrial control system environment later.

For denial-of-service type attacks, the effect of these can be clearly seen. For example, the controller crash and flow mod blocking attacks clearly prevent communication from occurring. Whilst the controller crash and amplified DOS attacks would impact the whole network, the targeted amplified DOS and drop packets attacks can be targeted to particular hosts whilst leaving the rest of the network unaffected.

The service degradation attacks also have a clear effect on the network performance. The impact of these attack very much depends on the type of traffic being affected. For simple web browsing type traffic, the noticeable impact of these attacks would be minimal. However, in scenarios that rely on maintaining low latency communication, for example video conferencing, these attacks could have a major impact. Of particular concern are real-time systems, which I explore in the context of industrial control systems in Section 4.5. The differing attacks have different levels of detectability. The flow mod blocking attack is going to create a very noticeable change in the switch-controller traffic, as the switch now has to forward every packet to the controller. On the other hand, the route increase attack is set up as a normal flow would be, just with a longer path, which would be harder to identify.

Whilst the PacketIN eavesdropping attack has a noticeable impact on the underlying traffic, the redirect eavesdropping attack allows the flows to be routed as normal, with only the addition of the mirror ports. This can allow the attacker to capture traffic that they would otherwise not be able to. This attack could also be performed by compromising the switch, and if it is supported, configuring a mirror port.

Finally, the data tampering attacks represent the most interesting set of attacks that really show the potential impact om SDN. The traffic redirect attack in particular could be extremely useful to an attacker in facilitating further attacks such as phishing. Whilst these types of attacks could be achieved using more traditional means such as ARP spoofing, these are easy to identify using intrusion detection systems to the the more active nature of them. The SDN-based attacks, on the other hand, do not require the attacker controller machine to send

any additional packets, only to receive and forward traffic sent to them, which is far less obvious.

Some of these attacks could be detected/prevented by existing systems designed for preventing malicious controllers, such as [6, 7, 111, 185, 195]. This particularly applies to attacks which terminate communication, such as the sinkholing DoS attack, or the Amplified DoS attack, which introduces network loops. As these type of systems usually apply some kind of detection function to identify attacks occurring, then it is not possible to be able to identify all malicious controller responses with 100% accuracy if a single controller is in use, in particular when the more subtle attacks, such as route increase, are used, as very similar effects could be caused by legitimate application such as load balancers. A consensus approach, such as SDBFT, will be able to identify any of these attacks occurring as long as there is a single correct controller, as all of these attacks will result in a controller response different to the normal, non-malicious case.

Difficulty in Performing Attacks Within this simple scenario, these attacks do not require a large amount of skill to perform. The most difficult aspect is reverse engineering the Forwarding application to identify the normal flow of operation and the locations where this application can be modified to achieve the desired goal. Within this testing, the malicious applications are hard coded to perform attack actions, such as forwarding packets to specific destination addresses out of specific ports. This itself represented a manual process to implement the logic within the malicious application, requiring knowledge of the ports to which hosts are connected. A particular example of this is in the `RouteIncrease` application, which requires almost 300 lines of additional code consisting of conditionals to handle the target flow at each switch on the path. Rather than implement this type of attack through a malicious application, it could be possible to instead modify the datastore used by Floodlight, which stores connection information for devices and switches, to cause the Floodlight topology manager to return desired routes for target flows.

Similarly, some of the described attacks require the attacker to know the topology of the network, to the level of which ports on switches connect to which

devices or other switches. Again, this information is stored by Floodlight (assuming the network has been operational for some time and all device locations are known), and so an attacker who has access to the Floodlight instance could extract this information.

4.5 Real World Impact — Industrial Control Systems

Our previous attack demonstrations show the impact of attacks happening within a simulated network environment featuring simple traffic. Whilst that can clearly show the base impact of the attacks, it is not a real-world application. In this section, I use a case study of Industrial Control Systems (ICS) as an example of a real-world environment where SDN use is proposed, and attacks could have a major impact. In particular, within an ICS setting real-time protocols are used extensively, and so I can demonstrate the impact of some simple SDN based attacks on such protocols in a lab environment consisting of physical industrial devices interacting with a simulated physical process. This work has been published as "Controller-in-the-Middle: Attacks on Software Defined Networks in Industrial Control Systems" [87].

4.5.1 Industrial Control Systems

ICS, as the name suggests, are the systems that control industrial processes. Typical examples of these types of processes include manufacturing, power generation and distribution and water treatment. Many elements of critical national infrastructure (CNI) involve the use of industrial control systems as key components [36]. As ICS interacts with real-world, physical processes, it is an example of a cyber-physical system (CPS). Operational technology (OT), as opposed to information technology (IT), refers to the devices and software that makes up an industrial control system. Whilst there is some overlap between OT and IT systems, generally they will be handled by different teams within an organisation,

with OT largely maintained by engineers, and IT systems by more traditional IT professionals.

In a traditional IT system, the *confidentiality, integrity and availability* triad of attack impact is largely applicable in that order i.e. the confidentiality and correctness of data is prioritised over its availability. In an OT system, a 4th concept is introduced in *safety*, which is given the highest priority, very closely followed by *availability* [124, 73]. The safety aspect refers to the fact that an ICS often controls large, physical process in dangerous environments. If the control system were to malfunction, either through error or malicious action, it could potentially cause serious injury, or even death, to those in the vicinity of the process. Availability has both a financial impact, as a process that is not running does not generate revenue (e.g. in manufacturing), and also the knock on effect of a services process being disrupted, e.g. if a water treatment plant or power generator goes offline, then it could potentially affect thousands of people who rely on those services.

To demonstrate a standard architecture for an industrial control system, I will use the Purdue Enterprise Reference Architecture (more commonly known as the *Purdue Model*), which is commonly used as the basis for industrial control systems architectures [237, 246]. A representation of the Purdue model can be seen in Figure 4.8.

Levels 0 to 3 represent the OT environment. At the lowest level, level 0, there is the physical process, made up of a number of devices such as sensors, actuators, drives and robots. These are connected to the process control devices, found in level 1. This connection is often physical wiring, or sometimes wireless protocols such as WirelessHART. The devices in level 1 are the key specialist devices within ICS, consisting of devices such as programmable logic controllers (PLCs), remote telemetry units (RTUs) and distributed IO devices. Common vendors for such devices include Siemens, Allen Bradley (Rockwell Automation), Honeywell, Delta, Schneider and General Electric. In the simplest ICS, a PLC runs a program (referred to as the "logic"), which reads inputs from the sensors and sets the state of the drives and actuators. It is usually physically wired to these inputs and outputs. A distributed IO device can reduce the amount of physical wiring required by providing a local device which communicates with

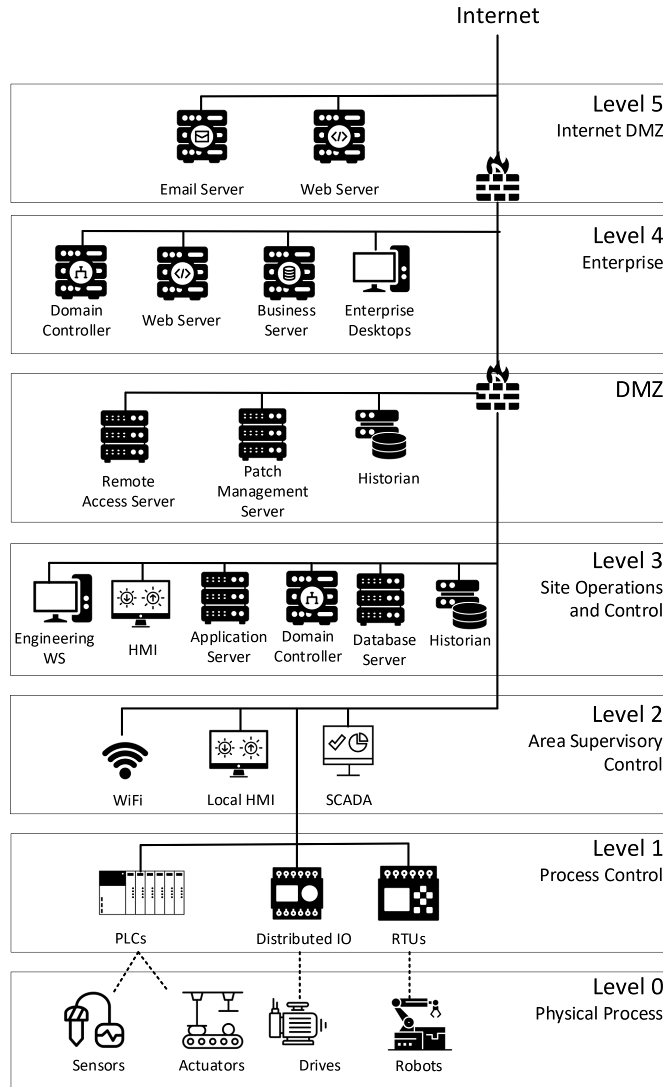


Figure 4.8: Purdue Reference Architecture [237, 246]

a secondary controller over either a serial or ethernet connection. ICS devices are designed to last a number of years and operate with close to zero downtime, however in terms of security they are often lacking even basic security controls by default and suffer from many simple yet critical vulnerabilities, which often go unpatched [227, 228].

The level 1 devices can communicate with local area supervisory control systems in level 2, commonly referred to as Supervisory Control and Data Acquisi-

tion (SCADA) systems. These provide human-machine interface screens within a control centre, allowing operators to interact with and monitor the process.

Level 3 covers the site wide operations and control, including engineering workstations, HMIs, application servers, data historians and domain controllers.

Devices in level 1 use a number of ICS-specific protocols for communicating with each other, and level 2 and 3 services. Some of these are vendor specific and proprietary, such as Siemens S7Comm, whilst others such as Ethernet/IP and Modbus are open standards used by multiple vendors.

Between the OT and enterprise environments is the industrial demilitarised zone (**DMZ**), which is accessible to both the enterprise and OT networks, but provides a barrier between the two.

Finally **Levels 4 and 5** represent the enterprise network of the organisation. In level 4 there are the core enterprise systems, including domain controllers, business servers, internal web servers and enterprise PCs. Level 5 is the internet DMZ, where the publicly accessible servers are located, such as email and web servers.

Communication from levels 1 upwards is nowadays provided through IP networks. Traditionally, for security purposes, there would be a physical air-gap between Levels 3 and 4 with no connection between the two, however this is often no longer the case, especially with the increased use of the cloud and the industrial internet-of-things with the shift to "Industry 4.0" [191].

OT networks are relatively static when compared to enterprise IT networks. It is not common to add new devices, and devices will stay connected for long periods of time, with predictable traffic patterns. For example, a data historian will consistently read from a PLC once every second. Within OT networks it is very common to use security appliances such as firewalls and intrusion detection systems. Firewalls, in particular, are an important tool in preventing unauthorised access to devices, with only specific servers and workstations able to communicate with devices and network segmentation being a key security mechanism. This lends well to an SDN deployment — the security functions of the network can be built into the controllers, and new devices added to the network can be handled appropriately. In particular, an ICS-SDN deployment would most likely

heavily relay on proactive flow rule configuration rather than reactive due to the static nature of the underlying network

Proposed uses of SDN in ICS There have been a number of proposed uses of SDN in an ICS environment. As well as utilising SDN to help reduce network management overhead, SDN is used as a tool to enable dynamic defences against attack [48]. I discuss a selection of these here.

Silva et al. propose a multipath routing mechanism built using SDN as a method for mitigating eavesdropping attacks [217] in ICS networks. In this approach, shortest paths are computed between pairs of devices (using Dijkstra's algorithm [67]), and chosen. After a short timeout, the cost of the used path is increased, and the shortest path is recalculated, with the new shortest path is then used. This means that the flow changes path frequently making it harder to eavesdrop on a flow for a continuous period.

Another use for SDN as a security mechanism is as a network intrusion detection system (NIDS). Silva et al. propose a one-class NIDS in which the SDN controller collects snapshots of Openflow statistics from switches which are sent to a data historian and then used to detect attacks, relying on the generally static nature of ICS networks [218].

Derhab et al. propose an SDN-WAN based architecture for IDS, which migrates the control layer to the cloud [66], along with a intrusion detection system to detect forged commands to ICS devices, and a blockchain-based integrity checking system for identifying attacks which modify switch flow rules.

One particular use case that has been proposed for SDN in an ICS environment is within smart grids. Rehmani et al, provide a details survey of SDN use within smart grids, including the security and privacy scheme within such architectures [193].

4.5.2 Attacker

ICS networks are generally not the targets of low-level attackers, such as script kiddies, due to the highly sensitive nature of the targets. As ICS often make up aspects of critical national infrastructure, then there is a much higher risk

in attacking such systems, for relatively little financial gain. Rather, ICS are generally the target of far more highly motivated attackers, including nation states, cyber-terrorists and organised crime who are trying to cause widespread disruption to services, or extort for financial gain.

Attacker Goals The overall goal of the attacker is usually going to be one of the following two scenarios. First the attackers overall goal is to cause disruption to the physical process to cause physical damage or disruptions. The well known example of this is Stuxnet, which was a worm, believed to be a joint effort between the US and Israel, that targeted the Iranian nuclear program. Stuxnet targeted the centrifuges used to enrich uranium. By targeting the controllers, the worm was able to rapidly speed up and slow down the centrifuges, causing them to become damaged, disrupting the Iranian nuclear program. It is believed that up to one fifth of Iran's nuclear centrifuges were destroyed by the attack [110]. This form of attack has largely non-financial motivations, and will be the goal of nation state attacker and cyber terrorists. A similar example of a successful, yet mitigated, attack on an ICS system occurred in 2021 where a hacker attacked a Florida water treatment facility and increase the amount of sodium hydroxide in the treated water to toxic levels [18]. Fortunately, this attack was detected by an operator and no unsafe drinking water escaped the plant.

The other potential attack goal is to cause the threat of disruption to the physical process, and require ransoms to be paid to recover the process/prevent attack. As well as the increasing threat of ransomware, of which ICS are becoming an increasing target [180], more targeted attacks where an attacker shuts down a process and refuses to allow it to recover until a ransom has been paid could occur. Though this hasn't occurred on OT devices as of the time of writing, it inevitably will. Ransomware has had an impact, however. In 2021, the US Colonial Pipeline in the US, which supplies fuel to the east coast [17], was affected by a ransomware attack affecting its billing systems. Whilst the attack did not directly affect the OT networks, it shut down operations on the pipeline as billing could not be carried out. This type of attack is more likely to be attributed to organised crime gangs or hacker groups than nation state level attackers, as traditional ransomware is an attack for financial rather than political gain.

Many ICS focussed protocols are real time in nature, and require minimal latency. An attack launched by a malicious SDN controller could easily introduce latency into the communication, enough to break real-time properties. As an example, the IEC 61850 standard for power systems specifies that the latency for fault isolation and protection services shall not exceed 3ms [92]. As I have shown in Section 4.4.2.4 above, and demonstrate against the real-time Profinet protocol below, achieving this additional latency is feasible even with a single switch.

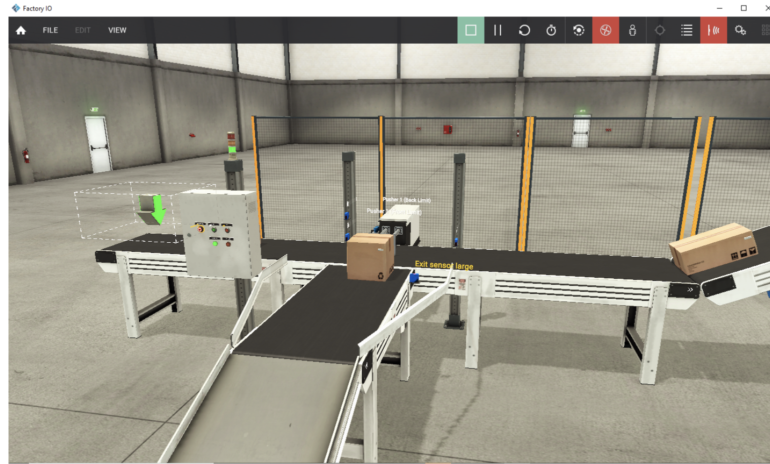
The compromise of the SDN controller can both be used to directly launch attacks, as well as to assist in performing traditional host-based attacks within the network. As the attacked gains a large amount of control over the routing of the network, as well as potentially other networking functions such as firewalls, gaining control over the SDN controller could become a key target for an attacker, in particular for facilitating further attacks.

4.5.3 Setup

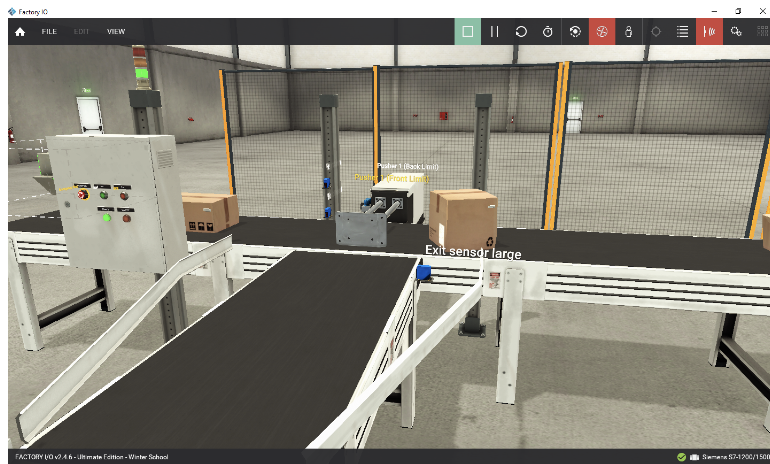
To measure the impact of SDN-based attacks within an ICS I have created a small-scale testbed, consisting of real-world, physical PLCs controlling a virtual factory process. This small-scale testbed represents Layers 0 and 1 of the Purdue model, as found in Figure 4.8.

4.5.3.1 ***Physical*** Process

In order to measure the effect on a physical process, I use FactoryIO from Real-Games [192]. FactoryIO is designed to be used for learning PLC programming, with the ability to build large scale factory simulations which are controlled by real-world devices. The software talks to PLCs using an Ethernet connection and is able to utilise a number of industrial protocols, including S7Comm, Ethernet/IP and Modbus. This allows me to easily connect it up to different devices over a SDN network and measure the impact of the attacks on the different protocols. The scene I test with is a simple sorting scene, in which small and large boxes are moved down a conveyor belt and measured. The larger, taller boxes are



(a) FactoryIO scene used for testing. The larger, cube-shaped boxes are pushed off the main conveyor, the rectangular boxes are not



(b) FactoryIO when under the flow rule blocking attack

Figure 4.9: FactoryIO

pushed onto a secondary conveyor, whilst the shorter boxes are not. An example of this scene can be seen in Figure 4.9a.

4.5.3.2 SDN

Networking is provided by a Dell EMC PowerSwitch S3048-ON 1000BASE-T 48-port 1GbE top-of-rack (ToR) switch, which features support for SDN using OpenFlow (versions 1.0 and 1.3), and can operate with 3rd party controllers

and operating systems (the *ON* portion of the model number represents *Open Networking*). The switch is running Dell EMC Networking OS9 (specifically 9.13), and has been configured to use Openflow 1.0. I use the Floodlight controller [187] to provide network control.

4.5.3.3 Devices and Protocols

I make use of four common ICS protocols in this testing:

Profinet - Profinet is a real-time protocol commonly used by Siemens devices. In particular, it is the protocol in use for providing communication between Siemens PLCs and HMIs, as well as when using remote IO . I make use of two S7-1200 PLCs running firmware version 4.2. One operates as the controller, and the second as the remote IO device. The laptop running FactoryIO is connected to the remote IO device over Ethernet, using S7Comm to communicate. The SDN switch sits between the two PLCs for these tests.

S7Comm - The S7Comm protocol is the primary protocol used for workstations and SCADA systems to interact with Siemens PLCs. As well as downloading programs to the device, it can be used to read and write memory addresses to the device. As an example, it is common for software such as data historians to use S7Comm to read values from devices. S7Comm is unencrypted, though newer devices use S7CommPlus which do use encryption (though this has been shown to be insecure [26]). S7Comm is used between FactoryIO and the remote IO PLC, with the SDN switch moved between the two. The connection between the two PLCs is instead through a standard Mikrotik switch.

Ethernet/IP - Ethernet/IP is an open protocol, most commonly used by Allen Bradley devices for communication, including remote IO, although unlike Profinet it is not a real-time protocol. I use an Allen Bradley Micro850 PLC, connected to FactoryIO over the SDN switch.

Modbus/TCP - Finally, I use the common Modbus/TCP protocol. I use OpenPLC[1] installed on a Raspberry Pi 4 Model B (4Gb Ram). OpenPLC is an open source PLC commonly used for research projects using the Modbus

[1] `https://www.openplcproject.com`

protocol [9]. To use FactoryIO with OpenPLC, FactoryIO runs a Modbus server, which OpenPLC treats as a slave device. The SDN switch sits between OpenPLC and FactoryIO.

(a) Siemens S7-1200 PLC

(b) Siemens S7-1200 PLC and HMI in box

(c) Allen Bradley MicroLogix 850 and HMI in box

Figure 4.10: ICS devices

The topologies for each protocol are shown in Figure 4.11. In the diagrams, the labels on lines indicate the protocols in use. Note that the SDN controller is not shown, however it running on a blade server directly connected to the switches management port. Where two devices are not shown to be connected using a switch, they are connected through a direct Ethernet connection.

4.5.4 Attacks

4.5.4.1 Flow Rule Blocking

I apply the flow mod blocking attack to the ICS environment. This attack should cause a large amount of additional latency, which should have a major impact on the real-time Profinet protocol, and a noticeable impact on the three non real-time protocols. For the non real-time protocols, I expect this latency will introduce delays into the operation of the physical process.

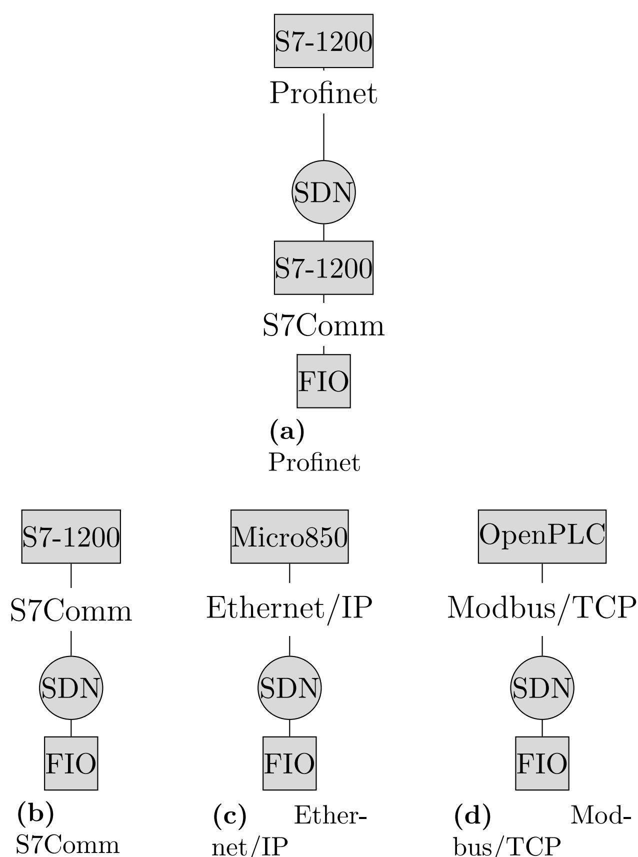


Figure 4.11: Topologies used for testing. SDN = SDN Switch. FIO = FactoryIO host.

Demonstration I apply this attack to all four protocols in the test setup. The attack is targeted to the PLC device; any other flows will be unaffected. On first seeing a request to the target device, the malicious application will allow any handshakes to complete, and after 20 second clear the flow table of the switch and then apply the attack. I apply flow rule blocking with no additional delay introduced by the controller (except the delay of contacting the controller itself, which is 3-4ms), and also with artificial additional delay on sending responses from the controller to measure how much extra latency is enough to cause the physical process to no longer sort blocks.

I presents the results of this attack on the four protocols in Table 4.1. As can be seen in the third column, all of the protocols, other that Profinet, were able to operate whilst under this attack. Profinet, the only real-time protocol, quickly

Protocol	Real-time?	Operates with blocking?	Physical process affected?	Additional delay for failure
Profinet	Yes	No	Yes	-
S7Comm	No	Yes	No	40ms
Ethernet/IP	No	Yes	Yes	10ms
Modbus/TCP	No	Yes	Yes	10ms

Table 4.1: Results of flow-rule blocking attack.

raises an alarm once the attack begins, triggering an error state on the PLC. On inspecting the requests to the controller, Profinet messages are sent at a sufficient rate that the controller is unable to process requests and forward packets quick enough and a backlog forms. This, along with the additional latency, breaks the real-time properties of the protocol and causes Profinet to fail.

For those protocols that still managed to operate when under attack, only S7Comm showed no obvious impact on the physical process. Both Ethernet/IP and Modbus/TCP, when faced with the additional latency of passing packets through the SDN controller, both exhibited a noticeable lag in the pusher operation in the physical process, pushing blocks late and in some cases late enough that the block remain on the main conveyor. This clearly shows that there is a potential safety impact from this attack, as the reaction time of the process for any aspects which rely on this network communication are impaired.

Finally, I measure how much additional latency is required to cause the pusher to completely miss blocks, preventing sorting. S7Comm requires 40ms of additional delay, whilst Ethernet/IP and Modbus/TCP both only required 10ms additional controller delay to fail. This small value indicates that both Ethernet/IP and Modbus/TCP could be vulnerable to other SDN attacks that increase latency, for example through increasing path lengths. This effect can be seen in Figure 4.9b.

4.5.4.2 PacketIN Eavesdropping

The PacketIn eavesdropping attack could be applied to an ICS environment. For an attacker there are two obvious benefits to this. First, it is useful in asset discovery, as it reveals IPs, MAC addresses and protocols in use to the controller. Further, in some protocols, such as S7Comm, device specific details such as model

numbers and firmware versions are sent as part of the protocol which can also aid in asset discovery. The second benefit is that if the protocol in operation can be observed, then the attacker can potentially learn about the behaviour of the underlying process by observing transmitted data. For example, Ethernet/IP reads and writes each individual register, including the register name, as individual requests. If these are used for remote IO or a data historian, an attacker could monitor these over a period of time to learn which registers could be tampered with to affect the process.

4.5.4.3 Redirect Eavesdropping

I apply the redirect eavesdropping attack to the physical topology. Whilst this attack performs no differently to the example shown in the simulated environment, this does show the attack also works on a physical topology with clearly separated devices.

Demonstration Our malicious forwarding application, on creating a flow rule, adds an additional action field to output packets to the port where the attack laptop is connected, on which I run Wireshark. As soon as a new target flow is setup, then Wireshark will start capturing all traffic on the flow. S7Comm packets only appear into the network capture only begins when the attack is started, as the laptop does not usually have visibility onto those packets, even when running in promiscuous mode.

It is important to note that for established flows, the simple version of this attack will not work as the controller will not be contacted to install the flow rule for the target device. The controller will have to direct the switch to delete the existing flow rule, and then install an updated flow rule proactively. The controller can do this in one command, which will prevent any disruption to the existing network.

4.5.4.4 Controller Packet Tampering

I apply the controller packet tampering attack to modify the packets of the industrial protocol. In cases where the full packet is sent to the controller, and the

protocol is unencrypted (as S7Comm, Ethernet/IP and Modbus/TCP all are by default) I can make arbitrary modifications to the packets to perform a person-in-the-middle attack. The advantage of this attack is that no new routes are created — the switch already communicates with, and forwards packets to the controller. The primary effect on the network is a large increase in the volume of packets sent from the switch to controller. In this example, I wish to overwrite the values relating to the pusher in the example process, to prevent blocks from being pushed off the main conveyor. Packets are modified by modifying the packet data when the `PacketOUT` message is created by the malicious forwarding application.Note that I cannot apply this to the Profinet connection, as the connection fails if packets are routed through the controller due to the additional latency.

One small issue arose when developing this attack. If the data is simply changed, the packets TCP checksum is then incorrect and FactoryIO disconnects. This means that the malicious application needs to deconstruct the TCP layer of the packet, modify the TCP payload, and then rebuild the TCP header. This adds a very small amount of additional overhead to the packet processing, and hence latency to the connection.

Demonstration For S7Comm, FactoryIO uses a ReadVar S7Comm packet to read the state of the output addresses on the PLC. Within the response, the final 2 bytes of the packet, containing the read values, are returned, which are `B100` when the pusher is not being pushed, and `B900` when the pusher is active. Therefore, for any packets from the target device which contain `B900` as the final 2 bytes, these are replaced by `B100` successfully preventing the pusher from operating.

In the Modbus/TCP setup, OpenPLC updates the coil state to FactoryIO using the `Write Multiple Coils` request type, sending 2 bytes of data over. When the pusher is inactive, these 2 bytes are `B300`, changing to `BB00` when the attack is active. Similarly to S7Comm, these are the final 2 bytes of the packet. Again, the attack successfully blocks the pusher from operating.

The attack is a little more complicated in the case of Ethernet/IP. Whereas S7Comm and Modbus/TCP both request/write all of the output registers in a

single request, within Ethernet/IP each individual output register is requested individually. This means that I need to observe the request from FactoryIO to the PLC for the register corresponding to the pusher (in this case `BOOL_OUT_3`), and then only modify the following response, replacing the final 3 bytes of the packet — `C10001` with `C10000'`. As with this other two protocols the attack is successful.

In this demonstration I prevent the pusher from pushing, however the same attack could be applied to, for example, overwrite values being sent to a SCADA server, potentially preventing alarms from being raised.

4.5.4.5 Person-in-The-Middle

In this implementation of the person-in-the-middle attack, the attacker wishes to modify the packets sent between the PLC and FactoryIO (the physical process) in order to make the process go wrong.

Demonstration I demonstrate this attack using FactoryIO with a Siemens S7-1200 PLC using S7Comm. I introduce a third device into the network, a laptop running Ubuntu 18.04. This laptop simulates a host within the network that is under control of the attacker, for example a host they were able to successfully compromise. The laptop has two Ethernet connections to the switch, and on the laptop a virtual network switch is deployed using OpenVSwitch (OVS), with both physical adapters added as ports to the virtual switch. The OVS switch is controlled by its own Floodlight instance, which simply directs packets out of the adapter which the did not come in on, effectively making the virtual switch a proxy. The OVS controller instance also has the controller packet tampering application installed, and so will intercept all target packets and modify them in the same way as the previous attack, however this behaviour is contained to the attacker laptop. Note that using the controller packet tampering approach within the attacker machine does introduce a small amount of additional latency, and is only used as a proof of concept. With greater effort the attacker could use an alternate approach, such as through use of a proxy, to remove this additional

latency. On the core SDN switch, flow rules are manually installed through a malicious application to direct target flows through the laptop.

Through this attack, I was successfully able to perform a person-in-the-middle attack without the use of traditional techniques such as ARP spoofing. Within the data plane there are no unusual packets, only flows taking an unusual route. As with the controller packet tampering attack, as well as interfere with the physical process this attack could be used to modify packets sent back to SCADA systems to prevent monitoring and alarms.

4.5.5 Discussion

The aim of these demonstrations were to show the impact of some of the proposed attacks within a real-world setting, in particular the setting of an industrial control systems environment. I show the impact on one real time, and three non real-time (though used as real-time protocols in some scenarios) protocols, and also demonstrate the attacks agains a commercial, physical SDN switch.

As I have shown, the timing related attacks are able to completely break the real-time properties of the Profinet communication. If used for distributed IO, as is the case here, this could have a devastating impact on the operation of the physical process as all control would be lost, with only separate safety systems remaining to ensure the process returns to a safe state. Even in the non-real time protocols, adding only small amounts of additional latency (<10ms) allows the protocols to continue operating, but provides a delay to the physical process which causes failures in the normal operation of the process. Whilst Ethernet/IP and Modbus/TCP are not real-time protocols, they are used for distributed IO and so latency should be kept to a minimum.

The two tampering attacks also demonstrate a clear serious safety issue within the industrial control system. In particular, the person in the middle attack could allow an attacker to intercept packets with minimal additional latency. As I demonstrate, this could be used to interfere with IO packets to cause disruption to the physical process. It could also similarly be used to modify data sent to historians and SCADA control systems, which could cause human operators

within control centres to see incorrect information, which is again a major safety concern as operators may take incorrect actions.

Difficulty in Performing Attacks The core routing modifications of these attacks is no more difficult than the examples provided in Section 4.4.2, as the underlying network is similar. The attacks could be made more difficult if non-SDN firewalls and access control were used, which could potentially require the attacker to have to be more careful in the routes they configure. Usually, ICS networks have much tighter firewalls and access control than enterprise and other networks, which the attacker may have to work around. Of course, if these network functions were performed by the SDN controller, then this would be far less of an issue.

4.6 Conclusion

In this chapter I aimed to provide an overview of the attacker who would compromise an SDN controller, and demonstrate the attacks that such an attacker could perform.

I began by modelling the type of attacker, including their goals and attack vector. I envision that these attacks would be restricted to well-resourced attackers, such as nation state or organised crime gangs who are well resource, highly skilled and well motivated. Their goals can include both directly interfering with the operation of the underlying network to perform attacks such as denial-of-service, but also to facilitate traditional, host-based attacks such as person-in-the-middle attacks which would usually require techniques such as ARP spoofing.

I then propose a number of attacks, some of which are new and some of which are taken from the literature, and demonstrate a number of these within a simulated network environment. I then go on to demonstrate a subset of these attacks within a real-world setting, namely that of industrial control systems, where in particular I show the impact of attack on real-time protocols, and demonstrate how these relatively simple attacks can have a major safety impact on a physical process.

To the best of my knowledge, this represents one of the most comprehensive demonstrations of attacks that can be launched using a compromised SDN controller, and the first such exploration of these attacks within an industrial control systems setting.

In the next chapter, I propose a system, Software-Defined BFT (SDBFT), which is able to prevent all of the described attacks from occurring.

Chapter 5

Designing An Efficient Consensus Approach for SDN Control

5.1 Introduction

In the previous chapter, I discussed and demonstrated the potential impact from a compromised SDN controller. I showed that even simple malicious updates from a compromised controller can have severe impacts on the operation of the network, which could cause major issues, in particular in real-time systems. In this chapter I propose a protocol, Software-Defined Byzantine Fault Tolerant control (SDBFT), for providing fault handling in a distributed SDN controller architecture, which is able to prevent a compromised SDN controller from pushing malicious updates to switches.

5.2 System Overview

In the traditional SDN model, as seen in Figure 5.1, an SDN switch is controlled by a single controller. The switch is connected to a single controller, and sends requests to the controller, usually in the form of `PacketIn` messages, and receives a response from that controller, usually consisting of `PacketOut` and `FlowMod`

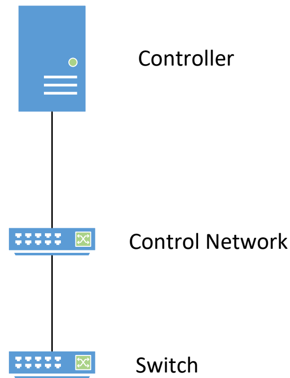


Figure 5.1: Traditional controller architecture

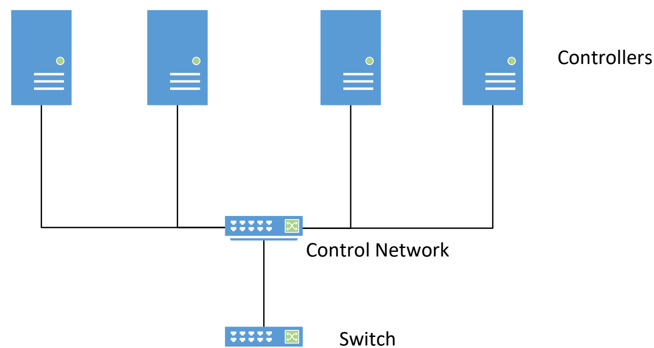


Figure 5.2: SDBFT controller architecture

messages. The connection from the switch to the controller is either direct, or through a network. The control network can either be separated from the underlying SDN-controlled data plane, or could in some cases be provided over the controlled data plane. I assume for this work that the switch to controller network is isolated from the SDN controlled data plane provided by the switches. Whilst multiple controllers may exist within this environment to provide scalability and redundancy in the case of controller failure, at any one point in time the switch is only controlled, and receives instruction from, a single controller. This is a clear single point of failure, either for genuine controller fault, or malicious

compromise.

I propose Software-Defined Byzantine Fault prevenTing control (SDBFT), an architecture in which the switch contacts multiple controllers simultaneously, and applies a simple fault-preventing (fault detecting with recovery) protocol to prevent compromised controllers from installing malicious flow rules onto switches. Figure 5.2 provides an overview of the SDBFT controller architecture. In this design, a switch has an additional component, the response processor, which collects the responses from the multiple controllers and chooses a command (flow rule, packet out etc) to be followed by the switch. If a faulty or compromised controller is detected, then a fault recovery protocol is applied to bring in further controllers from a backup pool and apply a majority vote in order to prevent the malicious flow rules from being installed. Previous work where a full byzantine fault-tolerant protocol is applied [136] is also able to provide fault tolerance against malicious controllers, although has the downside of requiring $3f+1$ controllers to handle f faults, and requires multiple rounds of communication (typically 5 as is the case with PBFT [41]), which increases the amount of time to reach consensus. This extra communication also utilises a large amount of bandwidth on the control network, which has an impact on scalability. The key difference between SDBFT and traditional BFT algorithms is the relaxing of the definition of *fault tolerance* to *fault detection, with recovery*, which allows me to, in the case without fault (which should represent the default situation), require $2f+1$ controllers, utilising just $f+1$ controllers, and a just two rounds of communication under normal operation, with the addition of f further backup controllers, and a additional two rounds of communication when a fault has been detected, reverting to two rounds of communication for subsequent switch requests. When acting as a backup, a controller performs minimal processing relating to the switch, and so only a small amount of controller capacity needs to be reserved to act as a backup if required.

5.2.1 Requirements

I define a set of four requirements for the SDBFT protocol:

R1: Low latency The additional actions required to apply quorums to the controller architecture should introduce little extra latency to responses to switch requests.

R2: Fault-preventing The protocol should be able to prevent both arbitrary byzantine failures, as well as targeted malicious insiders.

R3: Consistency As the system relies on properly replicated controllers, it is important to ensure that there is consistency in the states of the different controllers, as well as the back end datastore(s). Consistency is an important requirement in any distributed SDN control architecture [16].

R4: Scalable The protocol should provide scalability and be able to support large numbers of controllers and switches as the size of the network (and network load) grows.

There are further optional requirements that, whilst not required for operation of the protocol, can be met with simple additions to the protocol in order to provide stronger security guarantees, with a cost to performance:

R5: Non-Repudiation Message senders should be verifiable in order identify which controllers send malicious updates to switches. Similarly, non-authorised controllers should not be able to send updates to switches by spoofing authorised controllers.

R6: Message Integrity Messages should not be modifiable on the wire, for example through the use of man-in-the-middle attacks.

5.2.1.1 Message Ordering

A key requirement of traditional BFT approaches (as described in Section 2.11.2) is the message ordering requirement, which dictates that server replicas should process messages in the same order in order to guarantee consistency. This message ordering is responsible for a large portion of the required communication steps within these protocols, which introduces both communication overhead and additional latency. This ordering ensures that the state of the replicas is consistent, as requests to the replicas can modify the state of each replica, and hence

the responses they generate. As an example, a data store is commonly used as an example use case for BFT algorithms — if reads and writes are not synchronised then subsequent reads and writes across replicas may return different results.

The SDBFT protocol does not enforce message ordering within the protocol. For many core SDN applications, including routing, the operation of these applications is deterministic. For example, the routing applications in the Floodlight controller (`Forwarding` and `LearningSwitch`), both apply the deterministic Dijkstra‘s algorithm to the current view of the network topology held by the controller. Therefore, two SDN controllers holding the same view of the network should return the same result to a request from the switch. This can hold for other applications such as Firewalls. As long as the knowledge maintained by the controller is up to date, then requests from the switch do not update the state of the controller. When a new device is added to the network this knowledge is included into the controller's view, and so as soon as a device sends its first packet in the network then the controller gains knowledge of its location within the network and adds it to the view it maintains. This mode of operation means I can relax the ordering requirement for the SDBFT protocol to reduce traffic overhead, on the assumption that only deterministic controller applications are used, and that there is a process for propagating network view updates across controllers.

5.2.2 Notation

I define the set of all switches $S = \{S_1, S_2...S_m\}$ where m is the number of switches. I also define the set of all controllers $C = \{C_1, C_2...C_l\}$ where l is the number of controllers. Each controller is a replicated state machine, where I assume that in the correct state each controller on receiving the same event notification from a switch will produce the same response.

I use the term *quorum* to define the set of controller that are controlling one particular switch. Within a network, there will be a number of possible quorums. I define the set of quorums $Q = \{Q_1, Q_2...Q_q\}$ where q is the total number of quorums, and $Q_i \subseteq C$ where $0 \leq i \leq q$. The size of the quorums $|Q_i| = k$ is a

Table 5.1: Notation

Symbol	Description
C	Set of all controllers
$F \subset C$	Set of faulty controllers
$CC \subset C$	Set of non-faulty controllers
S	Set of switches
l	Number of controllers
m	Number of switches
k	Primary quorum size
b	Backup quorum size
f	Number of faulty controllers
Q	Set of possible quorums
q	Number of possible quorums
PQ_{S_i}	Primary quorum for switch S_i
BQ_{S_i}	Backup quorum for switch S_i
EQ_{S_i}	Extended quorum for switch S_i
H	Hash function
rq	Request from switch
R	Multiset of responses for a given request rq
$r_c \in R$	Response for controller c for given request rq

system parameter, chosen to provide the required level of fault tolerance. This is discussed further in Section 5.3.1.

A full overview of all used notation is available in Table 5.1.

5.3 Quorums

I use the term *quorum* to refer to a set of controllers that are responsible for the operation of an individual switch. As mentioned, a switch will communicate with a set of controllers rather than a single controller, as is the case in the traditional model. The formation of quorums is dictated by the controller assignment method. I discuss some approaches to this problem in Section 5.6 below.

5.3.1 Quorum Size

A limitation of byzantine fault tolerant agreement systems from the literature is the requirements for the number of replicated processes that are required to

handle a pre-defined number of faults, f. The best that can be achieved while maintaining the fault tolerance guarantee is $n \geq 2f + 1$. Typically, BFT approaches, including PBFT requires $3f + 1$ replicas. These systems are designed to reach agreement with the presence of faults. This has two limitations. First, it requires the designer to choose a value for f beforehand based upon on the expected number of faults. Secondly, it requires a large amount of replication in all runs of the algorithm to handle the case where there are faults, when in reality faults should only occur in the minority of cases. In particular for complex services such as SDN controllers, this can have an impact on scalability and cost, as extra replicas require further resources. Increasing the number of replicas can also lead to increased processing time for switch events due to the extra network communications involved.

Therefore, I make use of an approach that only requires $2f + 1$ replicas, using $f + 1$ when under normal operation, reverting to $2f + 1$ if a fault occurs. This is done by relaxing the *fault tolerant* requirement to a level more akin to *fault detection*, with a recovery process built into the protocol to provide fault tolerance, but only when required. I assume that there are further quorums of controllers outside of the primary quorum, which are able to be called upon if required. This allows me to use an *all-or-nothing* approach to consensus. If at any point there is disagreement, the switch will choose a second quorum of size f, combining the responses from this new quorum with the original and take a majority vote with a $2f + 1$ level of fault tolerance. This allows for more efficient processing of flows in scenarios where no fault occurs, but still provides tolerance when faults do occur.

5.4 SDBFT protocol

SDN controllers can operate in both reactive mode, where the switch sends a request to the controller, and the controller sends a response, and proactive mode, in which the controller sends instructions to the switch without a switch event to trigger the action. I focus on the reactive case, and leave the handling of proactive controller behaviour to future work. This does limit the suitability of controller

applications to those that are purely reactive in nature. Even some applications which at first glance appear to be reactive are not necessarily suitable, for example the Floodlight controllers `Forwarding` application only activates in reaction to a request from a switch, however will proactively install flow rules to route a new flow on all switches in the path under its control.

This protocol differs from existing Byzantine fault tolerant protocols in two ways. Firstly, the fault-tolerance of the protocol is relaxed to instead be *fault-detecting*, with a recovery process to prevent a detected fault from impacting the network. This allows for the simpler consensus protocol described below, and meet requirement **R1 (low latency)**, whilst also meeting requirement **R2 (fault tolerant)**. Secondly, the protocols relaxes the ordering requirement usually found in fault tolerant protocols. Within an SDN network there is not a strict requirement that requests from switches are handled by a controller in any particular order, especially if the controller has a reliable view of the network topology. I do not enforce message ordering, which reduces the amount of communication steps required and further helps meet requirement **R1**. By reducing the number of controllers required as part of the primary quorum, I also help to meet requirement **R4 (scalable)** as fewer controllers are required within the system compared to traditional BFT approaches, with a reduced number of messages (and hence network load).

5.4.1 Assumptions

I assume controllers are deterministic. Controllers are running applications which, on receiving a request from a switch, should give the same result, on the assumption that different controller instances share the same up to date view of the network. As an example, the Floodlight `Forwarding` and `LearningSwitch` applications, the two default used for routing, utilise an implementation of Dijkstra‘s algorithm to identify the shortest route to the destination, which is a deterministic algorithm. Other applications, such as firewalls, should operate similarly. For more complex applications, such as load balancers, this may not hold true and would rely on sufficient data being shared amongst controllers to ensure an up-to-date view of the network state is maintained.

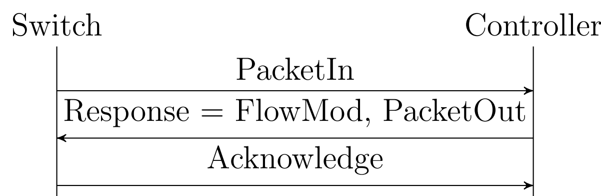


Figure 5.3: Typical switch to controller communication

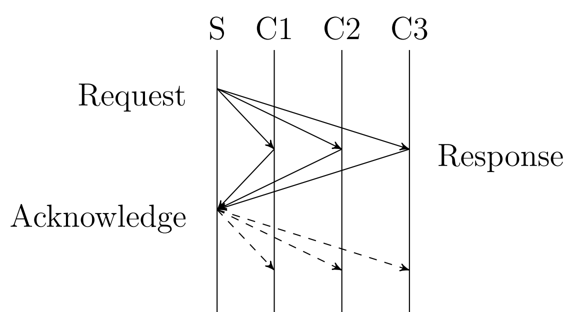


Figure 5.4: Switch to controller consensus, working state

I assume that an individual controller only requires knowledge of the switches for which it is a member of the primary quorum and the backup quorum. A controller itself does not require knowledge of which other controllers form the primary quorum, and which controllers form the backup quorum, because the SDBFT protocol does not require messages to be sent between controllers for the protocol. Further, a controller does not know which controllers are responsible for other switches within the network. This knowledge is only known to the network administrator. In the current SDBFT approach, controllers learn the identity of the other controllers in the primary and backup quorums due to the publisher-subscriber model used for consistency (as discussed in Section 5.7). An attacker in the network could use this information in order to perform targeted attacks on switches by identifying which controllers to compromise. At this stage, I do not consider this kind of insider threat as part of the threat model.

Algorithm 1 Switch action on receiving packet not covered by existing flow rule, normal operation (reactive control)

```
Require: Primary Quorum PQ_{S_i}
  procedure ProcessPacketIn(p)
      for all C_j ∈ PQ_{S_i} do
          sendPacketIn(p,C_j)
      end for
      waitForAllResponses()
      if allResponsesEqual() then
          forwardToSwitch(response)
          sendAcknowledgement(PQ_{S_i})
      else
          PQ_{S_i} = faulty
          contactBackupQuorum(BQ_{S_i})
      end if
  end procedure
```

5.4.2 Normal Operation

On receiving an unmatched packet, the switch will send a request rq (usually a `PacketIn` message) to the set of controllers in the current primary quorum PQ_{S_i}. Each controller will process the request and generate a response (r_c), which is returned to the switch. The switch will then collect the responses from each controller, forming the multiset R (under normal operation, all elements should be equal and so a multiset is required). All responses have been received when $|R| = k = |PQ_{S_i}|$. The root set (the set of distinct elements) of R should have a cardinality of 1, and the *correct* controller response should have a multiplicity of the quorum size k. If this holds, then all responses are equal and the switch will process the response and perform the command (install a flow rule, send packet out, etc.), and will send an acknowledgement to each controller indicating acceptance. The acknowledgement contains a copy of the switch request, and controller response. Each controller can then update its datastore with the new switch state. Optionally, all messages can be signed by the sender to allow for verification later. An example of the communication between a switch and single controller can be seen in Figure 5.3. Figure 5.4 shows the full communication between a switch and quorum in the non failure state for a quorum of size 3.

Note that only two rounds of communication are required between the switch and controller to complete the routing decision and allow the network flow to proceed, as the acknowledgements are only used for verification. As acknowledgements do not need to be received instantly by the controller, they may also be buffered on the switch and then sent in batches at regular intervals to reduce communication overhead.

5.4.2.1 Proof

I now provide a proof of correctness of the protocol in normal operation. In the first instance, I prove that under normal operation all honest (non-faulty) controllers should return the same result. I then provide a proof that under normal operation, the protocol will complete in at most two rounds of communication.

Lemma 5.4.1. *Given a primary quorum PQ of controllers of size k, if all controllers are honest (non-faulty) then the set of returned responses R for a given request rq should be equal, which is accepted by the switch.*

Proof. If all controllers (and running applications) are deterministic and share an equal view of the network (see Section 5.4.1), then for a given request rq, each controller $c \in PQ$, where $c \in CC$ should handle this request and return a response $r_c \in R$, such that $\forall r_c \in R \bullet c \in CC \rightarrow \mathit{correct}(r)$, and on receiving all responses $|R| = k$ As all controllers are non-faulty, the switch will eventually receive a response from every controller. □

Lemma 5.4.2. *Given the multiset of received responses R where $|R| = k$, with the cardinality of the root set of R being 1, then the protocol will complete in no more two rounds of communication*

Proof. If, as by Lemma 5.4.1, all responses returned from the set of controllers are equal, indicating that all controllers are honest (non-faulty), then a switch will not need to take any further communication with the controllers in order

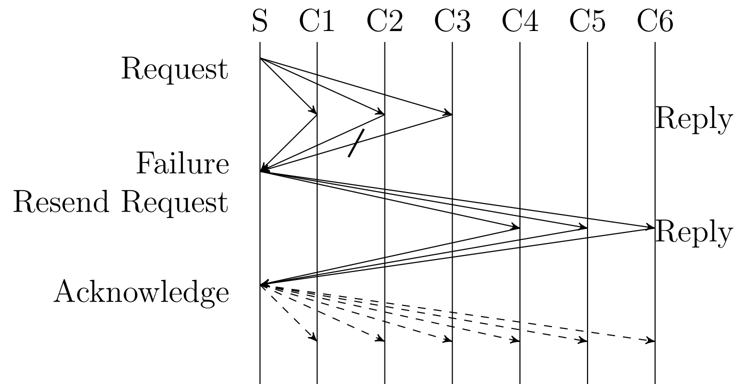


Figure 5.5: Switch to controller consensus, failure state. C3 sends an incorrect response to the switch. Second quorum of {C4,C5,C6} is contacted.

to take an action with the processed network flow/packet. Given the primary quorum PQ, then if $\forall c \in PQ \bullet c \in CC$, the switch will receive a correct response $r_c \in R$ from each controller. Once $|R| = k$, the response can be sent to the switch. Therefore, only two rounds of communication are required — the first is the broadcast of the request to the controllers of the primary quorum, and the second is the sending of a response from each controller to the switch. Controllers can send this response simultaneously without communicating with each other. This is equal to the traditional SDN control model, where communication is also completed within two rounds. □

5.4.3 Failure Operation

There are two fault scenarios which could cause disagreement: fail-stop and byzantine faults.

Fail-stop faults A fail-stop fault will manifest when a controller goes offline and does not respond to a switch request. This is either caused through controller crash, or maliciously through termination of the controller process or a block on the controller from responding to messages. In the case where the controller process is terminated, this can be caught through a broken socket. In the case

Algorithm 2 Switch action on receiving packet not covered by existing flow rule whilst in failure state (reactive control)

```
Require: ExpandedQuorum EQ_{S_i} = PQ_{S_i} ∪ BQ_{S_i}
  procedure ProcessPacketIn(p)
      agreed=false
      for all C_j ∈ EQ_{S_i} do
          sendPacketIn(p,C_j)
      end for
      waitForResponses()
      while agreed=false do
          receiveResponse()
          if hasMajorityResponse() then
              forwardToSwitch()
              acknowledge()
              agreed=true
          end if
      end while
  end procedure
```

where the controller takes too long to respond, a timeout can be used. The timeout should be set based on the average response time of controllers. In this scenario the cardinality of the multiset of responses $|R| < k$, however the cardinality of the root set is still 1.

Byzantine faults A byzantine fault will manifest as a controller response that does not match the correct operation. As mentioned, to allow for smaller quorum sizes I take an *all-or-nothing* approach for consensus. If even one controller return a response different to that of the others, then the switch will enter recovery mode and choose a new quorum. This activity can be performed as soon as the switch receives a response that does not match those previously received. In this scenario, the cardinality of the multiset of responses is $|R| = k$, whilst the cardinality of the root set is greater than 1.

In practice, a scenario could occur where both a fail stop and byzantine fault occurs simultaneously, however the same protocol applies.

On receiving a set of responses from the control quorum that are not in complete agreement, the switch will move into failure mode. The switch will contact

the assigned set of backup controllers, BQ_{S_i}. The goal of the switch is to construct an extended quorum, EQ_{S_i}, of size $|EQ_{S_i}| \geq 2f+1$, where $EQ_{S_i} = PQ_{S_i} \cup BQ_{S_i}$. The switch will send the original request to the quorum of backup controllers, and collect responses from each. The switch will then accept the majority response from both the primary and backup controllers. For future requests, the switch will contact both the full extended quorum EQ_{S_i} in one step, and always take the majority response. This will continue until manual intervention from an administrator to remove the faulty/compromised controller(s).

When a failure occurs, the switch acknowledgement will include the original request and a copy of each response from the controllers. This can be logged by controllers to be used by network administrators to identify the malicious controller based on the response.

Figure 5.5 shows the recovery protocol for an initial quorum of size 3, where one is faulty. A second quorum of 3 controllers is incorporated, and the switch forwards the initial request to them. In the recovery state, 2 extra rounds of communication are required over the normal case. For all subsequent communication whilst under the failure state, the switch will send all request to the expanded quorum as is done in the working state, resulting in 2 rounds of communication (including acknowledgements), with the only difference being the use of a majority vote for responses.

5.4.3.1 Proof

I will now prove the correctness of the protocol under failure operation. I will first show that as long as one controller is honest (non-faulty), then the presence of faulty or malicious controller can be identified and failure operation triggered. I will then prove that, on the occurrence of a fault the protocol will complete in at most 4 rounds of communication, and then revert to two rounds of communication in further rounds.

Lemma 5.4.3. *Given a primary quorum of controllers of size k, where $k = f+1$, as long as one controller is honest (non-faulty), then the protocol will enter failure mode even if all other controllers are malicious or faulty.*

Proof. In the case of failure there exists a faulty subset of the primary quorum $FQ \subset PQ$, where $\forall c \in FQ, c \in F$ and $|FQ| \leq f$. There also exists a correct subset $CQ \subset PQ$, where $\forall c \in CQ, c \in CC$, $|CQ| \geq 1$. These two sets are disjoint, $CQ \cap FQ$ and $|CQ| + |FQ| = k$.

If all controllers in PQ return a response r_c to form the set of responses R, then it can be assumed that the set of responses will contain a subset of incorrect responses $FR \subset R$, where $|FR| = |FQ|$, and a subset of correct responses $CR \subset R$, where $|CR| = |CQ|$. This will result in the cardinality of the root set of R becoming greater than 1, which will trigger failure mode. Note that this assumes byzantine failures — any indication of a fail stop failure where a controller disconnects from the switch or fails to return a response within a given timeframe will trigger failure mode. This can be formalised as if the request was sent at time T with a pre-defined timeout t, then if at time $T + t$ the set of responses $|R| < k$, then a failure can be assumed to have occurred.

□

Lemma 5.4.4. *Given an extended quorum* $EQ = PQ \cup BQ$*, where* $|EQ| \geq 2f+1$*, then as long as* $f + 1$ *controllers are honest (non-faulty) the protocol will return a correct response, meaning that a switch is required to be mapped to* $2f + 1$ *controllers to handle* f *faulty controllers.*

Proof. In the case of failure mode there exists a faulty subset of the extended quorum $FQ \subset EQ$, where $\forall c \in FQ, C \in F$ and $|FQ| \leq f$. There also exists a correct subset $CQ \subset EQ$, where $\forall c \in CQ, c \in CC$ and $|CQ| \geq f + 1$. These two sets are disjoint, $CQ \cap FQ$ and $|CQ| + |FQ| \geq 2f + 1$.

The expanded set of responses R from the extended quorum will contain $FR \subset R$ faulty responses where $|FR| = |FQ|$, and $CR \subset R$ correct responses where $|CR| = |CQ|$. In order to provide a majority vote, $|CR| > |FR|$, and therefore $|CQ| > |FQ|$, so to handle f faulty nodes $|EQ| \geq 2f + 1$. To simplify,

on receiving a set of responses R from the extended quorum of controllers of size $2f + 1$, the switch will look for the most common response within this set, representing the majority response from the controllers. If $f+1$ of the controllers are honest and non-faulty, and therefore return a correct response that is received by the switch, then this will always form the majority response, as the non-faulty controllers outnumber the f faulty controllers. Therefore the minimum number of controllers to ensure this majority is $2f + 1$ to handle f faulty nodes. In the case of fail stop failures, if faulty controllers fail to send responses, the switch can complete the protocol as soon as it receives $f + 1$ matching responses from the correct controllers. □

Lemma 5.4.5. *Given an extended quorum EQ, where $|EQ| \geq 2f + 1$, then the protocol will complete in no more than 4 rounds in handling the first request where a fault is identified.*

Proof. As per Lemma 5.4.3, on the initial request rq from the switch where at least one controller is faulty such that $FQ \subset PQ$ and $|FQ| \geq 1$, then failure mode will be triggered. This initial request will require two rounds of communication as in normal mode (see Lemma 5.4.2).

The protocol now needs to form the extended quorum $EQ = PQ \cup BQ$ by resending the request to the backup quorum BQ. This will again require two rounds of communication — to send the request to each $c \in BQ$ and to receive for a response from each with an aim to collect a set of responses R where $|R| = |EQ|$.

On the assumption that the correct subset $CQ \subset EQ$ is sufficiently large, such that $|CQ| \geq f + 1$, then after all correct members of the backup quorum BQ have returned a response, the subset of correct responses $CR \subset R$ will represent the majority response of all controllers such that $|CR| \geq f + 1$, resulting in a accepted response by the switch and the completion of the protocol. □

Lemma 5.4.6. *Given an extended quorum EQ, where $|EQ| \geq 2f+1$, the protocol*

will complete in no more than 2 rounds for all further requests following the occurrence of a fault.

Proof. As in the normal operation and Lemma 5.4.2, on future requests following the triggering of failure mode the switch broadcasts the request to both the primary and backup quorums (the extended quorum EQ in one round. Each controller in EQ will return a response whether faulty or non-faulty (or will not send a response in the case of fail-stop failures), such that the set of responses R will contain a subset $CR \subset R$ of correct responses, where $|CR| \geq f + 1$. On receiving $f + 1$ matching responses, then the switch can perform the action contained in the response and requires no further communication with controllers. No inter-controller communication rounds are performed. □

5.5 Signatures

The first benefit to introducing signatures into the SDBFT protocol is to meet requirement **R5 (Non-Repudiation)**. By only permitting signed messages from controllers, the switch can verify that a controller is part of its primary (or backup when in failure mode) quorum and discard messages from other controllers, or third-party hosts injecting forged messages from a spoofed controller. Furthermore, the use of signatures also provide message integrity, meeting requirement **R6 (Message Integrity)**. Once the controller has signed a message, the signature can be used to ensure that the message has not been changed in transit.

In order to use signatures, both controllers and switches are required to generate a public key, pk and a private key, sk. The private key must be kept securely on the switch, or controller host. I assume that each switch knows the public key of all the controllers within the network, and similarly each controller knows the public key of all switches. In the simplest case, the keys can be generated and distributed manually on setup. Secure key distribution is a complex problem and is out of scope of this work, and so I make the assumption that public keys have already been securely distributed to the relevant parties.

For each message sent between the switch and controller, a cryptographic signature is generated, and sent along with the plaintext message. The specific signature algorithm in use can be chosen by the implementor of the protocol to provide the require level of security to performance impact (see 5.5.2). In this implementation of the protocol, I use the Java Crypto library for providing signatures, using RSA with a 1024 bit key with the SHA512 hashing algorithm (as discussed in Section 6.3.4).

To generate the signature, the plaintext message is first hashed, and then this hash is encrypted using the private key of the sender. I represent this action through the use of a $Sign\{message, key\}$ function where $message$ is the message to be signed, and key is the private key sk of the sender. I use the sk_n notation to refer to the secret key of sender n. The $Sign$ function returns a cipher-text, cs which is the cryptographic signature. This is appended to and sent with the plaintext message.

On receiving a signed message, the receiver extracts the signature. Using the known public key pk_n of the sender, they use a $Verify\{message, cs, pk_n\}$ function, which takes the plaintext message, signature and public key and verifies that the message is correct. This is usually done by computing the hash of the plaintext message, decrypting the signature (which should return the hash of the sent message), and check that both the hash of the received message and signature hash match. If both match, then the message is genuine and the sender is verified. An example of signed communication can be seen in Figure 5.6

The $Sign$ and $Verify$ functions are provided by a cryptographic library, for example the Java Crypto library or OpenSSL.

For the SDBFT protocol itself, only messages sent from controllers to switches need to be signed, as the SDBFT protocol is designed to protect against malicious controllers, with the assumption that the switch itself is not compromised. It is strongly recommended, however, that messages from the switches are also signed. This ensures message integrity and non repudiation on behalf of the switch. As I have previously mentioned, an attacker could intercept responses from a controller to a switch through a person-in-the-middle attack to inject faults into the network. It is possible that an attacker could also intercept a request from a switch to a controller, and modify the `PacketIn` message to cause the controller to return

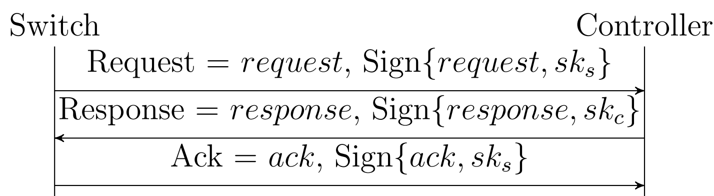


Figure 5.6: Signed switch to controller communication

an incorrect response. Signing messages from the switch provides a mitigation against such attacks. Further, it provides some level of protection against a compromised switch, which even though is out of scope of this work, is a valid concern in a network under attack.

5.5.1 Controller Verification

When a fault occurs, the switch, on receiving an update from the expanded quorum, will send an acknowledgement to all controllers containing the original signed request, as well as the signed responses from all controllers. This can be logged by controllers and used by network administrators to verify which controllers were compromised, with the signatures used to verify that controllers send the updates (and can be crossed checked with controller logs). The acknowledgement messages could be expanded to apply automated verification amongst controllers, for example by applying a version of the SDN-RDCD protocol as proposed by Zhou et al., although I leave this to future work [245].

5.5.2 Limitations

The primary limitation of introducing signatures is the cost associated with signing all messages. For an individual switch request, two signing and two verification operations need to be carried out before the flow rule can be installed on the switch (the signing of acknowledgements does not need to occur for a flow rule to be installed). These operations increase the processing time for a request, and so increase request latency. Furthermore, in particular for a high-traffic, dynamic network where a large number of switch requests need to be processed, this can

represent a substantial additional amount of processing, in particular on the controller side, which reduces the volume of requests controllers can handle. This represents a decision on behalf of the network operator in order balance the required network performance, with the security benefits of using signed messages.

Whilst OpenFlow is designed to be sent over TCP connections, which provides some protection against replay attacks through the TCP protocol, OpenFlow itself does not feature any anti-replay mechanism. It is therefore feasible that an attacker who can sit on the connection between the switch and controller and perform a person-in-the-middle attack, for example through the use of ARP spoofing, could replace packets with previously collected and signed OpenFlow requests or responses, which would pass the verification step. This could be prevented by incorporating an anti-replay mechanism into the OpenFlow protocol. A simple approach to this would be to use the `xID` field within the OpenFlow packets as a sequence number, and ensuring that messages received from the switch do not use a previously seen `xID` field. Alternatively, an anti-replay token could be incorporated into the encrypted signature, which can be checked and removed before the hash is verified.

5.6 Controller Assignment

The problem of controller assignment is one of the more difficult problems to solve when considering a consensus based SDN control approach. Whilst the problem is also an issue in traditional distributed SDN architectures where multiple controllers are in use, it becomes more complicated when a single switch needs to communicate with multiple controllers at the same time.

The controller assignment has two main impacts on the network. First, controller assignment in the SDBFT architecture corresponds to the construction of the primary and backup quorums for a given switch. Secondly, the controller assignment approach, in particular how efficiently controller resources are utilised, can dictate how many controllers are required in the network.

In this Section I define the requirement for controller assignment in SDBFT, and provide a simple algorithm for performing controller assignment in the SDBFT

architecture. Using a Java-based simulation I then perform a simple evaluation of its performance across various network sizes.

There has already been a large amount of work in the literature providing solutions to the controller assignment problem, both in the context of single-primary with multiple backup controller architectures, as well as in the multiple primary controller scenario. In Section 5.6.4 I provide an overview of existing approaches to this problem from the literature, indicating how these could be applied to the SDBFT architecture.

5.6.1 Requirements

The factors that need consideration when assigning controllers to switches are as follows:

Switch to controller latency The switch to controller latency is the primary limiting factor on the performance of the SDBFT system. More specifically, as the protocol requires a response from all controllers within the primary quorum, the highest latency controller limits the time to a response. Therefore, minimising this latency is important. The switch to controller latency to controllers in the backup quorum is also important, as in the failure state the performance is then limited by the slowed switch to controller connection in the controller set formed by the primary and backup quorums.

Controller load A controller has a maximum number of requests per second that it is able to handle. Different switches will generate a varying number of requests per second, depending on their traffic load. Therefore, when assigning switches to controllers, this controller capacity must be respected. A controller also needs to maintain some capacity as a member of backup quorums to ensure it does not become overloaded when it becomes in use.

When using a traditional BFT approach, the controller to controller latency is also an important factor as controllers need to communicate with each other as part of the protocol. Whilst the controllers within the SDBFT system do communicate to share network state information, this is out of band of the main protocol and so inter-controller latency is not a major issue.

5.6.2 Simple Algorithm

Algorithm 3 Simple Controller Assignment Algorithm

```
procedure ASSIGN PRIMARY CONTROLLERS
    for all S_i ∈ S do
        SortControllerList()
        while |PQ_{S_i}| < k do
            for all C_j ∈ C do
                if hasCapacity(C_j) then
                    assign(C_j, PQ_{S_i}))
                    reduceCapacity(C_j)
                end if
            end for
        end while
    end for
end procedure
procedure ASSIGN BACKUP CONTROLLERS
    for all S_i ∈ S do
        SortControllerList()
        while k + |BQ_{S_i}| < 2f + 1 do
            for all C_j ∈ C do
                if hasCapacity(C_j) and C_j ∉ PQ_{S_i} then
                    assign(C_j, BQ_{S_i}))
                    reduceCapacity(C_j)
                end if
            end for
        end while
    end for
end procedure
```

I present a simple algorithm for performing controller assignment in the SDBFT architecture, an overview of which can be seen Algorithm 3.

I assume a set of switches, S, and a set of controllers, C. Each controller $C_i \in C$ has a $capacity(C_j)$, which indicates the available capacity of the controller, i.e. the number of switches it is able to control. In the simple case, this will be equal for all controllers. For each switch $S_i \in S$, there is a list of $latency(S_i, C_j)$ pairs, which represents the latency of switch S_i to controller C_j.

The algorithm then operates as follows:

1. For each switch $S_i \in S$ do the following to assign primary controllers:

    (a) Construct a list of $< C_j, latency(S_i, C_j), capacity(C_j) >$ tuples, one for each available controller.

    (b) Sort this list of controllers

    (c) Iterate through the list of controllers, and if the controller has available capacity, add it to the primary quorum PQ_{S_i}. Reduce the available capacity for controller C_j.

    (d) Stop once $|PQ_{S_i}| = k$

2. For each switch $S_i \in S$ do the following to assign backup controllers:

    (a) Construct a list of $< C_j, latency(S_i, C_j), capacity(C_j) >$ tuples, one for each available controller.

    (b) Sort this list of controllers

    (c) Iterate through the list of controllers, and if the controller has available capacity, and the controller has not been assigned to the primary quorum, $C_j \notin PQ_{S_i}$, add it to the backup quorum BQ_{S_i}. Reduce the available capacity for controller C_j.

    (d) Stop once $k + |BQ_{S_i}| = 2f + 1$

The step of sorting the list of controllers is the key aspect in how the protocol will assign controllers. The algorithm can either prioritise minimising switch–controller latency, or maximising controller usage (ensuring control load is as evenly spread as possible). If latency is prioritised, then the list of controllers will be sorted first by increasing switch–controller latency, then decreasing available controller capacity. This has the effect that switches will pick the lowest latency controllers for the primary quorum first, but at the expense of a less evenly distributed switch load amongst the controllers. If controller usage is prioritised, then the controller list is first sorted by decreasing controller availability, and then increasing switch–controller latency. If there is sufficient controller capacity, then this will result in an even distribution of switches across the pool of controllers, at the expense of switch–controller latency.

Assuming each controller has the same initial capacity, the minimum number of required controllers is equal to the size of the number of controllers required per switch, multiplied by the number of switches, and divided by the initial capacity of a single controller, such that:

$$requiredCapacity = s(k + b)$$

$$requiredControllers = \lceil requiredCapacity / controllerCapacity \rceil$$

So, for example, with a primary quorum size $k = 4$, and backup quorum size $b = 3$, with 100 switches, then the total required capacity is 700. If I assume, for example, that a single controller can handle 40 switches, then the minimum number of controllers is 17.5, which must be rounded up to 18 full controllers.

Note that by default this algorithm assumes that the assignment of a backup controller requires as much controller capacity as the assignment of a primary controller. In normal operation the switch does not utilise any capacity of the backup controllers, unless a fault has occurred. However, to ensure there is sufficient capacity amongst controllers to handle failures, then the capacity for backups should be reserved. This is however up to the network owner — the algorithm can be configured to reduce utilised controller capacity by a smaller amount for backup assignments than primary assignments, which will result in a requirement for less controllers, but at the expense of backup capacity in the case of a large number of faults.

I also assume that each switch has an equal impact on controller capacity meaning that every switch added to a controller will have the same reduction in available capacity for that controller. In practice, however, switches have different capacity requirements based upon their type and location within the network (backbone switches will require much more capacity than edge switches). The assignment algorithm can be extended to support different controller capacities by assigning each switch a cost L_{S_i}, which is deducted from the available capacity of a controller when that switch is assigned as a primary or backup. This would

change the computation of required capacity in the network to:

$$requiredCapacity = \sum_{S_i \in S} L_{S_i}(k + b)$$

5.6.3 Simple Algorithm Performance

5.6.3.1 Implementation

In order to test the performance of the controller assignment algorithm, I utilise a Java-based simulation. The simulation takes as a parameter the number of switches and controllers to generate, the capacity of an individual controller and the size of the primary and backup quorums. The simulator will then generate the required number of switches and controllers, and for each switch will randomly generate a latency from that switch to each controller. For testing purposes this is an integer value in the range 1 to 10.

The simulator is able to calculate the minimum number of controllers required to operate by taking the number of switches, and attempting to perform controller assignment with an increasing number of controllers until there is sufficient capacity for all switches to have complete primary and backup quorums.

To provide a comparison, I also implement a fully randomised approach in which the list of potential controllers is shuffled instead of sorted, resulting in a random set of controllers for the primary and backup quorums.

5.6.3.2 Simple Example

To demonstrate the controller assignment algorithm I first use a simple example of a 5 switch, 7 controller network, in which $k = 4$ and $b = 3$. In this example, I assume a controller has a capacity of 10, and each switch has equal cost. I provide the results of three tests — assignment prioritising switch-controller latency, assignment prioritising controller capacity, and fully random assignment. Switch to controller latencies are constant across the three tests, and can be seen in Figure 5.7a.

	ID	Controllers 1	2	3	4	5	6	7
Switches	1	6	6	3	1	5	3	5
	2	2	10	7	5	2	1	3
	3	4	4	8	10	2	9	4
	4	1	1	4	7	8	7	5
	5	2	10	2	8	4	2	3

(a) Switch to controller latencies

			Latency	**Capacity**	**Random**
Switches	1	Primary	3,4,5,6	3,4,5,6	1,2,3,5
		Backup	1,2,7	1,2,7	4,6,7
	2	Primary	1,5,6,7	1,2,6,7	2,3,4,6
		Backup	2,3,4	3,4,5	1,5,7
	3	Primary	1,2,5,7	1,2,5,7	1,2,4,5
		Backup	3,4,6	3,4,6	3,6,7
	4	Primary	1,2,3,7	1,2,3,4	2,4,5,7
		Backup	4,5,6	5,6,7	1,3,6
	5	Primary	1,3,6,7	3,5,6,7	2,3,4,6
		Backup	2,4,5	1,2,4	1,5,7

(b) Controller assignments

	Latencies				**Assignments**							
	Primary		Backup		Primary				Backup			
	Mean	S.D.	Mean	S.D.	Mean	S.D.	Min	Max	Mean	S.D.	Min	Max
L	2.7	1.27	7.33	1.85	2.86	0.99	1	4	2.14	0.99	1	4
C	3.3	2.15	6.53	2.39	2.86	0.35	2	3	2.14	0.35	1	3
R	5.3	2.88	3.87	2.36	2.86	1.25	1	5	2.14	1.25	0	4

(c) Statistics. Modes L=Latency, C=Capacity, R=Random

Figure 5.7: Controller assignment example with 5 switches, 7 controllers. Primary quorum size 4, backup quorum size 3, controller capacity 10

Figure 5.7b provides the output of this example, and Figure 5.7c shows the overall characteristics of the outputs of the three approaches. Latency values represent the mean switch-controller latency for a primary (or backup) quorum connection. As can be seen, prioritising latency results in primary quorums with the smallest mean latency, with less variance, whilst prioritising for controller capacity results in slight higher latencies, but with much more variance. A random controller assignment results in a high switch-controller latency. However, prioritising latency has the impact that the lower latency connections are used by primary quorums, resulting in higher latency connections for backup quorums when compared to the approach prioritising controller capacity.

	Latencies				Assignments							
	Primary		Backup		Primary				Backup			
	Mean	S.D.	Mean	S.D.	Mean	S.D.	Min	Max	Mean	S.D.	Min	Max
L	1.78	0.95	3.47	1.1	30	2.65	26	36	22.5	3.51	15	20
C	3.01	2.13	3.74	2.17	30	0	30	30	22.5	0.5	22	23
R	5.79	2.82	5.55	2.85	30	6.26	18	43	22.5	4.56	16	34

Figure 5.8: Controller assignment example with 150 switches, 20 controllers with 100 capacity each.

Conversely, prioritising capacity results in much more evenly distributed controller load, shown by the lower variance in Figure 5.7c. Note that all approaches result in the same mean value for primary and backup assignments - this is because the total of primary and backup assignments is always the same. The way in which this load is distributed, shown by the variance, does change across approaches. The effect is limited in this example as a large amount of controller load is utilised, however if I look at a large example of 150 switches, 20 controllers (with 100 capacity each), as seen in Figure 5.8, then the benefit of prioritising controller capacity is made more apparent. In this example, controller load is almost perfectly distributed when prioritising controller capacity over latency. Conversely, in this example prioritising latency results in substantially lower latencies for primary quorums, and comparable latency for backup quorums. As in the simple example, the random approach generates slower quorums with much more uneven distribution of controller load.

5.6.4 Existing Approaches

Li et al. [136, 137] explore the issue of controller assignment in an architecture utilising a BFT approach for SDN controllers. They define the controller assignment in fault-tolerant SDN (CAFTS) problem. CAFTS represents the problem of assigning controllers to switches, satisfying the requirements of the BFT algorithm in use, minimising the latency between controllers assigned to a single switch (to aid in the performance of the BFT algorithm) and to maximise the utilisation of controller resources. The proposed controller assignment algorithm takes into account the differing levels of security required by individual switches

(a switch with higher security requirements is assigned a greater number of controller replicas to provide greater resilience). Their requirements first assignment (RQFA) algorithm is shown to provide more efficient controller assignment than a randomised approach.

Mohan et al. explore a solution for fault tolerant SDN control that is similar to the SDBFT protocol, with the work focusing on the controller assignment problem. [158, 159]. The work focuses on the issue of controller assignment, where they propose an algorithm that aims to minimise the total number of controllers whilst considering the switch to controller latency and controller capacity. The approach is similar to the simple approach described above, however also supports the remapping of controllers when a failure occurs. The proposed algorithm, MINCON, also prioritises controller assignment for switches with higher loads in terms of flow counts to increase performance. It is worth considering that although the approach aims to minimise the number of controllers, this may not always be the best approach if the capacity to host additional controllers is available, as a greater number of controller replicas can handle a greater number of faults, with fewer switches under the control of controllers if they were to become compromised.

5.7 Controller Consistency

Controller consistency is a well-known issue within distributed SDN controller architectures [16]. Requirement **R3 (consistency)** dictates that controllers should be properly replicated. The primary element of replication required within a distributed SDN control plane is the the knowledge stored by the controller on the current state of the network. In order to provide routing functionality, the controllers need to maintain information about where devices are located within the network. At a low level, this includes the specific ports on switches to which devices and hosts are connected, as well as the connections between switches. Different routing controller applications required different amounts of information about the network topology when pushing a route. For example, the Floodlight controllers layer 2 `LearningSwitch` application only considers the next hop on

the route, i.e it knows which port to forward the packets to on a per-switch basis (previously learnt using LLDP probes or observed packets). The `Forwarding` application operates differently to this, and uses the Floodlight `Topology` service to compute the complete route to the destination using Dijkstra's algorithm, installing appropriate flow rules on all switches along the path.

In the scenario where a single controller controls the network however this is not an issue as the controller has a singular view of the network topology. If a device location is not known, then it can flood packets in order to locate it and updates its datastore. However, if multiple controllers are in use, each controlling only a limited portion of the network, then controllers will have different views of the current network topology. In particular, controllers in a backup quorum will need to have a up-to-date view of the switch state so that they can make suitable routing decisions if called upon.

If controllers are not properly replicated, the returned responses will not necessarily match. As a simple example, if controller A has an up to date view of a switch and sees a packet to device D, which it knows the location of, it will return a PacketOut and a FlowMod directing the flow out of the correct port. If controller B does not have this information, then it will only return a PacketOut which will be flooded to all ports to identify which is the correct port to use. An example of when this may occur is when the SDBFT protocol moves into failure mode and the set of backup controllers are utilised, which have not received any earlier requests from the switch.

Therefore, there needs to be a process for controllers to synchronise in order to share information on the network topology. This approach could be expanded to also include more detailed information about the network state to aid in more complex applications, such as load balancing.

Consistency is either defined as strong or eventual. Strong consistency provides the guarantee of consistency — updates are pushed and action is only taken once all replicas have received and processed the update, but at the cost of additional latency (whilst an update is being replicated, no further requests can be processed). Eventual consistency, on the other hand, only assumes that all replicas will become consistent eventually, with the cost that some replicas may

respond to requests with stale data. Applications using eventual consistency must be able to handle responses based upon stale data.

I assume for SDBFT a requirement of eventual consistency. The primary quorum will be consistent for a given switch state, as each request from a switch will update the network state information stored by that controller. Backup controllers will receive updates from primary controllers, and on being contacted in the case of failure will also receive switch requests which can update stored network state. Further controllers within the control plane will need to receive updates, but a small amount of latency on these updates is acceptable. In the case where inconsistency occurs, this will appear as a fault, triggering the fault recovery protocol. As long as $f + 1$ of the combined $2f + 1$ controllers of the primary and backup quorum have an updated network state view, then the switch will receive a correct response.

This does limit the potential applications that can be utilised. In the Floodlight example, the single hop `LearningSwitch` application should be utilised rather than the `Forwarding` application, which requires a wider view of the network state.

5.7.1 Publisher-Subscriber Protocol

I use a publisher-subscriber model for distributing network state updates throughout the network. In the simplest approach, all controllers act as publishers, and are subscribed to updates from all other controllers. On handling a switch request that generates new knowledge about the network state, e.g. device a is connected to port b on switch c, a controller will publish an update. I do not specify the exact format for this update, as it may vary based upon the specific SDN controller being used. As an example, the Floodlight controller `DeviceManager` application already writes updates to the Floodlight `CommsService`, which is a publisher-subscriber system, whenever a new device is seen (for more information see Section 6.4.4). All controllers controlling a switch will publish updates whenever a new controller is seen.

In normal mode, on receiving $f + 1$ matching updates from the primary quorum, the controller will then update its datastore with the new value. If $f + 1$

updates are received, but they are not matching, then it can be assumed a fault has occurred. In this scenario, the receiver will wait for $f + 1$ matching responses to be received, as this represents a majority from the primary and backup quorums. This is a simplified version of the SDBFT protocol.

In the case where a fault occurs on a switch request for a new device, the backup may not have received an update from a primary controller, or may have received a malicious update from a compromised primary controller. In this scenario, the backup will be sent the original request from the switch, which it can process as the primary controllers would in order to extract sufficient information to bring it up to the same state as the primary quorum.

In order to provide non-repudiation and message integrity, the updates can be signed using the same public and private keys used for signing switch-controller messages, previously described in Section 5.5. This also prevents controllers from publishing false updates, as a full quorum of controllers will need to agree on the update. For an additional level of protection, the original signed request from the switch can also be included within the update, which would allow a receiving controller to verify the switch state update is caused by an actual switch event.

5.7.2 Existing Approaches

There are a number of works which explore the concept of adaptive consistency for distributed SDN controllers [13, 16, 200]. Rather than make the assumption that all updates must be propagated fully consistently (strong consistency), it is assumed that updates may take longer to propagate leading to different results being returned from different controllers (eventual consistency), however applications are designed to tolerate this inconsistency. The adaptive approach applies different consistency requirements to different types of controller decisions, with more network critical decisions reliant upon strong consistency, with less critical updates allowed to take longer to propagate. Whilst this approach may work in the distributed controller architecture in which a single controller is responsible for a particular switch, it would not work in solutions where multiple controllers are responsible for a single switch, which requires strong consistency.

A common approach taken by existing work is to apply a distributed datastore across the distributed controllers. For example, the Onix distributed controller architecture applies multiple techniques in order to synchronise its Network Information Base (NIB) which stores information about switch and network state [115]. Onix utilises a transactional database for slow, but reliable, network state updates. For network state updates requiring a higher update rate and availability a Distributed Hash Table (DHT) (based upon Dyname [63]) is used. This DHT supports one-hop, eventually-consistent, updates similar to my publisher subscriber model, however comes with the limitation that only a single controller can insert updates for a given value at any one time, otherwise collisions may occur.

Botelho et al. propose a fault tolerant key-value datastore for a distributed SDN controller architecture able to provide strong consistency [32]. Fault tolerance and consistency is achieved through the use of replicated state machines and a total-order multicast protocol, in this case BFT-SmaRt [22]. One particular element of the design is that datastore writes include an counter field, which aims to prevent inconsistencies from concurrent datastore writes, though it is unclear how well this would work when a large number ($f + 1$ or more) controllers are writing the same value simultaneously. A number of optimisations are proposed to increase throughput and reduce latency of datastore operations, though in all cases there is a noticeable impact on controller performance, as datastore reads can take 3-5ms to perform.

5.8 Limitations of Approach

In this section I discuss some limitations and open issues with the SDBFT protocol, and suggest some potential solutions that can be explored as part of future work.

5.8.1 Proactive Control

The SDBFT protocol focusses on the reactive SDN mode of operation, in which the controller only sends commands to the switch in response to a request from a switch. There will be some scenarios where the proactive mode of operation is also required in order to setup flow rules in advance. This could be down to proactive applications on the controller, or through human administrators manually configuring routes within the network. This presents two main challenges:

1. In the proactive application case, how can applications be synchronised to send requests at the same time with requests that can be matched across controllers? In the reactive case, a switch request triggers an immediate response form every controller, with each individual response matched to a switch request.

2. A human administrator would not want to connect to multiple controller instances in order to apply a single configuration change to a switch. This increases the human cost in terms of time, and increases the likelihood of erroneous updates being pushed by the administrator. This also has similar issues to the proactive application case in matching the controller requests on the switch.

It is feasible to assume that the majority of the functionality of proactive applications can be adapted to operate in the reactive mode. This will have the impact of a slight reduction in performance as the target flow will need to be sent to the controller rather than being handled by a flow rule already installed on the switch.

For the human administrator case, this could be partially solved through the use of a *master* controller that exists on a heavily secured machine, and does not take part in the normal control network but exists purely for manual administrator configuration. This controller can be kept offline when not in use to reduce the likelihood of compromise. This controller can authenticate to the switch for example using signed messages, with the master controllers public key loaded on the switch at install time and the private key kept securely. This

controller can proactively install flow rules on the switch without requiring the consensus protocol to be run.

Of course, this approach does not limit the impact of a malicious administrator. Similarly, if the master controller does become compromised, then it would gain arbitrary control over the network (though only when the master controller is *active*).

5.8.2 Controller Diversity

A particular challenge of consensus-based SDN control is the problem of controller diversity. The reasoning behind utilising multiple controllers is to limit the impact of a subset of these controllers from becoming compromised, assuming the majority of controllers are not compromised. The diversity of controllers, and the architecture supporting them, provides a challenge to this assumption. As an example, assume every controller consists of the Floodlight controller, deployed into an Ubuntu 18.04 virtual machine, running Java version 10, with the same user credentials across all virtual machines. On top of potential vulnerabilities within the Floodlight controller itself, a critical vulnerability found within the used libraries, java subsystem or operating system (or any other software on the machine), or indeed compromised credentials could allow an attacker to potentially gain control of every controller instance as if they find an access route into one controller, then the same route can be applied to all as every instance is identical.

This problem affects all consensus-based SDN control architectures. [136, 137, 22, 158, 159, 190]. The work by Qi et al. suggests controller diversity through deploying different controllers (in the paper POX, NOX and Floodlight are used), with the assumption that these controllers will still deterministically return the same result to a switch request [190]. Utilising multiple controllers in this fashion provides extra challenges — for example each application needs to be implemented for each controller and tested to ensure it is deterministic. Further, this only limits the compromise of the controller itself — if all controllers are running on identical hosts then an attacker could still gain control of each instance.

Whilst I do not attempt to solve this problem in this work, there are some steps that can be taken to help increase diversity and reduce the potential for an adversary to compromise all machines. Simple actions such as ensuring different passwords are configured for individual hosts can help reduce attack surfaces. Operating system defences such as Address Space Layout Randomisation (ASLR) and stack canaries can make exploiting vulnerabilities such as buffer overflows more difficult to exploit across multiple machines. Finally, removing unnecessary software and regularly patching machines can help reduce the attack surface to an attacker.

5.9 Conclusion

In this chapter, I discussed the design of the SDBFT protocol for fault-tolerant SDN control. I gave an overview of the operation of the protocol, including how it operates both under normal circumstances, and when a failure occurs. I then discussed the application of signatures in order to enforce message integrity and provide non-repudiation for controller messages. To support the SDBFT protocol, I also provide the design of a solution to the controller assignment problem, and a method for ensuring consistency amongst controllers.

In the next chapter, I discuss the challenges of implementing the SDBFT systems, and then in Chapters 7 and 8 I experimentally evaluate the performance of the SDBFT controller.

Chapter 6

Implementing the SDBFT Protocol

6.1 Introduction

In Chapter 5 I proposed a novel protocol for Byzantine fault tolerate agreement amongst SDN controllers, SDBFT. In this chapter I describe an implementation of the SDBFT protocol, and the challenges of implementing such a protocol. The primary implementation is that of the SDBFT proxy, which sits on the connection between a switch and a controller and carries out the switch-side logic of the SDBFT protocol, without having to modify a switch firmware.

I also describe the implementation of a comparative system using a tradition byzantine fault tolerant protocol. For this purpose I use the BFT-SMaRt protocol, implemented through the use of a publicly available Java library [22, 24].

6.2 SDBFT Implementation Overview

As the core SDBFT algorithm computation is performed on the switch side, the main focus of the implementation needs to be on the switch. The optimal

approach would be to take an existing virtual switch implementation, such as OpenVSwitch, and modify the switch *firmware* to incorporate the SDBFT logic. Whilst this would likely provide the most efficient implementation, this approach has some drawbacks. Firstly, this would require kernel-level programming, which is complex, and inserting the new logic into the switch whilst maintaining functionality could prove challenging. Secondly, this limits evaluation to that one particular switch implementation. For example, if I were to modify OpenVSwitch, a virtual switch commonly used for testing, I would be unable to test the logic on a physical switch without also implementing the logic into the physical switches firmware.

Rather than modify physical or virtual switch firmware, I decided to produce an OpenFlow proxy that will sit on the connection between the switch and the controller acting as a bump-in-the-wire device, leaving the switch unmodified. As well as providing a generalised solution for testing purposes, the proxy can support older switch hardware that would not be modifiable to handle multiple controllers, and so the use of a proxy can be applied to older SDN networks. This is a similar approach as taken by ElDefrawy and Kaczmarek, and Sherwood et al., within SDN controller architectures [76, 215].

Whilst implementing SDBFT as a proxy is easier, and more flexible for testing, it also has some potential drawbacks. Primarily, the proxy introduces an extra communication step, which requires the processing of packets, including parsing the OpenFlow messages from the intercepted packets, and then re-encoding them as raw data to send onto the controller. These extra steps would not be required if implemented on the switch, as the SDBFT logic can be applied before and after messages are sent to and received from the controller. To measure this effect, I perform testing with a simple TCP proxy in Chapter 8, which enables me to learn the baseline impact of introducing a proxy into the connection. There are potentially further efficiency improvements to be had with a native C based implementation rather than using a Java implementation, however I are unable to measure this without implementing the native version.

As well as the SDBFT proxy, I also require an SDN controller. I decided to use the Floodlight controller for this purpose, which is a Java based open-source controller [187]. Floodlight has a number of benefits for this use. Floodlight uses

the OpenFlow Java library provided by Loxigen [189], which can also be used for the proxy to provide consistency. Whilst other Java based controllers exist, such as ONOS, Floodlight is a fully featured controller, which supports multiple applications to be run simultaneously, whilst being a relatively small code base. As the controller requires modification, the smaller code base makes this task easier.

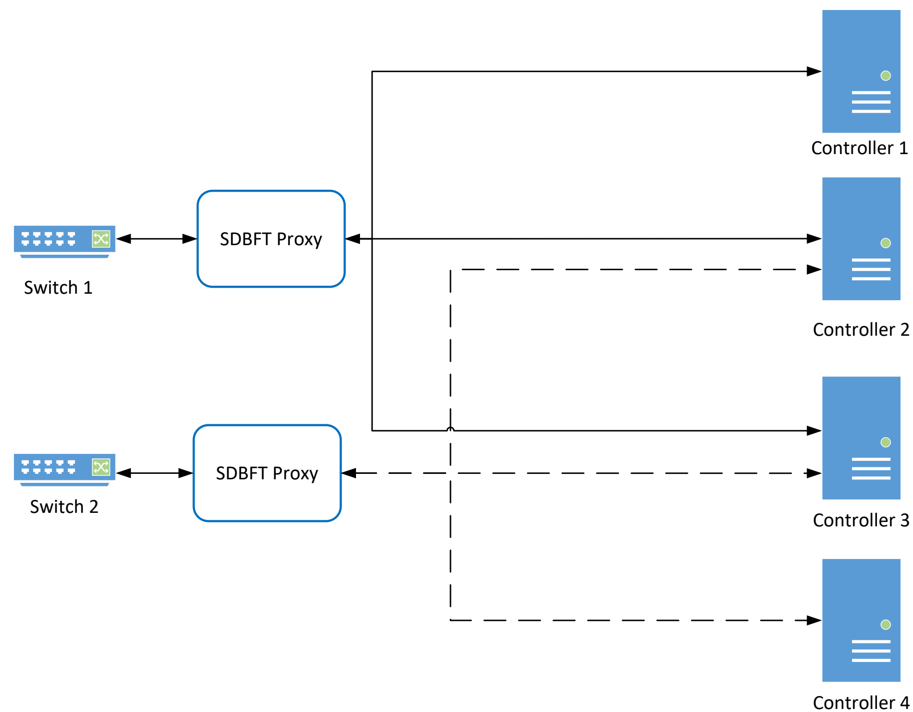


Figure 6.1: SDBFT Proxy Architecture

While Floodlight supports all OpenFlow versions up to 1.5, I focus on OpenFlow 1.3 for development and testing purposes due to its wide support on SDN switches. The SDBFT proxy also supports all OpenFlow versions up to 1.5 due to the capability provided by the OpenFlow Java library.

6.3 Proxy Implementation

The SDBFT proxy, implemented in Java intercepts all OpenFlow messages between the switch and controller(s). When configuring the switch, the IP address and port number of the proxy is used as the controller configuration. An individual instance of the proxy needs to be deployed for each switch in the network, however a single proxy will connect to multiple controllers. An example of this architecture can be seen in Figure 6.1.

The proxy is implemented using the OpenFlow Java library that is generated by Loxigen [189]. Loxigen generates OpenFlow libraries for C, Python and Java, based upon the OpenFlow specification. All versions of OpenFlow up to 1.5.1 are supported, though versions 1.0, 1.3.1 and 1.4.1 are intended for use in production [189]. The library is generated as source code, which is directly included within the Java project and compiled along with the proxy code. The library includes readers for extracting OpenFlow messages from raw byte arrays, and also includes factories for generating OpenFlow messages for a specified version of the Openflow protocol.

The only modification to the switch is that it is configured such that it connects to the proxy rather than to the controller. On receiving a connection from a switch, the proxy connects to the appropriate controller instances. On receiving a packet, the proxy will use the OpenFlow library to parse the packet and extract the OpenFlow message. It can then extract message type and perform the appropriate action, which are discussed in Section 6.3.5.

6.3.1 Configuration

The implemented proxy is configured by the use of a simple configuration file, an example of which can be found in Appendix A.1. The proxy implementation actually includes three proxy implementations — a simple TCP proxy, the SDBFT proxy and the BFT-SMaRt proxy (see Section 6.5), and the same configuration file is used for all three. The key parameters for the SDBFT proxy are the list of primary controllers, and the list of backup controllers. If signatures are in use, the configuration file is used to specify the parameters for the signing engine.

6.3.2 Communication

Java sockets are used for communication. On launching the proxy, a `ServerSocket` is started to accept a connection from a switch. On receiving a connection from a client, a client specific socket is generated, and a new `ClientConnection` object is generated. The `ClientConnecticon` is a Runnable object, which is responsible for reading messages form the client socket, and sending messages to the client over the socket. When the `ClientConnection` is created, a new thread is started to allow the connection to be repeatedly read from.

When the client has connected, the proxy then creates a connection to each controller, and creates a `ServerConnection` instance for each controller, which is functionally similar to the `ClientConnection`. A thread is started for each controller to allow concurrent communication with multiple controllers.

For certain switches, the switch makes an initial connection to the controller before participating in the OpenFlow handshake. For example, this always occurs when using the Mininet network simulator built upon OpenVSwitch. When these types of switches are used, the first connection from a switch is ignored, though from the switch perspective the controller is live. This connection is due to Mininet first opening a connection to the controller to test for liveness before instructing the switches themselves to connect. The proxy then waits for the second connection from the switch before connecting to the controllers. If this is not done, the controller will initiate the handshake on receiving a connection which the switch will then ignore, and the handshake will not complete.

6.3.3 Message Acknowledgements

As part of the SDBFT protocol, the proxy needs to send an acknowledgement of switch responses. When processing a response from the controller, the proxy generates this acknowledgement and sends to each controller. As these acknowledgements are not time-critical, the proxy can be configured to either send these acknowledgements immediately on receiving a response, or alternatively these responses are added to a buffer on the proxy. A new thread process is started, which sends the contents of this buffer to all of the controllers, clears the buffer and then

waits for a pre-defined amount of time (for testing purposes this is 100ms). This batch sending of acknowledgements reduces the amount of messages that needs to be send between the switch and controller, allowing the proxy to handle a greater number of switch requests.

6.3.4 Signatures

The signing of messages is performed using the Java `java.security` package. The signing algorithm, key generation algorithm, key size and signature length are selected using the configuration file. For testing purposes, keys are generated when launching an instance of the proxy. As I are primarily interested in the performance of signatures during testing, whilst keys are generated using the `SecureRandom` class provided as part of `java.security`, the seed is hard coded to '999', which causes every instance of the proxy to generate the same key pair (and can be repeated on the controller). Whilst this is unsuitable for a real-world deployment, it is sufficient for testing purposes, and will need to be replaced by a full key-distribution algorithm in a production environment.

The signatures are generated just before messages are sent to the controller, and after they have been converted into a byte array for sending (as the sign function requires a byte array as input). The returned signature is also a byte array. The signature and message are sent to the controller in a single request, with the signature sent first. This signature is sent first for easier parsing, as it is of a constant size.

On receiving a message from the controller, the parser is responsible for extracting signatures and verifying against the received message. Whereas in the unsigned version the OpenFlow library can be provided with a byte array and extract multiple OpenFlow messages, this is not possible when signatures are also included within the array. The parser instead first extracts the signature, and then manually reads the length of the OpenFlow packet from the packet header (stored in the second 2 bytes of the header, and so 2 bytes from the end of the signature bytes in the byte array). Using the OpenFlow length, the bytes of the current message are extracted, and the signature is then verified. If the signature is verified, then the reader provided by the OpenFlow library is used to extract

the OpenFlow message. The next signature and message are then extracted and verified. This process is repeated until all messages have been extracted, and verified, from the received data.

6.3.5 OpenFlow Message Handling

The primary activity of the proxy is the handling of OpenFlow messages. The proxy parses these messages using the OpenFlow java library, and then performs actions on them depending on the specific OpenFlow message type. As part of this, the proxy needs to maintain state regarding switch and controller requests. Two objects are used for this purpose:

SwitchEvent A switch event represents a request from the switch. It stores the Xid of the PacketIn message, a copy of the PacketIn message, and two ArrayLists to store the returned PacketOut and FlowMod messages.

ControllerRequest A controller request represents a request from a controller to a switch. These are primarily used during the switch-controller handshake phase. A controller request stores the controller id, OpenFlow packet type, and Xid of the controller request.

The proxy maintains a collection of each of these. The collections are formulated as HashMaps, with the message Xid as the key, and an instance of a SwitchEvent or ControllerRequest as the value. A HashMap is used due to the efficient value retrieval compared to other collection types. In order to reduce memory usage, once all replies to a request have been received, the SwitchEvent or ControllerRequests is deleted from the relevant HashMap.

6.3.5.1 Switch to controller communication

The proxy need to ensure that communication from the switch is handled correctly. Introducing multiple primary controllers means that the proxy needs to handle certain switch requests to prevent multiple replies coming back from the multiple controllers, which can cause the switch to reset its connection to the proxy.

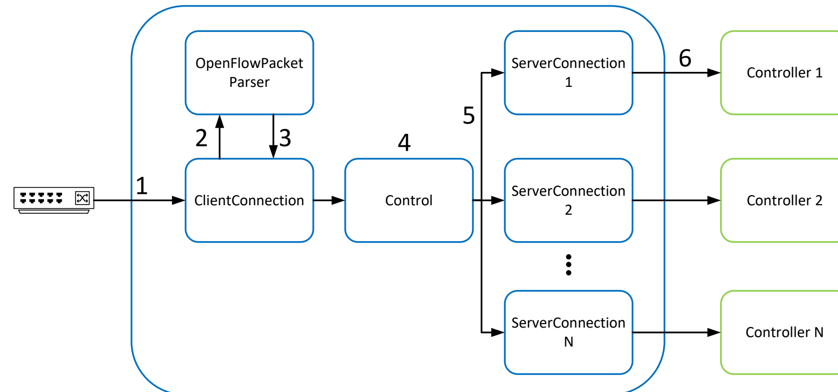


Figure 6.2: Proxy switch to controller message handling

Figure 6.2 shows the typical flow of OpenFlow messages from the switch to the controller. The flow of messages is as follows:

1. The switch sends a request to the controller through the proxy. A single request may include multiple OpenFlow messages. The `ClientConnection` class reads the message from the socket into a byte array.

2. The `ClientConnection` class calls the parse method in the `OpenFlowPacketParser`, which receives a byte array, and uses the OpenFlow Java libraries reader methods to extract the OpenFlow messages. If signatures are used, these messages are signed.

3. The `OpenFlowPacketParser` returns a list of OpenFlow messages. The `ClientConnection` loops through these messages. For any `Hello` and `EchoRequest` messages, a response is generated and sent back to the switch. Any other messages are forwarded to the master `Control` class for processing.

4. The `Control` class processes the messages. For specific message processing by OpenFlow packet type, see below.

5. After processing each message, the `Control` class calls the send method in the `ServerConnection` instance matching the controller (or set of controllers) the message is destined for.

6. The `ServerConnection` writes any messages destined to the controller it represents onto the socket, and the message is sent to the controller.

   Specific message processing for OpenFlow message types:

PacketIn On receiving a `PacketIn` message from the switch, the proxy creates a new SwitchEvent, with a copy of the PacketIn message. The Xid field of the `PacketIn` message is set to a new value generated by the proxy. This is used to link switch requests to responses from the controller.

EchoRequest and Hello If the switch sends an `EchoRequest` or `Hello`, the proxy will generate a reply and send it to the switch directly.

Reply packets Messages replying to controller requests, such as `FeaturesReply` or `RoleReply` messages, are matched by Xid to a controller request, and forwarded only to the controller that sent the matching request. To aid in this, the proxy maintains a lookup matching OpenFlow request message types to their relevant response types.

Other packets All other packets are sent on to the set of controllers without any further processing.

The majority of messages from the switch are sent to all controllers. However, I found that Floodlight does not handle multiple or unexpected replies to certain request messages, primarily those exchanged during the OpenFlow handshake, often disconnecting the switch. To correct this, specific requests from the controller are logged by the proxy by Xid, and replies from the switch are only sent to the controller that sent the matching request.

6.3.5.2 Controller to switch communications

Figure 6.3 shows the typical flow of messages from the switch to controller. Note that a single `ServerConnection` is shown, when in reality there is one instance per controller. Each instance follows the same process, so I only include one into the diagram for clarity. The flow of messages is as follows

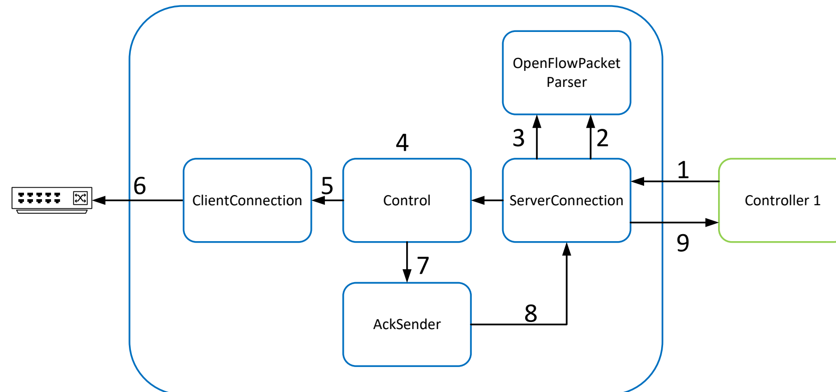


Figure 6.3: Proxy controller to switch message handling

1. The controller sends a packet to the switch, consisting of one or more OpenFlow messages. The `ServerConnection` reads the message from the socket into a byte array.

2. The `ServerConnection` class calls the parse method in the `OpenFlowPacketParser`, which receives a byte array, and uses the OpenFlow Java libraries reader methods to extract the OpenFlow messages. If signatures are used, these are verified.

3. The `OpenFlowPacketParser` returns a list of OpenFlow messages. The `ServerConnection` loops through these messages. As is also the case for the switch connections, for any `Hello` and `EchoRequest` messages, a response is generated and sent back to the controller. Any other messages are forwarded to the master `Control` class for processing.

4. The `Control` class processes the messages. For specific message processing by OpenFlow packet type, see below.

5. After processing each message, the `Control` class calls the send method in the `ClientConnection` instance.

6. The `ClientConnection` writes any messages destined to the switch it represents onto the socket, and the message is sent to the switch.

7. (Optional) If the message from the controller is a response to a switch request, the `Control` class generates an acknowledgement message, and forwards to the `AckSender`.
8. (Optional) If batched acknowledgements are in use, the `AckSender` adds the acknowledgement to a buffer, and then forwards onto the appropriate `ServerConnection` to be sent to the controller. If batched acknowledgements are not used, the message is not buffered and forwarded directly to the `ServerConnection`.
9. (Optional) The `ServerConnection` sends the acknowledgement to its controller.

Specific message processing for OpenFlow message types:

PacketOut The proxy will match the `PacketOut` to a switch event, and will store the message in switch event object. If the number of `PacketOut` messages stored in the switch event match the number of controllers, the `PacketOut` is forwarded to the switch.

FlowMod As with `PacketOut` messages, the proxy will wait for a full set of matching `FlowMod` messages before sending one to the switch.

Requests Request messages, such as `FeatureRequest` or `RoleRequest` messages, are logged by the proxy with the request type, sending controller and Xid. This allows for replies to be matched to requests.

6.4 Controller modification

As mentioned above, the Floodlight controller was chosen as the controller used for testing the SDBFT system. Floodlight was chosen partly due to the authors familiarity with Java, therefore providing an easier pathway to understanding the architecture. It was chosen over other Java based controllers, such as ONOS and OpenDaylight, due to it being a smaller code base therefore making modifications easier without causing unknown effects.

While being a well-known controller, there is limited documentation which made locating parts of the code base required for modification difficult. For example, locating the classes responsible for receiving and parsing OpenFlow messages was difficult and was only achieved by tracing the flow of OpenFlow messages through the controller backwards from the Forwarding application.

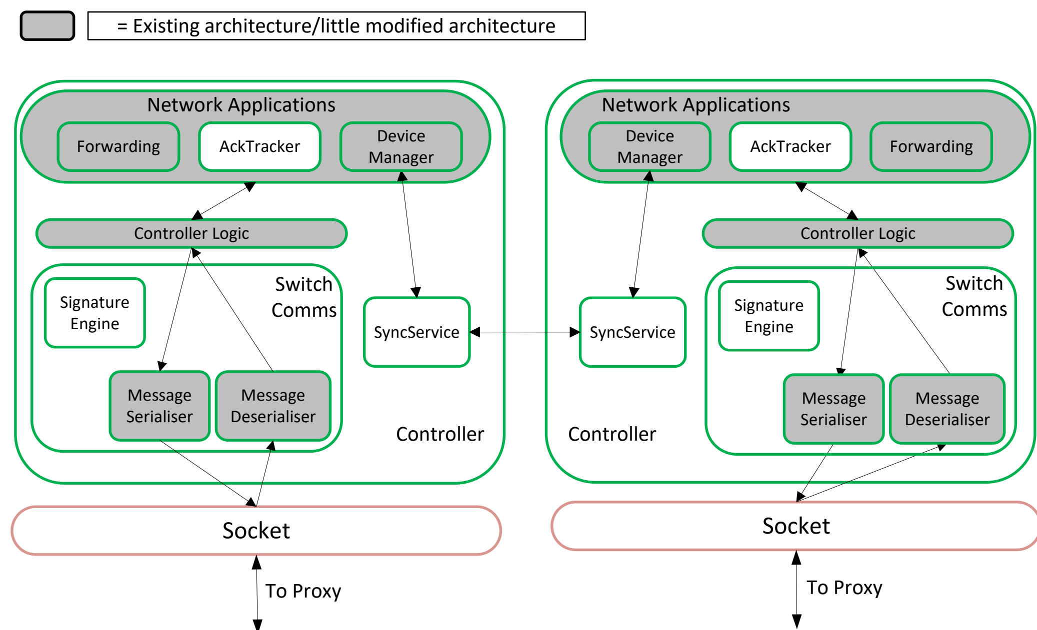


Figure 6.4: Controller system model

Figure 6.4 provides a high level overview of the SDN controller architecture. This model only shows the core components that were modified to support the SDBFT protocol. Elements highlighted in grey required only minimal modification compared to the base implementation. The only additional elements that were added were the AckTracker application, and the additional message serialiser and deserialiser to handle signed messages. All other modifications were achieved through the use or implementation of existing services provided by the controller.

6.4.1 Message serialising/deserialising

Floodlight uses Netty sockets in order to communicate with switches. The Netty library requires a serialiser and deserialiser interface which converts a ByteBuffer to a list of Objects, and vice versa. In Floodlight, these two classes are `OFMessageEncoder` and `OFMessageDecoder`. These classes were duplicated and modified to produce versions which sign messages destined for the switch (`OFMessageEncoderSign`), and verify messages sent from the switch (`OFMessageDecoderSign`). By incorporating signatures here, no further modification is required to the core Floodlight controller. The controller chooses the signed versions, or the original unsigned versions, depending on the current configuration, as specified by the controller properties file.

6.4.2 Xid setting

Openflow packets feature a Xid field, which is used to identify messages and match requests to responses. The default Floodlight applications do not set this field when sending responses to the switch. As I need to match controller responses to a particular request so I can ensure grouping, I must ensure the controller sets this field to match the request it received. Therefore, I went through any applications which communicate with the switch in response to `PacketIn` messages, and ensure they set the Xid field on any responses they send to match that of the incoming `PacketIn`.

6.4.3 Acknowledgement handling

For ease of implementation, switch acknowledgements are sent using OFError messages. OFError messages are used as they support writing arbitrary data into the message, which can then be parsed by the controller. This required modifying the error handling mechanism of the controller. Normally, error messages are not handled by applications on the controller (though applications can register to receive them), instead the controller, upon seeing an error message, enters an error state. The controller was modified to inspect error messages, and if

the message is of the acknowledgement format (where the message begins with `###ACK###`), to push them to the application layer and not enter an error state. The AckTracker application can then read these messages. Note that for the purposes of testing, the AckTracker simply reads the acknowledgement messages. In future versions, the AckTracker could write these acknowledgements to a log file or similar.

The OpenFlow protocol officially supports custom, deployment specific messaging through the `Experimenter` message type (`Vendor` in version 1.0 of the protocol), which allows messages containing arbitrary data for vendor specific use. This would have been a better implementation approach, however this message type has not been properly implemented by Loxigen, and so is not included in the OpenFlow Java library used by both the proxy, and the Floodlight controller. The intention is that protocol definitions used by Loxigen would be updated by a vendor. It is unclear how this would impact the Floodlight controller, in part as the internal controller logic sorts messages by type, so I decided to use the similar OFError message type which are already handled by the controller, with the small modification to the controller to support my particular use case.

6.4.4 Synchronisation

It is important that the backup controllers that will form the expanded quorum when a failure is detected know the state of the devices connected to the switch before they take over. As a simple example, if the backup controllers do not know which switch port a device is connected to on receiving a PacketIn message, it will flood the PacketOut through all active ports on the switch, and not install a flow rule. This prevents consensus from occurring as the controllers of the existing quorum know the topology of devices whereas the backups do not. In order to implement the publisher–subscriber model as described in Section 5.7, I utilise the experimental `SyncService` built into Floodlight. The `SyncService` provides the ability for controllers to writes to a datastore, and publish the updates to other controllers, in a publisher-subscriber model. This functionality can be built into any module. One module that uses this service is the `DeviceManager`, which maintains a record of devices attached to switches under its control. By default,

this service pushes an update to the datastore for every new device that it sees, however it will only pull from the datastore if the controllers is moving from the SLAVE to the MASTER state for a given switch. I modify this module to subscribe to receive updates from other controllers, and update its own datastore. This means that the controller will know the correct switch topology as soon as is required. In order to allow this to function, the switch connects to the controllers of the backup quorum, as well as the primary, on initialisation, and completes the setup handshake, but sends no further packets to the backups until a fault occurs.

This functionality could be extended to provide a simple synchronisation method to allow all controllers within to learn the topology of the full network, to aid in routing. As the `SyncService` operates on a publisher-subscriber model, then it would only requires all controllers to sign up for updates for all other controllers. Currently, the `DeviceManager` requires a handshake with a switch in order to store information about the devices connected to it. The two options are that either the switch can complete a handshake with every controller, which would allow the controller to store the relevant information received from other controllers, or alternatively the DeviceManager could be modified to incorporate a datastore of switches and their connected devices which have not completed the handshake, but can be used by the other services (such as the TopologyManager) which use this information.

6.5 Implementation of Comparative System

In order to provide a comparison to related work, a version of the system was constructed which makes use of a traditional BFT algorithm for providing fault-tolerance. Rather than implementing this from scratch, this was achieved through the use of the BFT-SMaRt library [22, 24]. BFT-SMaRt was previously applied to the SDN control scenario by ElDefrawy and Kaczmarek [76], and is the only example from related work which has been fully tested for performance (although code is not available). The protocol requires 5 rounds of communication, and

$2n + 3n^2$ messages for n replicas. The BFT-SMaRt protocol is discussed in more detail in Section 2.11.2.

BFT-SMaRt provides a Java library for providing a fault-tolerant system, with the provisioning for both replicas (servers), and clients. Within the library, multiple instances of a server are configured (as the system uses a PBFT approach, four ($3f+1$) instances are required as a minimum). BFT-SMaRt was implemented as an extension to the SDBFT proxy, with both a client-side and server-side proxy being implemented.

6.5.1 Configuration

The BFT-SmaRt library uses its own set of configuration files for configuring the BFT system. The two primary files are the `system.config` and `hosts.config` files. The `system.config` file is used to configure the parameters of the BFT-SmaRt instance, such as the IP address to bind to, the number of replicas in use, the use of signatures, and whether to handle byzantine faults, or just fail-stop faults.

The `hosts.config` file contains a list of all replicas within the system. For each replica, the file contains an ID number, the IP address, and the port number which the replica is to run on. The ID number is used when launching a server replica, and the IP address of the host must match the IP address provided in the file. The replicas to be used by a client are configured in the `system.config` file by providing a list of the ID numbers. Errors in this file prevent the BFT-SMaRt library from functioning.

The BFT-SMaRt proxy also utilises the SDBFT configuration file, as seen in Appendix A.1

6.5.2 Protocol

One issue with adapting BFT-SMaRt for use with floodlight, is that the utility type communication between the switch and controller, in particular the handshake, needs to be done on an individual basis between the switch and each controller. To allow this, two connections are used. The first connection utilises

portions of the SDBFT proxy to handle handshake messages. This connection is used to send any messages which do not require use of the BFT protocol.

The second connection is the connection provided by the BFT-Smart library, built on Netty sockets. This connection carries all messages which need to be handled by the BFT protocol, primarily `PacketIn` messages and the associated responses. This connection is not accessed directly, rather the BFT-SmaRt library provides an abstraction layer over the communication.

BFT-Smart can handle requests both in synchronous and asynchronous mode. In synchronous mode, on sending a request the code must then wait for a reply before proceeding. This is not suitable for a situation where multiple flows are coming into the switch in short succession. In the asynchronous mode, on submitting a request to the server a response handler is set up which then waits for the reply. This means that queueing happens on the server side, whilst allowing the switch to send multiple requests in short succession and not lock. I only make use of asynchronous requests in my implementation.

6.5.3 Proxies

Two proxies are used to allow the BFT communication. A client proxy sits in front of each switch, acting as the client for the BFT-SMaRt library. A server proxy also sits in front of each controller, and is implemented as a BFT-SMaRt server replica. It was decided to utilise a server side proxy rather than modify the floodlight controller to natively utilise the BFT-SMaRt library as it would require a large amount of modification to the controller architecture.

Figure 6.5 presents an overview of the BFT-SMaRt proxy architecture. In this example, 2 controllers are used. Dashed lines represent components that are implemented fully within the BFT-SMaRt library.

6.5.3.1 Client Proxy

A BFT client proxy is deployed for each switch within the network. A single client proxy is responsible for the communications of a single switch, meaning there should be as many instances of the client proxy as there are switches.

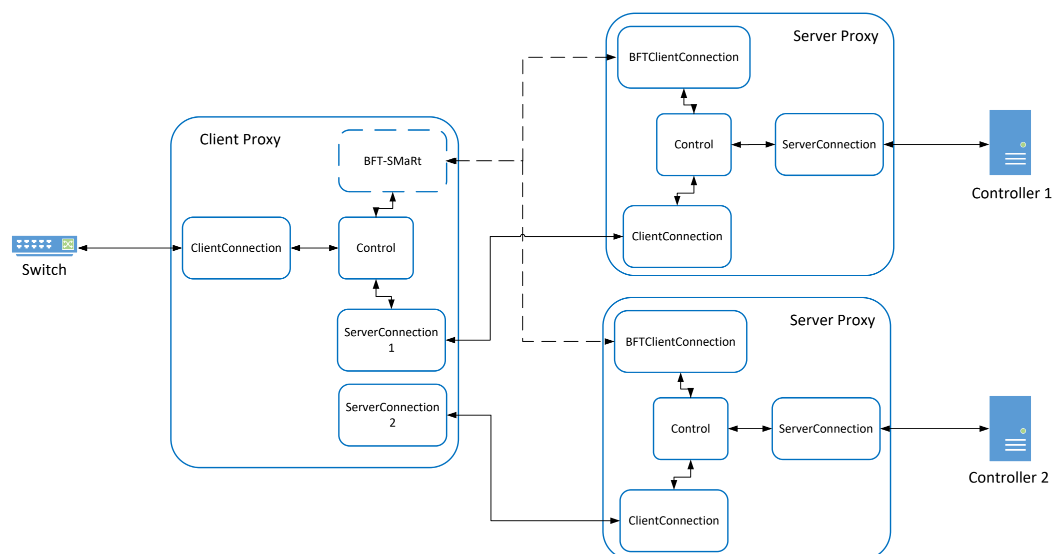


Figure 6.5: BFT-SMaRt proxy architecture

A client proxy will talk to just one switch, but multiple server proxies. The client proxy is very similar to the proxy used within the SDBFT system, with the primary difference that on receiving a packet from the switch, the proxy will either forward it directly to the controller proxy using a TCP connection (for packets which should not be processed using the BFT-SMaRt system), or through the BFT-SMaRt system for packets which should be processed by the BFT-SmaRt system. Largely, any packet other than a `PacketIn` messages bypasses the BFT-SMaRt system.

The proxy parses all packets it has received over a TCP connection, either from the switch, or through the TCP connections to the server proxies. This is only used to direct the packets to the correct communication method for forwarding based upon their OpenFlow type. No other processing is performed on the packets.

The client makes use of a `AsynchServiceProxy` object from the BFT-SMaRt, which allows the application to submit asynchronous requests to the BFT replicas. Asynchronous requests are used as it prevents the client from blocking whilst waiting for a response from the sever (it is expected that the switch will often have to deal with multiple simultaneous new flows that need to be processed in quick suc-

cession). On establishing the `AsynchServiceProxy`, a client id is provided, which is user supplied in the proxy properties file. This is used to identify clients on the server side. The request is submitted through use of the `invokeAsynchRequest()` function of the `AsynchServiceProxy`, which also requires the provision of a listener to process the response. I provide a `BFTReplyListener`, which collects the responses from the servers through the implementation of an inherited `replyReceived` method, and checks that there are a sufficient number of matching responses, `q`, as defined in Listing 6.1. In this calculation `getCurrentViewN()` returns the number of replicas, and `getCurrentViewF()` the handled number of faults. N will equal $3F + 1$. The implementation of the `ReplyReceived` method can be seen in Listing 6.2. It is worth noting that the listener does not need to wait for a response from every controller before returning a result to the switch. As long as the number of matching responses exceeds the threshold, on receiving enough matching responses, the listener forwards the response onto the switch, and clears the request, closing the listener. Responses are in the form of byte arrays, and are matched using the Java `Arrays.equals(a,b)` utility.

```
int q = Math.ceil((double) (serviceProxy.getViewManager().getCurrentViewN() +
    serviceProxy.getViewManager().getCurrentViewF() + 1) / 2.0);
```

Listing 6.1: BFT-SMaRt threshold calculation.

```
ArrayList<TOMMessage> responses;
@Override
public void replyReceived(RequestContext context, TOMMessage reply) {

int sameContent = 1;
for (TOMMessage tomMessage : responses) {
  if (Arrays.equals(tomMessage.getContent(), reply.getContent())) {
        sameContent++;
  }
}
responses.add(reply);
if (sameContent >= q) {
```

```
  control.sendToClient(reply.getContent(), reply.getContent().length);
  serviceProxy.cleanAsynchRequest(context.getOperationId());
}
}
```

Listing 6.2: BFT-SMaRt ReplyListener message verification

The proxy reads two lists of server addresses, which should be the same. The first is in the SDBFT properties file, which is used for setting up the list of TCP connections to controller proxies.

The second list is defined in the BFT-SMaRt `hosts.config` file, where each server is represented by a replica id and an IP address. The number of servers is defined in the `system.config` file, which specifies the number of servers, number of handled failures (the number of servers must be greater than $3f + 1$), and the list of replica ids to connect to.

6.5.3.2 Server Proxy

A BFT server proxy is deployed for each controller within the network. A server proxy talks to a single controller, but will accept connections from multiple client proxies representing multiple switches. As with the client proxy, it is based upon the SDBFT proxy with modifications to support the BFT-SMaRt library. The major challenge with the server proxy is handling connections from multiple switches at the same time, and ensuring that responses from the controller are forwarded to the correct switch.

As with the client proxy, the server proxy supports two different communication channels with each client proxy (one TCP and one BFT-SMaRT).

On receiving a connection from a new client proxy, the server proxy will setup a new socket connection to the controller. For each connected switch, the server proxy will maintain an individual connection to the controller. The controller expects that each switch will connect on its own socket, so this helps ensure the controller operates as normal. The socket from the client and socket to the controller are linked, so that any packets can be sent between the two. For example, if client A connects to the server proxy over a TCP connection $A - SP$,

then a connection $A - C$ is set up between the server proxy and the controller. Any packets coming in on $A-SP$ are then forwarded onto $A-C$, and any packets coming to the proxy on $A-C$ are forwarded onto $A-SP$. This removes the need to processing these packets on the server side, as communication on this channel should be sent directly to the controller, and responses straight back to switches.

On setting up the TCP connection, the first packet sent by the client contains the BFT client ID set by the user, which is read by the server proxy. This is used to populate a HashMap, with the BFTClientID as keys, and the ServerConnection created for that client as a value. This allows packets that come in over the BFT system to be sent to the controller over the previously established controller connection for that client.

The BFT-SMaRt connection is provided through implementing a class that extends `DefaultSingleRecoverable` from the BFT-SMaRt library, which is called by the library on receiving a message. The class can implement two methods: `appExecuteOrdered` and `appExecuteUnordered`, which are called depending on whether the request was sent from the client as an ordered or unordered request. In this system I only use ordered messages, and so the `appExecuteOrdered` method is implemented.

On receiving a request over the BFT system, the method first extracts the BFT client id from the request. The server proxy then creates an empty ByteBuf and stores in a HashMap, with the requests packet XID as the key. This is where the response from the controller will be stored. The request is then forwarded to the controller, and the method then enters a loop, checking for that buffer to be filled. On each loop, there is a test for either the buffer containing two openflow messages, or a single `PacketOut` message. Floodlight will reply to a `PacketIn` either with both a `PacketOut` and a `FlowMod` packet if the destination is known, or just a `PacketOut` with the `flood` action type if the destination is unknown. The proxy parses responses and checks the action field on the packet out, setting a flag if the output action is to flood. As the BFT-SMaRt library expects the full response to a request to be sent in a single response, I must ensure that all responses from the controller are gathered by the server proxy before forwarding to the client (the controller send `PacketOut` and `FlowMod` packets separately).

6.5.3.3 Other Elements

Signatures As with the SDBFT proxy, signatures within BFT-SMaRt are also provided using the `java.security` package. The signature handling within BFT-SMaRt is hidden from the user. Signatures are enabled through the `system.config` file, where the user is also able to configure the specific signature algorithm.

System State The BFT-SMaRt library maintains a view of the replicas through the use of a `system.currentview` file. This file contains the state of the replicas, and is used to reduce startup time. The issue with this file is that if the set of replicas change, then the setup stage of the BFT-SMaRt protocol can fail as the current real-world state is different to the state stored int his file. This is solved by deleting this state file whenever any changes are made to the set of replicas.

6.6 Conclusion

In this chapter, I provide an overview of the practical implementation of the SDBFT protocol, through the development of the SDBFT proxy, and the deployment of a modified instance of the Floodlight SDN controller. I also provide the details of the implementation of a comparative system using a traditional BFT approach, built using the BFT-SMaRt library. I use these implementations as the basis of the evaluation in Chapter 8, using a number of testbed configurations discussed in the next chapter.

Chapter 7

Experimental Setup

In this chapter I give an overview of the testbed setups and tools I make use of to evaluate the performance of SDBFT. Three different experimental setups are used, two virtual and one physical. The two virtual setups consist of a virtual network consisting of virtual machines and OpenVSwitch virtual switches to emulate a network setup similar to that found in a data centre, and a Mininet virtual network used for testing more complex network topologies. Finally, I also test with a commercial, off-the-shelf physical SDN switch to measure real world performance. A summary of these testbeds can be seen in Table 7.1.

7.1 Testbeds

7.1.1 OpenVSwitch (OVS) Virtual Environment

I evaluate the system using a virtualised environment that resembles a simple datacentre setup. The hardware consists of a server running the KVM hypervisor and Open vSwitch (OVS) version 2.7. OpenVSwitch (OVS) in an open source, production quality virtual switch designed for use in virtual environments, including cloud deployments[1]. As a particular example, OVS is the core networking

[1] `https://www.openvswitch.org`

Table 7.1: Summary of testbeds

Testbed	Type	Size	Equipment/Software
OpenVSwitch Virtual Environment	Virtual	1 Switch, 4 Hosts	CirrOS virtual hosts, OpenVSwitch virtual switches, Floodlight controller in Ubuntu Server 20.04 VM (1 per instance)
Mininet	Virtual (Simulated)	1-10 Switches, 2 Simulated Hosts	OpenVSwitch virtual switches, emulated hosts managed by Mininet, All Floodlight controller instances running on host.
Physical Switches	Physical (Hardware)	3 Switches, 4 Hosts	3 Dell EMC PowerSwitch S3048-ON switches, 4 Raspberry Pi 4 hosts, Floodlight controller instances running on physical server

component used in the open source OpenStack cloud infrastructure platform[1]. OVS supports many protocols, but of particular importance to me is its support of the OpenFlow protocol which allows it to function as a SDN switch. The server has 4 AMD Opteron 6376 16 core 2.3GHz CPUs, with 1TB RAM, and runs Centos 7. Controllers are run inside headless Ubuntu Server virtual machines with 4 vCPUs and 4GB RAM. The control and data planes are separated, such that the controllers are connected to a Linux Virtual bridge deployed using `ip link add br0 type bridge`, with virtual hosts connected to OVS virtual bridges (which act as our SDN switches). The SDBFT proxy sits between the OVS bridge and the Linux bridge.

In order to maximise server capacity an extremely lightweight Linux distribution, cirrOS[2], was used to create host virtual machines for baseline testing. Originally designed for testing Openstack installations, it contains all basic OS functionality, including ping and ssh support, which is ideal for our use case.

Tests are launched using a bash script, which uses SSH to run commands on the controller VMs and cirrOS host VMs to start controller instance and perform pings between hosts. An example of one of these scripts can be found in

[1] `https://www.openstack.org`
[2] `https://launchpad.net/cirros`

Appendix B.1.

Since version 2.5, OVS has not worked with the Floodlight controller due to extra fields sent along with PacketIn messages which Floodlight is unable to parse, so OVS version 2.7 was patched to not send this extra data. The switches were configured to use OpenFlow version 1.3 in all tests.

Baseline Setup An overview of the baseline experiment setup is provided in Fig. 7.1. This setup consists of a basic topology of a single switch (an OVS bridge) with four connected virtual hosts (cirrOS vm). The OVS bridge connects to the controllers through a Linux virtual bridge, except for when a proxy is in use, in which case this sits between the OVS bridge and Linux bridge, running directly on the host OS. There are 10 controller VMs deployed within the system to allow for testing with quorum sizes of 1 to 10 controllers.

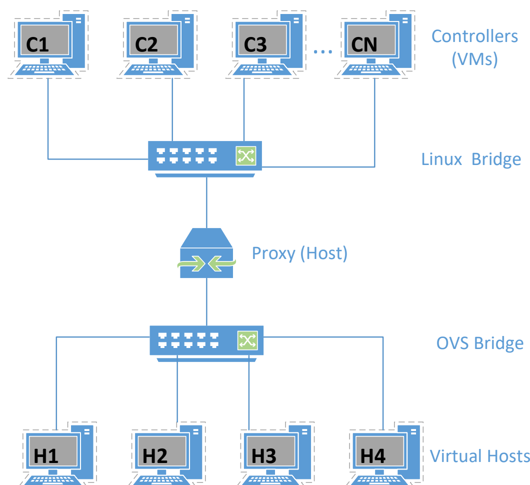


Figure 7.1: Baseline Setup

7.1.2 Mininet

Mininet is a python-based network simulation tool, specifically designed for experiment with software-defined networking [130]. Mininet deploys virtual switches

using OpenVSwitch, and can launch virtual *hosts*. The hosts are not virtual machines, rather they are spawned processes that can perform basic actions such as pings or file transfers. Whilst Mininet can be launched using a simple command line utility, deploying a basic topology of switches and hosts, the preferred approach is to write Python scripts which specify switches, hosts, the connections between them and the actions that hosts can take. The advantage of Mininet is that complex topologies can be deployed and torn down within seconds, far easier that with the manual approach taken above.

To enable automation when running experiments, Mininet was configured using Python scripts. An example of such a script, for launching a single switch, single controller network with two hosts, and performing pings between the hosts, is available in Appendix B.2. Note that this assumes that any controller instances are already launched.

To run a complete experiment, a simple Bash script was used, an example of which is available in Appendix B.3. In this example four instances of the Floodlight controller are launched, along with an instance of the SDBFT proxy. The Mininet Python script is then run. This bash and python script handles a complete test run, and can be repeatedly launched using a looping script to repeat the experiment the required number of times.

Mininet 2.2.2, utilising OpenVSwitch 2.9.8, is installed on Ubuntu Server 18.04.5 LTS. This is running on a Dell Precision T7610 workstation, equipped with two Intel Xeon CPU E5-2650 processors (8 core, 16 thread, 2.0-2.8 GHz per CPU) CPUs, along with 128GB RAM.

There is a slight issue when using Mininet with the SDBFT proxy. Whilst a normal SDN switch will only make one connection to the controller, Mininet actually makes two connections to the controller. Mininet will first make a connection to the controller when adding a controller to the network using the `net.addController` function (as can be seen in Appendix B.2), and then the switch will connect to the controller once the controller is linked to the switch using the `switch.start` function. As the proxy expects only one connection from the switch at a time, this prevents the OpenFlow handshake from completing. To prevent this, when using Mininet as a test platform, the first connection from a *switch* is ignored, and the connection to the controllers is only initialised on the

second connection from the switch. This applies to both the SDBFT proxy, and the BFT-SMaRt proxy.

7.1.3 Physical Switch

I also evaluate SDBFT with a physical, commercial SDN switch. I use a Dell EMC PowerSwitch S3048-ON 1000BASE-T 48-port 1GbE top-of-rack (ToR) switch[1], which features support for SDN using OpenFlow (versions 1.1 and 1.3), and can operate with 3rd party controllers and operating systems (the *ON* portion of the model number represents *Open Networking*). The switch is running Dell EMC Networking OS9 (specifically 9.13), and has been configured to use OpenFlow 1.3. Configuration is carried out using a serial connection to the management port of the switch for initial setup, and then through telnet to start and stop OpenFlow instances.

The switch can simultaneously run as an SDN switch, and traditional switch, depending on the specific configuration. To utilise SDN, OpenFlow instances are launched, with either individual ports, or VLANs, assigned to that instance. Any traffic over those ports or VLANS is then controlled using SDN. I use the OpenFlow instance in VLAN mode — I create a VLAN and attach it to the OpenFlow instance, and add the required ports to that VLAN. The OpenFlow instance is assigned a remote controller, and when the OpenFlow instance is started it will complete the handshake with the controller. When the OpenFlow instance is stopped, the connection to the controller is closed and communication will no longer happen between devices on the assigned VLAN.

Our physical setup is visible in Figures 7.2 and 7.3. Specifically, I use three switches, S1, S2 and S3. Each one has ports 1-6 assigned to a vlan, which is connected to the OpenFlow to the network, with 2 connected to S1, and one each to S2 and S3. There is also a connection from S1-S2 and S2-S3 within the OpenFlow VLANS. I call this the core network, forming a line topology. The first switch has two further port groups assigned to two separate VLANS. The first is used for orchestrating the Pi hosts over SSH. The second of these operates

[1] `https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-S3048-ON-Spec-Sheet.pdf`

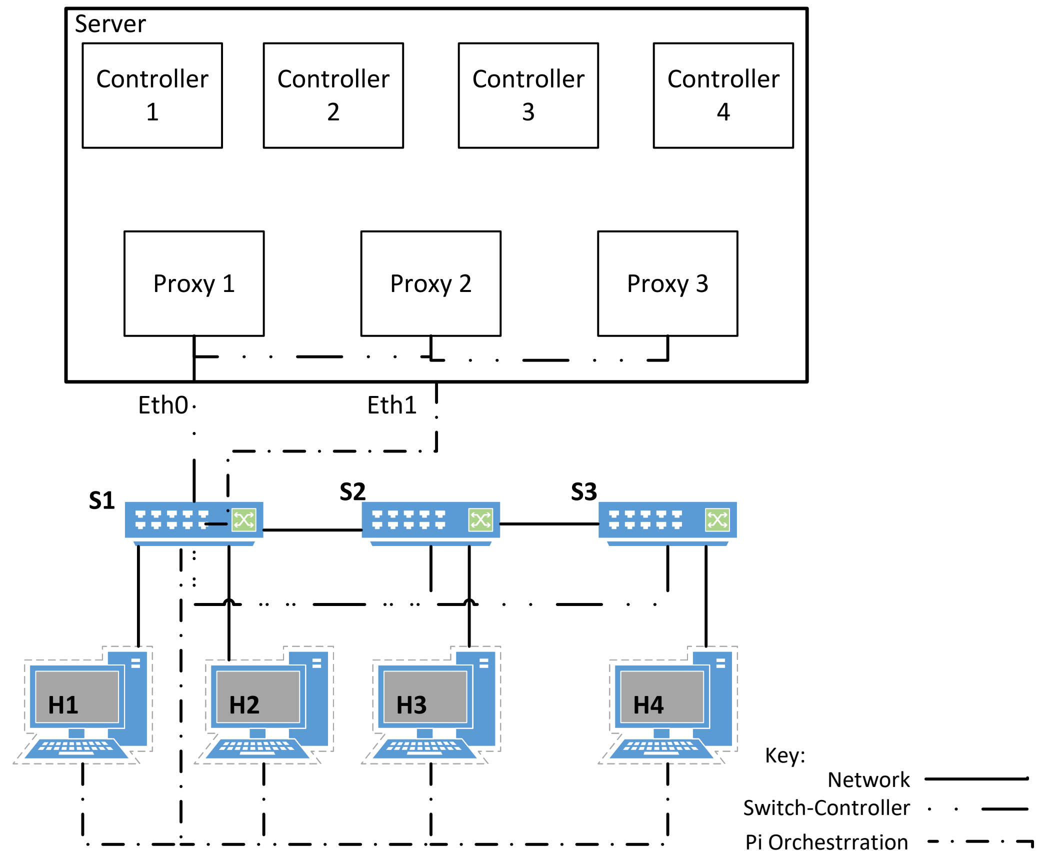


Figure 7.2: Physical Network Setup Design

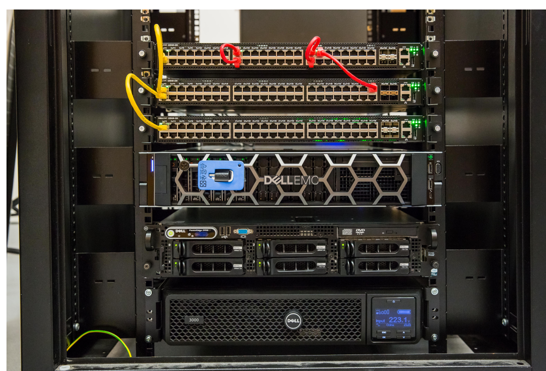


Figure 7.3: Physical Network Hardware

as a management network for the switches, with each switches management port

connected into this port group. The switch management port is used for the connection to the SDN controllers. The server has two connections into the testbed, one for accessing the management network, and one for accessing the Pi network. The three subnets (core, Pi orchestration and controller management) are all assigned their own /24 subnet, with all IP addresses manually configured.

The server is a Dell PowerEdge R740 with two Intel Xeon Silver 4114 2.2G, 10C/20T CPUs, along with 192GB of RAM. Networking to the switches is through a Broadcom 5720 Quad Port 1GbE ethernet card. The server is responsible for running both proxy and controller instances. The SDN switches OpenFlow instances are instructed to connect to a controller at the servers IP address on the management network, with the port specifying which controller or proxy instance each switch connects to.

7.2 Simple TCP Proxy

In order to measure the impact of introducing a proxy between the switch and controller, I make use of a simple TCP proxy to provide baseline results. The simple TCP proxy is based on the SDBFT proxy, however all packet processing has been removed and so it only receives a messaged from the client, and forwards it onto the server, and vice versa. The simple proxy is designed to only work with a single client and server, and so, as is also the case with the SDBFT and BFT-SMaRt proxies, an instance should be deployed for each switch in use. As the proxy is limited to a single server connection, it can only connect to a single controller and so all tests using the simple proxy use a single controller.

7.3 Measurement Tools

In order to capture experimental data the ping and Cbench tools re used for testing the network performance of the SDN network and to benchmark the SDN controller architecture.

7.3.1 Ping

The primary tool used to evaluate the performance of SDBFT and the comparative BFT-SMaRt implementation is through use of the standard `ping` tool that is distributed with most modern OSs. For the OVS and physical switch setups, this is the ping client that is installed on the cirrOS virtual hosts on the virtual testbed and Raspbian hosts as used with the physical switch. For the Mininet setup, it is the ping tool that comes as part of Mininet. Ping is used for this as it is a simple tool for testing connectivity between two hosts, and also outputs packet latency for each individual packet sent, the first packet displaying the flow setup time.

The output from the ping command was redirected to text files, which are then parsed using a simple Java program to extract the latency values for the required packets. For the majority of tests, this is the first packet as is the packet which experiences delays due to flow setup, however for some tests I extract the latency values for all subsequent packets.

7.3.2 Cbench

Cbench is an OpenFlow controller benchmarking tool available as part of the OFLOPS framework[1] for benchmarking OpenFlow switches [196]. Written in C, Cbench launches a number of basic virtual switches (the default is 16, however this is a tuneable parameter). In both cases a large number of `PacketIn` messages are sent to the controller, and then the virtual switch waits for either a `PacketOut` or `FlowMod` response. The latency and throughput modes differ as follows:

Latency Mode In latency mode, the switch sends a single `PacketIN` to the controller, and waits for the response. On receiving the response, it will then send another `PacketIN`, and repeat this process measuring how many requests can be processed sequentially by the controller per second.

Throughput Mode In throughput mode, the switch fills a buffer (initial size 65535) with `PacketIN` messages, sends them all to the controller and sees how

[1] `https://github.com/mininet/oflops`

many responses are returned per second. The primary difference to latency mode is that all the `PacketIN` messages are sent immediately without waiting for responses between each packet. This test measures how many requests can be handled concurrently, and in particular shows how well multi-threading is implemented on the controller.

An example of Cbench running in throughput mode against a simple controller can be found in Listing 7.1. In this example, 16 tests of 1 second each are run, with the first discarded as a 'warmup' test. Only a single virtual switch is used. The final output is an average of 208.39 responses/second, meaning that the controller can, on average, handle 208 flows per second. Running against the same controller with 16 switches results in a slightly lower average of 188.5 response/second, representing 10-13 flows per switch per second. this reduction is due to overheads on the controller in handling a large number of requests form multiple switches, as every switch sends 65535 packets at the same time.

As Cbench is designed to test a single controller, I had to modify the code slightly to be able to test with more than a single virtual switch, as each switch is required to connect to a different instance of the SDBFT proxy. To achieve this, I hardcoded an integer array within the `cbench.c` class file that sets up a connection to the controller for each virtual switch. Instead of using the default 6653 or user supplied port, the function will instead use the port number defined in the array in the position matching the number of the virtual switch.

```
$ ./cbench -p 6653 -s 1 -t
cbench: controller benchmarking tool
   running in mode 'throughput'
   connecting to controller at localhost:6653
   faking 1 switches offset 1 :: 16 tests each; 1000 ms per test
   with 100000 unique source MACs per switch
   learning destination mac addresses before the test
   starting test with 0 ms delay after features_reply
   ignoring first 1 "warmup" and last 0 "cooldown" loops
   connection delay of 0ms per 1 switch(es)
```

```
  debugging info is off
08:59:55.581 1   switches: flows/sec:  198   total = 0.197892 per ms
08:59:56.684 1   switches: flows/sec:  204   total = 0.203424 per ms
08:59:57.785 1   switches: flows/sec:  200   total = 0.199931 per ms
08:59:58.888 1   switches: flows/sec:  197   total = 0.196416 per ms
08:59:59.992 1   switches: flows/sec:  212   total = 0.211152 per ms
09:00:01.096 1   switches: flows/sec:  205   total = 0.204187 per ms
09:00:02.200 1   switches: flows/sec:  212   total = 0.211216 per ms
09:00:03.302 1   switches: flows/sec:  208   total = 0.207486 per ms
09:00:04.404 1   switches: flows/sec:  214   total = 0.213602 per ms
09:00:05.508 1   switches: flows/sec:  215   total = 0.214332 per ms
09:00:06.609 1   switches: flows/sec:  216   total = 0.215735 per ms
09:00:07.710 1   switches: flows/sec:  208   total = 0.207893 per ms
09:00:08.812 1   switches: flows/sec:  215   total = 0.214482 per ms
09:00:09.916 1   switches: flows/sec:  209   total = 0.208249 per ms
09:00:11.024 1   switches: flows/sec:  205   total = 0.203434 per ms
09:00:12.127 1   switches: flows/sec:  215   total = 0.214319 per ms
RESULT: 1 switches 15 tests min/max/avg/stdev = 196.42/215.74/208.39/5.70
   responses/s
```

Listing 7.1: Example output of Cbench controller benchmarking tool. The '`-p 6653 -s 1 -t`' parameters instruct the tool to connect to a controller running on localhost port 6653, creating a single virtual switch and running in throughput mode. Note that by default, the first test is discarded as a 'warmup'

7.4 Floodlight Configuration

I deploy instances of the modified floodlight controller discussed in Section 6.4. The primary modifications to the controller are that applications that respond to switch requests set xID fields in the responses to match the requests (which is not done by default), and when using the signed SDBFT protocol, an alternate message serialiser and deserialiser that can sign and verify messages is used. When

using a direct controller connection or the simple TCP proxy, these changes have no impact on performance. When using the OVS setup described in Section 7.1.1, Floodlight instances are launched in individual virtual machines with unique IPs, and all run OpenFlow on port 6653. When using the Mininet (Section 7.1.2) and Physical (Section 7.1.1) testbeds, multiple Floodlight instances are deployed onto the host, with sequential OpenFlow ports.

7.4.1 Applications

There are two primary routing applications that come as part of the Floodlight package that I use for testing purposes: the `LearningSwitch` and the `Forwarding` applications.

Learning Switch The `LearningSwitch` is a simple L2 learning-switch application, a version of which can be found on many simple SDN controllers such as NOX and RYU. On receiving a switch request as a `PacketIn` message, the application will either forward the packet onto the next hop switch on the path to the destination (if the port that leads to the destination is known), otherwise the packet is flooded out all enabled ports. For both types of forwarding, a `PacketOut` message is used. If the outbound port leading to the destination is known, a flow rule is also installed into the switch table using a `FlowMod` packet. The `LearningSwitch` will only forward and install a flow rule on a single switch at a time. This has the downside that for a new flow, the controller needs to be contacted by every switch on the path if suitable flow rules do not exist within their tables. The benefit to consensus SDN is that this is a purely reactive approach: a switch sends a request to the controller, and only that switch receives a response.

Forwarding The `Forwarding` application is the routing application loaded as part of the default Floodlight configuration. The Forwarding application attempts to build the full path to the destination when receiving a new flow through a `PacketIn` message. This application uses the `TopologyManager` application,

which maintains information about the network topology and can compute shortest paths between pairs of nodes using Dijkstra‘s algorithm. On receiving a `PacketIn` message, the `Forwarding` application will query the `TopologyManager` for the route to the packets destination. If one exists, then the application will forward the packet out of the appropriate port (using a `PacketOut` message), and then attempt to install flow rules on each switch along the route that it controls, to allow the packet to be forwarded without querying the controller at each hop. The benefit of this application is that it is far more efficient and results in reduced flow setup times, as the controller can pre-load the path onto all required switches. The downside to this is that this then means that the applications becomes partially proactive from the perspective of the subsequent switches on the path, which will receive `FlowMod` without having seen a packet on the flow, which then has to be handled by the proxy.

7.5 Conclusion

In this chapter I have given an overview of the experimental setup used to evaluate the implementation of the SDBFT protocol and the comparative BFT-SMaRt implementation. I discuss the various testbeds used, along with the measurement tools used to gather data.

In the next chapter, I present the results of our evaluation of the SDBFT protocol using the setup discussed in this chapter.

Chapter 8

Evaluating The SDBFT Controller Architecture

8.1 Introduction

In this chapter I perform an experimental evaluation of the implementation of the SDBFT system (outlined in Chapter 6), comparing against the BFT-SMaRt approach described in Section 6.5 and a traditional SDN model with a switch connected to and controlled by a single controller instance. Testing is performed using the various testbeds described in Chapter 7.

As well as collecting a set of baseline results on the performance of the SDBFT protocol, I also apply further tests to evaluate the real-world performance of the SDBFT protocol, and analyse the benefits and drawbacks versus the BFT-SMaRt implementation. This includes introducing faulty controllers into the network, performing a stress test with a controller benchmarking tool and deploying the protocol with a physical, commercial switch.

Our evaluation consists of the following tests, (summarised in Table 8.1):

Baselines This set of tests measure the baseline performance of the SDBFT protocol in a single switch topology, versus the traditional SDN model and BFT-SMaRt protocol. I measure performance with an increasing number of

controllers forming the primary quorum, as well as measuring the performance impact of introducing signed messages into the protocol.

Multi-Hop Path Tests I measure how well SDBFT can handle multiple switches and routing flows across them.

Failure Operation I introduce compromised controllers into the network to evaluate the SDBFT failover protocol, and compare this to the fault-tolerant BFT-SMaRt protocol.

High Throughput Benchmark I use the Cbench controller benchmarking tool to measure how well the SDBFT system performs under load.

Testing on Physical Switch I replace the virtual switches used in previous tests with physical, commercial SDN switches to evaluate how well SDBFT works when controlling physical switches and hosts.

Deployment of Hardware Proxy I deploy the SDBFT proxy onto a Raspberry Pi to emulate a hardware proxy and measure the impact on performance when running the proxy on a low cost, relatively low powered device.

Network Traffic Load I run the SDBFT and BFT-SMaRt protocols and capture the network traffic generated by the two protocols. I measure the difference in network load that arises from the two protocols.

8.1.1 Method of Analysis

Each experiment in this evaluation shows summary results of repeated tests (for most tests $n = 50$), with median (η), mean (μ) and standard deviation (σ) presented for each. In the majority of tests I measure the flow setup time in milliseconds, measured through the use of a ping request and extracting the round trip time of the first packet. A baseline is generated using a traditional single controller setup. This is standard practice with the literature as it can be used to directly measure the additional overhead of the modified control plane by measuring the difference in the round trip time between the baseline and modified controller. I test for statistical significance of key results.

Table 8.1: Summary of experiments

Experiment	**Testbed**	**Purpose**
Baseline	OVS	Baseline performance, effect of signature use
Multi-hop Path Tests	Mininet	Performance over longer network path lengths
Failure Operation	OVS	Ability to handle failures and impact on performance
High Throughput Benchmark	None (Cbench tool)	Ability of solution to handle large volume of requests (controller load)
Physical Switch	Physical	Deployment of solution to physical switch and verify performance on real-world hardware
Hardware Proxy	Physical	Viability of proxy to physical hardware (bump-in-the-wire) for existing switches
Traffic Volume	OVS	Network overhead (number of packets) of solution

I make use of one of two methods to compute statistical significance. Where the test results are non-normally distributed (determined by visual inspection of the data combined with application of the Shapiro-Wilk's test [213]), an independent-sample Mann-Whitney U test is used [144]. If the data is normally distributed, then an independent t-test is used. In both cases, the result is classed as significant if $p < 0.05$. Both tests measure if there is a statistically significant difference between two sets of samples. If $p > 0.05$, then the two samples are said to be not statistically significantly different from each other, and therefore equal.

When using the Mann-Whitney U test, effect size is measured using the standardised effect size, computed as

$$\frac{z}{\sqrt{n_1 + n_2}}$$

where z is the z-statistic outputted by the Mann-Whitney U test, and n_1 and n_2 are the size of the two samples. The z-statistic reflects the difference between two samples — a negative value indicates that elements of the second sample are on average greater than the first. When using the independent-variable t-test, effect size is computed using the Cohen's d measure of the standardised difference

between two means, which assigns a value from 0 to infinity [53]. Cohen's d is computed as:

$$\frac{Mean_{Sample1} - Mean_{Sample2}}{Pooled \quad Standard \quad Deviation}$$

Effect size classifications, drawn from the literature, are shown in Table 8.2 [53, 222].

Effect Size	Classification
<0.2	None (S)
0.2-0.49	Small (S)
0.5-0.79	Moderate (M)
0.8-1.29	Large (L)
>1.3	Very Large (VL)

Table 8.2: Cohen‘s d effect size classification [53, 222]

8.2 Baselines

This set of experiments is designed to measure the core performance of the SDBFT proxy, enabling comparison against the traditional SDN model and the BFT-SMaRt system. These experiments are broken down into four main tests:

Traditional Model This simple test replicates the traditional SDN model of a single switch communicating directly with a single controller. This provides a baseline to compare against.

Effect of Proxy I use a simple TCP proxy to measure the impact of a proxy between the switch and controller, without performing any processing on messages other than forwarding. This measures how much additional latency is introduced through my decision to implement the additional switch side processing as a proxy. By measuring this extra latency, I can estimate performance of the SDBFT system if it were implemented directly into the switch.

Effect of Multiple Controllers In this test, I measure the performance of the SDBFT system, as well as BFT-SMaRt, with an increasing number of controllers forming the primary quorum. This measures how well the system scales to a greater number of replicas to provide tolerance to a larger number

of faults. Testing SDBFT with a single controller allows me to measure the impact of the extra packet processing that occurs on the proxy.

Signature Use In this final test, I add signatures to messages between the switch and controller and measure the impact on performance. Cryptographic signatures are computationally expensive and so this test measures how much extra latency is introduced through signature use.

8.2.1 Setup

I perform these tests using the OVS-based virtual network (as discussed in 7.1.1). I use a simple topology of four virtual hosts connected to a single OVS bridge. This topology can be seen in Figure 8.1. In each test, I perform a ping between hosts H3 and H4 to *warmup* the controller (the first flow setup after launching the controller is substantially slower than for further flows). I then perform two pings between hosts H1 and H2, logging the flow setup time for the second flow. I only log the second flow because during the first flow setup the controller is still learning which ports devices are connected to, which provides unreliable results for flow setup times in the normal case. These two pings are performed six seconds apart, because the default flow rule expiration on inactivity set by Floodlight is 5 seconds. For all tests the default `Forwarding` routing application (as discussed in 7.4) is used by Floodlight.

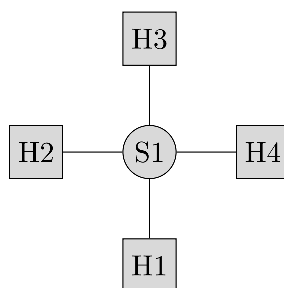


Figure 8.1: Simple network used for baseline testing

The specific setups for each of the four baseline tests are as follows:

Traditional Model The switch is connected directly to a single instance of the Floodlight controller.

Effect of Proxy I deploy the simplified TCP proxy (described in Section 7.2) which is connected to a single controller.

Effect of Multiple Controllers I use the full SDBFT and BFT-SMaRt proxy setups and connect to multiple controllers. For SDBFT, I are able to test with 1–10 controllers. For BFT-SMaRt, I test with 4–10 controllers (4 is the minimum number required for the protocol to function). Each controller is running inside its own virtual machine. Messages are unsigned in this test. BFT-SMaRt is run in both the default ordered mode, where client request order is maintained and the full BFT protocol is run. Unordered mode, where the server replicas simply return the result to the client, is also used. I test with unordered mode because BFT-SMaRt uses Netty sockets rather than the standard Java sockets used in SDBFT. Netty sockets, unlike Java sockets, are non-blocking. The primary difference between blocking and non-blocking sockets is that blocking sockets hold up execution whilst waiting to read data from the socket. Non-blocking sockets by contrast allow execution to continue, utilising a listener to process incoming data. This has benefits for programs with multiple concurrent connections.

Signature Use Finally, I use the SDBFT and BFT-SMaRt proxies with signed messages. For SDBFT, I use the signed version of the proxy, whilst the Floodlight controller uses modified message serialiser/deserialisers. For BFT-SMaRt, signatures are activated by setting the `system.communication.useSignatures` option '1' in the `system.config` file. Because both the SDBFT and BFT-SMaRt use the Java Security signature system, I can use the same algorithm specification for each. I test using the `SHA512withRSA` option (which uses a SHA512 hash signed using RSA with a 1024 bit key), which is the default option for BFT-SMaRt. I also test using the `SHA256withRSA` option, which uses a SHA256 hash signed using RSA with a 512 bit key. I test both options because the differing key sizes have substantially different impact on latency, representing a tradeoff between greater cryptographic security vs performance. SHA512 with 1024 bit RSA is the strongest signature provided by the Java Security library. RSA 512 is

breakable within a few hours and for $100 when using the cloud[1], therefore is shown for comparison only. Key generation uses a fixed seed, allowing all parties to compute the same set of keys for signing. Whilst this is not at all secure, it is sufficient for allowing me to measure the impact of signature use.

8.2.2 Results

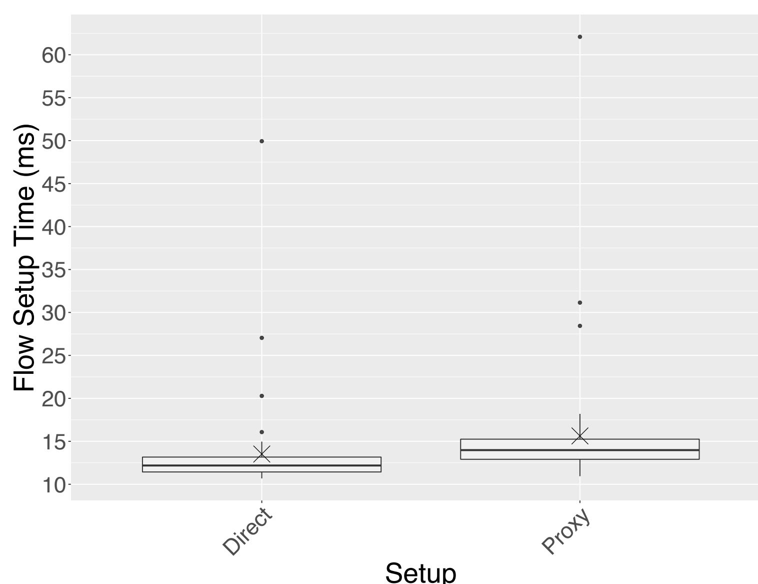


Figure 8.2: Flow setup time for a direct controller connection, and through a simple TCP proxy. X=mean.

Traditional Model A single *direct* controller achieves a median flow setup time of of 12.18ms (μ=13.53ms σ=5.87), shown in Figure 8.2 and Appendix C.1. Table C.1). There is one particularly large outlier of 49.939ms. I believe that these are caused by the Floodlight controller (a large, complex, multi threaded Java application) — the controller regularly communicates with the switch for liveness tests and to broadcast LLDP traffic, which if occurring at the same time as the flow setup test can result in a small amount of additional latency.

[1] `https://github.com/eniac/faas`

Effect of Proxy Figure 8.2 shows a slight increase in flow setup time when introducing a simple TCP proxy (η=13.98, µ=15.62ms ,σ=7.59), a statistically significant increase over the direct controller connection (Mann-Whitney U: $U = 578, z = -4.6292, p < 0.00001$, effect $= 0.46(S)$), however the effect size is small, so further testing may be needed to establish if this difference is truly consistent over a larger testing regime.

Effect of Multiple Controllers In Figure 8.3 I present the results of SDBFT, when using non-signed messages (see Appendix C.1 Table C.1 for complete results and statistical significance tests compared to the direct controller connection). All controller counts represent a statistically significant increase over the direct controller connection (see Appendix C.1 Table C.1). For a single controller, the median flow setup time is 15.75ms (µ=16.05ms, σ=2.01), increasing to 28.92ms (µ=30.0ms, σ=8.32) for 10 controllers. For a single controller, this represents a statistically significant increase over a direct controller connection with a moderate effect size (Mann-Whitney U: $U = 2286, z = -7.13857, p < 0.00001$, effect $= 0.71(M)$). This is also a statistically significant increase over the simple TCP proxy, although with a small effect size (Mann-Whitney U: $U = 616, z = -4.3672, p < 0.00001$, effect $= 0.44(S)$), suggesting that the additional processing required by SDBFT has only a minimal impact. This is as expected, as the additional processing required by SDBFT for a single controller is minimal. Appendix C.1 Table C.2 shows that each additional controller adds minimal extra latency, with each additional controller introducing a statistically significant (but small effect) increase over the previous one. This is as expected, as the primary source of latency is the SDBFT proxy. It is believed that the additional latency per extra controller comes from an additional socket write to send to each controller, and the deserialising of the response to process the OpenFlow messages and check all responses match. The controllers themselves can process packets in parallel and, in the absence of signatures, perform no additional operations on requests compared to a standard deployment. The amount of variance does increase with additional controllers, as shown by the standard deviation values for each test. This is likely due to variance in network latency on the proxy-controller connections, as well as the variance in a single controller responding to

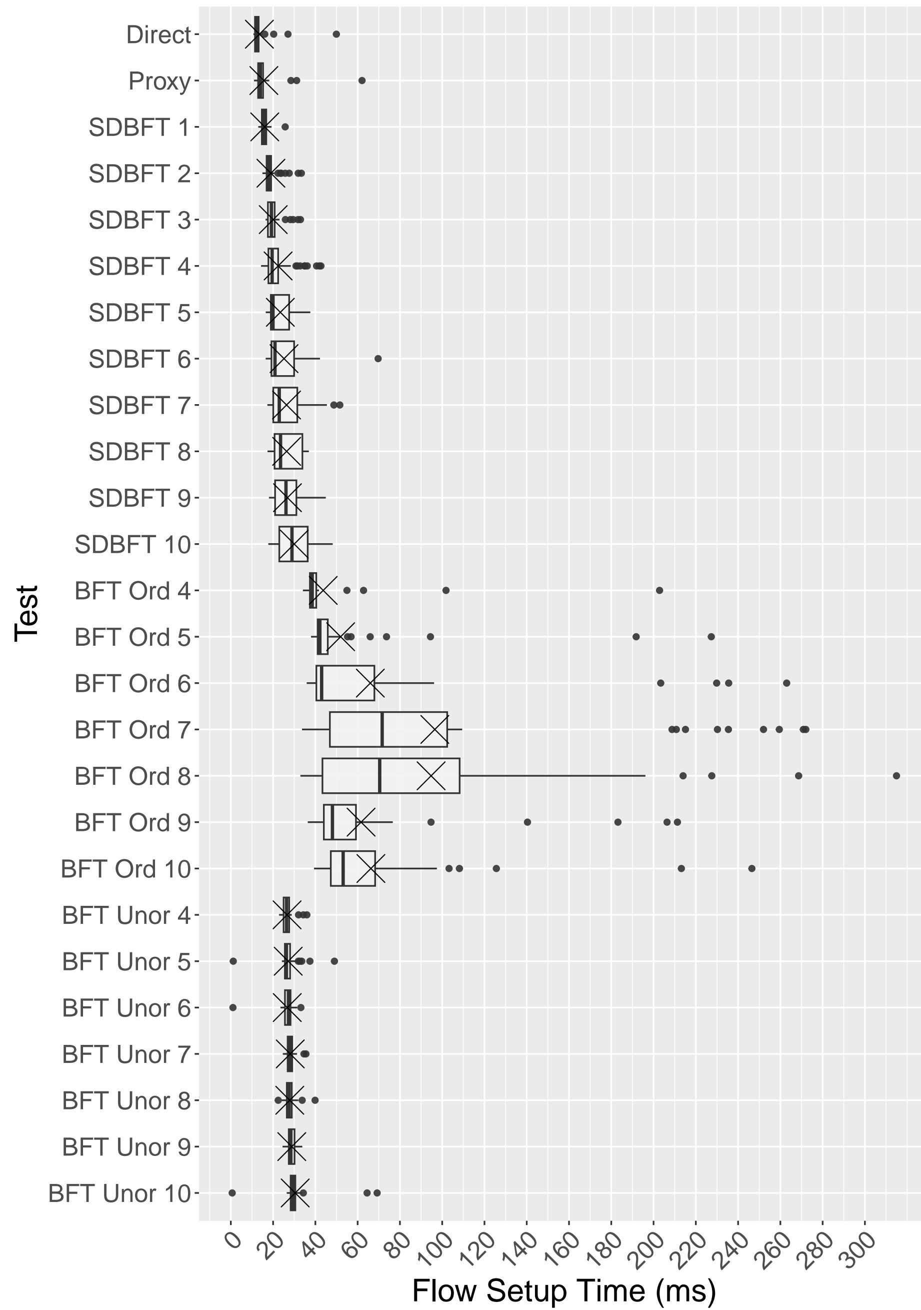


Figure 8.3: Unsigned protocol flow setup time (ms) for 2–10 controllers in quorum using SDBFT vs. 4–10 controllers using the BFT-SMaRt protocol in ordered and unordered mode. X=mean. 50 repetitions.

messages (which can be seen in the direct controller connection results presented in Figure 8.2). All controllers need to return a response in order for the protocol to complete, therefore additional latency on any single controller will result in a slower flow setup time; a greater number of controllers increases the chance that this may occur.

The results for BFT-SMaRt are presented in Figure 8.3. All tests show a statistically significant increase over the direct controller connection, with a large effect size (see Appendix C.1 Table C.1). When using 4 controllers in ordered mode, the median setup time is 38.5ms (μ=43.7ms, σ=25.03), increasing to 53.14ms (μ=66.27,σ=38.98) when using 10 controllers. This is a substantial increase over SDBFT with a large effect statistical significance, as seen in Appendix C.1 Table C.3. For example, if I compare this to SDBFT when using 4 controllers the difference is statistically significant with a moderate effect size (Mann-Whitney U: $U = 143, z = -7.62815, p < 0.0001$, effect $= 0.76(M)$), supporting my finding that BFT-SMaRt results in significantly higher flows setup times than SDBFT. The amount of variance is also substantial when using BFT-SMaRt. This is in particular noticeable for 7 and 8 controllers, where the median setup times are 71.61ms (μ=96.51ms,σ=71.33) and 70.44ms (μ=94.75ms, σ=67.26) respectively. This variance is largely caused by a number of outliers, as can be seen in Figure 8.3. This large amount of variance makes a direct comparison more difficult, although as show in Appendix C.1 Table C.3, this is still statistically significant against SDBFT (with a large effect size).

When using BFT-SMaRt in unordered mode, which forgoes the full BFT protocol, the results are closer to those of SDBFT. The median flow setup time increases only slightly from 26.39ms(μ=26.61ms,σ=2.49) when using 4 controllers, up to 29.5ms (μ=30.45ms, σ=8.72) when using 10 controllers. This is as expected. Firstly, BFT-SMaRt uses Netty sockets, which are very efficient for broadcasting, meaning there little extra latency introduced through sending requests to extra controllers. Secondly, the system only needs to receive a majority of matching responses in order to reach consensus, and pass the result onto the switch. It is worth noting that BFT-SMaRt is intended to be used in ordered mode only, and so this data is presented for comparison of the sockets used only.

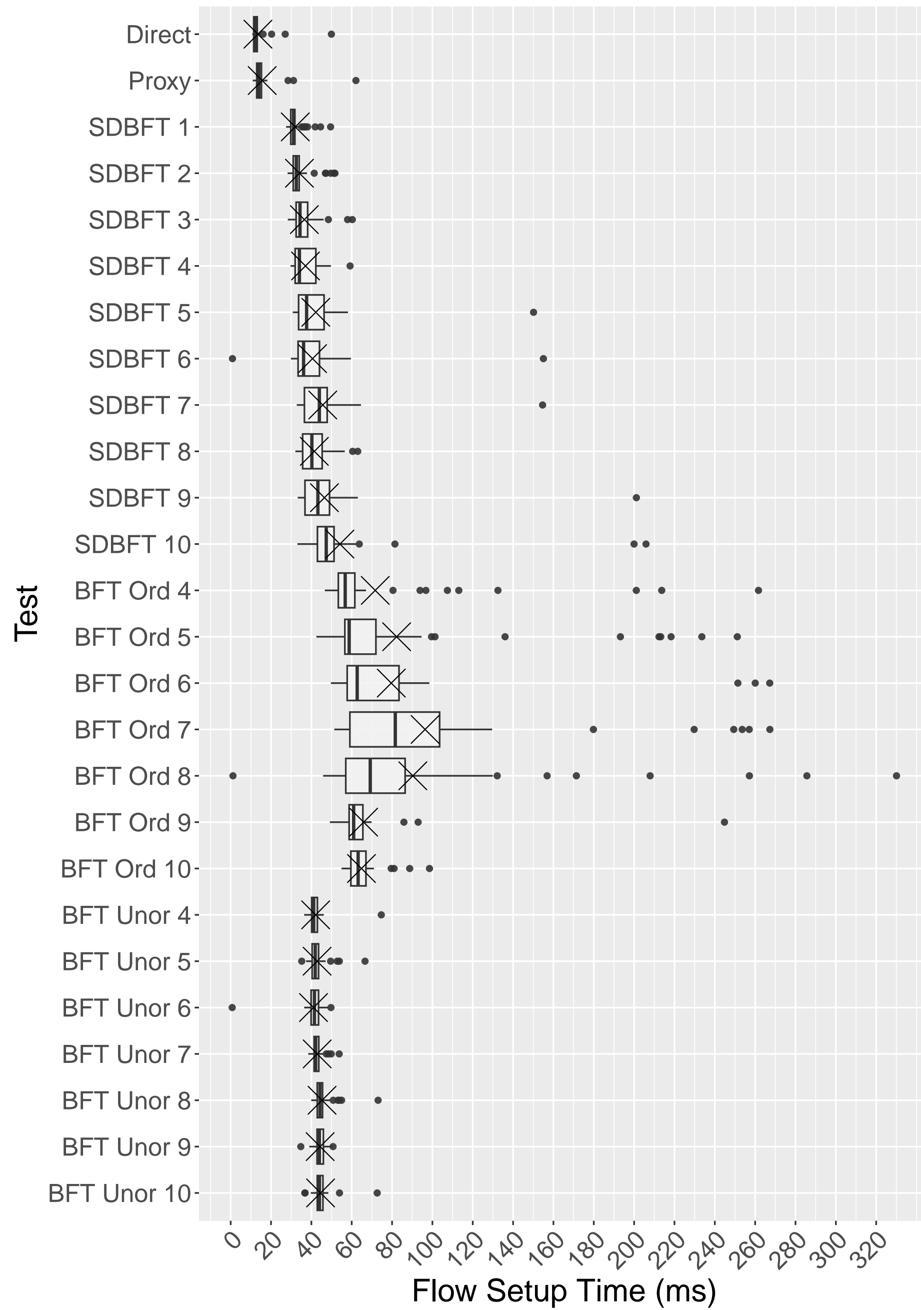


Figure 8.4: Signed protocol using `SHA512withRSA` flow setup time (ms) for 2–10 controllers in quorum using the SDBFT approach vs. 4–10 controllers using the BFT-SMaRt protocol in ordered and unordered mode. X=mean. 50 repetitions.

Signature Use Figure 8.4 presents the flow setup times for SDBFT when using the `SHA512withRSA` algorithm in the Java Security library, which uses RSA with a 1024 bit key to sign SHA512 hashes of the message. Appendix C.1 Table C.4 contains full results and statistical significance tests against a direct controller connection, whilst Table C.6 show statistical significance against the unsigned protocols with equivalent controller counts. For SDBFT, the median flow setup time for a single controller is 31.18ms (µ=32.02ms, σ=4.19), increasing to 47.37ms (µ=54.16ms, σ=31.74) when using 10 controllers. Both are statistically significantly different to the direct controller connection with a large effect size (see Appendix C.1 Table C.4). The increase in median between 1 and 10 controllers when using a signed connection (31.18 vs 47.37) is a smaller relative increase compared to the unsigned version (15.75 vs 28.92). When using signatures, signing is the more expensive operation, which is only performed once per request by the proxy before forwarding to the controllers. When the controllers are signing the responses, this happens across all of them in parallel and so there is little additional impact through using additional controllers.

Similarly, the results for ordered BFT-SMaRt show an impact when using signatures, as presented in Figure 8.4. The median flow setup time ranges from 56.77ms (µ=71.66ms, σ=42.83) up to 63.18ms (µ=64.83ms, σ=8.32) for 4 to 10 controllers respectively. These are interesting results. There is an immediate effect when using 4 controllers, with a statistically signifiant increase with moderate effect size when using signatures over the unsigned version (Mann-Whitney U: $U = 147, z = -7.59354, p < 0.0001$, effect $= 0.76(M)$). However, with an increasing number of controllers, this impact reduces and in fact for 8 and 10 controllers the flow setup time is lower than when not using signatures. The results for 7 (Mann-Whitney U: $U = 1081, z = -1.16161, p = 0.2454$, effect $= 0.12(N)$) and 8 (Mann-Whitney U: $U = 1155, z = -0.65147, p = 0.5147$, effect $= 0.07(N)$) show no statistical significant difference over the unsigned versions. The use of signatures is managed within the BFT-SMaRt library, with just the algorithm configured to use signatures through the configuration file. As with the SDBFT case, server replicas can perform signatures in parallel, and so, as with SDBFT the introduction of additional controllers should have minimal impact. This result, along with the unsigned results, indicates there may be some issue with the

BFT-SMaRt library which introduces additional latency when using more than 4 to 5 replicas, which masks the additional latency caused by signature use.

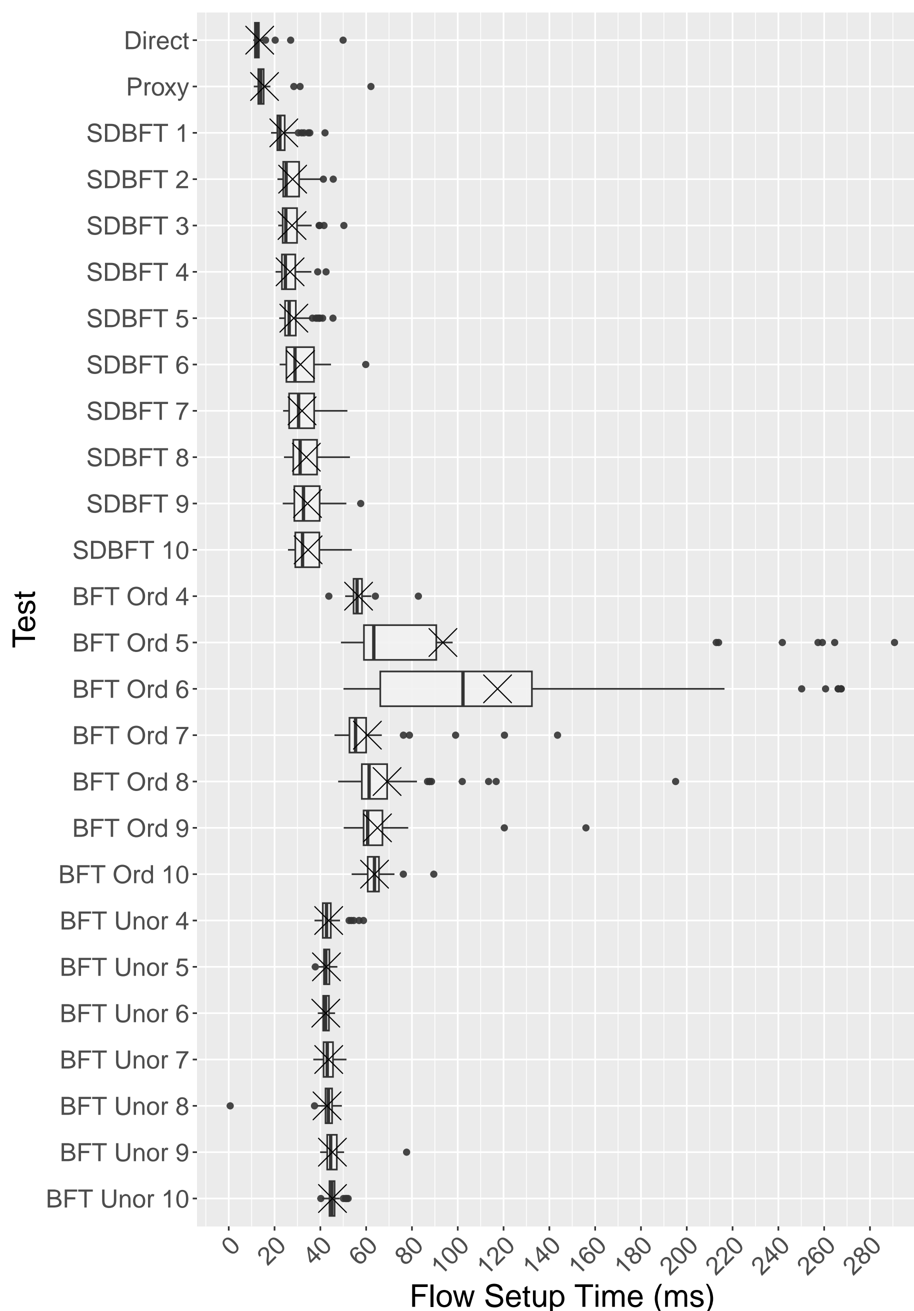


Figure 8.5: Signed protocol using `SHA256withRSA` flow setup time (ms) for 2–10 controllers in quorum using the SDBFT approach vs. 4–10 controllers using the BFT-SMaRt protocol in ordered and unordered mode. X=mean. 50 repetitions.

Figure 8.5 shows the performance when using the less secure `SHA256withRSA` algorithm, which uses RSA with 512 bit keys to sign SHA256 hashes. Appendix C.1 Table C.5 contains full results and statistical significance tests against a direct controller connection, whilst Table C.7 show statistical significance against the unsigned protocols with equivalent controller counts. In the case of SDBFT, whilst all tests show the addition of signatures results in a statistically significant differences in flow setup time, as the number of controllers increases, the effect size decreases from large to small. Interestingly, increasing from 2 to 3 and 3 to 4 controllers actually results in a slight reduction in median setup time (24.92ms vs 29.74ms), however this is not statistically significant (3 to 4 Mann-Whitney U: $U = 1323, z = -0.4998, p = 0.6172$, effect $= 0.05(N)$). It is likely this is caused by the outliers visible in Figure 8.5 for 2 and 3 controllers, skewing the distribution of the data and inflating means.

BFT-SMaRt, as shown in Figure 8.5, experiences a similar reduction in additional latency when using 4 controllers using the smaller key size (with a median flow setup time of 56.03ms (μ=56.9ms, σ=5.06)), however makes little difference when using 10 controllers, with a median flow setup time of 63.64ms (μ=63.66ms, σ=5.87). In the unordered case, the result are very similar to the SHA512withRSA case (for example, the Mann-Whitney U result for 10 controllers between SHA256 and SHA512 is $U = 1202.5, z = -0.324, p = 0.7459$, effect $= 0.032(N)$, which shows no statistically significant difference between the two), further confirming that the use of signatures within BFT-SMaRt has a minimal impact on latency.

8.2.3 Discussion

Effect of Proxy I find that the addition of the proxy alone results in a small, but significant increase in latency compared to a direct connection. This is a median flow setup time of 13.98ms. When I use the unsigned SDBFT proxy with a single controller, this median flow setup time increases to just 15.75ms. From this, I can infer that the additional processing when using the SDBFT proxy only introduces 1-2ms extra latency, with the use of a proxy responsible for a substantial proportion of the additional delay over a direct connection. From

this I can reason that if the SDBFT protocol were to be built directly into a switches firmware, then the flow setup time when using SDBFT could be much closer to a direct controller connection. This isn't true of BFT-SMaRt where a minimum of 4 controllers are used, with a median setup time is 38.5ms, an almost 25ms increase over the simple proxy, compared to around 7ms when using SDBFT with 4 controllers.

Effect of Multiple Controllers When increasing the number of controllers, both SDBFT and BFT-SMaRt scale well. However, I find that even when using 10 controllers, SDBFT outperforms BFT-SMaRt using just 4 controllers.

I do find some unusual results when using SDBFT, in that when using signatures with 4 and 8 controllers (when using SHA512withRSA) and 4 controllers (when using SHA256withRSA) both result in a reduction in flow setup time when compared to 3 and 7 controllers respectively. It is unclear why this occurs, however it is worth considering that across all of these tests, the difference between a single and 10 controllers is relatively small across each test, with only a small difference in adding each additional controller (as shown by significance tests in Appendix C.1 Table C.8), and so normal variance could account for these anomalies.

Signature Use I find that, even when using the stronger SHA512withRSA encryption, the SDBFT proxy performs well, increasing flow setup time by approximately three times compared to a direct controller connection. Using the less secure SHA256withRSA halves this increase. This is as expected, because the signing time for SHA256withRSA is around half that of SHA512withRSA. To demonstrate this, if I sign a 32 character string 1000 times using SHA512withRSA with a 1024 bit key, the mean signing time is 0.378ms, and mean verifying time just 0.027 ms. The signing time is reduced to 0.179ms for signing when using the SHA256withRSA signing algorithm with a 512 bit key. The verification time remains constant across both algorithms, with signing being the far more expensive operation. Note that these examples are from a test run on a Macbook Pro 2.3 GHz Quad-Core Intel Core i5. As well as the actual signing action itself, when using signatures the SDBFT proxy and controller signed message deserialiser have

to perform extra processing on received packets. The proxy and controller receive byte arrays, and have to manually read the OpenFlow packet length before extracting the OpenFlow message bytes and signature bytes before the OpenFlow library can parse the message, and the signature can be verified. This is an expensive operation which is not required when not using signatures and so likely accounts for a proportion of the additional latency. To verify this, I measure the time taken to complete the parsing on the both the unsigned and signed implementations. Note that on the signed version, this includes the verification of signatures. If I take a simple message, an `EchoRequest` from the controller, in the unsigned version this only takes around 0.03ms-0.04ms to parse the message. In the signed version, this increases substantially to 0.15-0.3ms. If I remove the verification step from the signed version, the parsing still requires 0.035ms-0.055ms, an increase on the unsigned version. This shows that the majority of this latency originates from the verification step. It is also worth noting that the verification is only one side of the communication. If I measure the time to sign and append the signature of an `EchoReply` (which is identical to the matching request), this takes around 0.5ms. Whilst individual messages do not introduce a large amount of latency individually, a flow setup for a ping requires at least 6 messages to be sent (a `PacketIn` from the switch to the controller, and a `FlowMod` and `PacketOut` in response, repeated in both directions of travel), which all need to be signed, parsed and verified.

8.3 Multi hop path test

While measuring a single hop path can provide insight into the raw performance of the controller, in most cases in the real world paths are longer than this. Introducing longer paths will produce a cumulative additional delay. To measure this effect, I measure flow setup time for a line topology of increasing length (from 1–10 switches), with a host connected at either end. For this test, I compare a direct controller connection, through a simple tcp proxy, through the SDBFT proxy and through the BFT-SMaRt proxy.

8.3.1 Setup

The mininet setup was used for this test. Mininet was used here, as it allows the easy deployment of more complex topologies. In each test, I launch the controllers, then any required proxies, and then launch Mininet. When using the simple and SDBFT proxies, an instance of the proxy is deployed for each switch. When using BFT-SMaRt, an instance of the client proxy is deployed for each switch, along with an instance of the server proxy for each controller.

I then perform two pings between the pair of hosts, and log the flow setup time for the second ping. The `LearningSwitch` Floodlight application was used for control. I use this application rather than the `Forwarding` application as it will only install flow rules on a single switch at a time, rather than attempt to build the full route on the first packet seen, allowing me to measure the worst case scenario.

BFT-SMaRt is only run in ordered mode in this test, as that is the intended mode of operation by the library. For SDBFT and BFT-SMaRt, 4 controllers are used. When using signatures, SHA512withRSA signatures using a 1024 bit RSA key are used. Each test is repeated 50 times.

8.3.2 Results

Baseline Figure 8.6 shows the flow setup time for 1 to 10 switches with a direct connection from the switches to a single controller, and with all switches connecting to a single controller through a simple TCP proxy deployed for each switch (see Appendix C.2 Table C.8 for full results, and significance test results of unsigned protocols against the direct controller connection). The results for these tests are normally distributed (confirmed through visual inspection and application of the Shapiro Wilks test), so a two sample T-test is used instead of Mann-Whitney U tests for this data.

For a direct controller connection, each additional switch on the path adds a similar amount of additional latency in flow setup time of 3-4ms per switch. The median flow setup time for a single switch is 4.07ms (μ=3.98ms, σ=0.64), increasing to 30.9ms (μ=30.35ms, σ=3.34) for 10 switches. With the addition of

a simple TCP proxy, shown in Figure 8.6, there is a large statistically significant increase over the direct connection to 4.86ms (µ=4.78ms, σ=1.06) for a single switch (T-test: $t = -4.57884, df = 81.01979, p < 0.00001$, effect = $0.90(L)$), up to 42.15ms (µ=41.32ms, σ=4.19) for 10 switches (T-test: $t = -14.46773, df = 93.32191, p < 0.00001$, effect = $2.89(VL)$). With each additional switch the effect size increases (see Appendix C.2 Table C.8) which is expected, due to the effects of additional latency caused by the addition of a proxy being cumulative with each additional switch.

Unsigned Communication For unsigned communication, SDBFT, shown in Figure 8.7 also follows the clear pattern of increasing latency as the path length increases. For a single switch, the median latency is 7.89ms (µ=7.81ms, σ=1.1), increasing to 53.65ms for 10 switches (µ=53.72ms, σ=6.84), with each switch adding an additional 5-7ms extra latency. This represents statistically significant increase over the direct connection with a very large effect size for all path lengths (see Appendix C.2 Table C.8).

BFT-SMaRt median flow setup time increases from 22.105ms (µ=21.95ms, σ=1.44) for a single switch up to 152ms (µ=151.76ms, σ=7.8) for 10 switches, with an additional latency of approximately 15ms per switch. This represents a very large increase in latency over a direct controller connection, both statistically significant with a very large effect size ((T-test: $t = 80.34816, df = 67.76206$, $p < 0.00001$, effect = $16.07(VL)$) for 1 switch, (T-test: $t = 101.15677, df = 66.34967, p < 0.00001$, effect = $20.23(VL)$) for 10 switches). Appendix C.2 Table C.10 provides a cross-comparison T-test between SDBFT and BFT-SMaRt, and shows that for all path lengths, BFT-SMaRt provides statistically significant increased flow setup times, with very large effect size. This clearly demonstrates that SDBFT far outperforms BFT-SMaRt. As an example, a 10-hop path is a statistically significant 183% increase over SDBFT (T-test: $t = -66.81835, df = 96.34406, p < 0.00001$, effect = $13.26(VL)$), although the BFT-SMaRt approach is more consistent, with less relative variance than the SDBFT approach.

Signed Communication As expected, the signed communication follows a similar pattern to the unsigned and baseline results. Figure 8.8 shows the perfor-

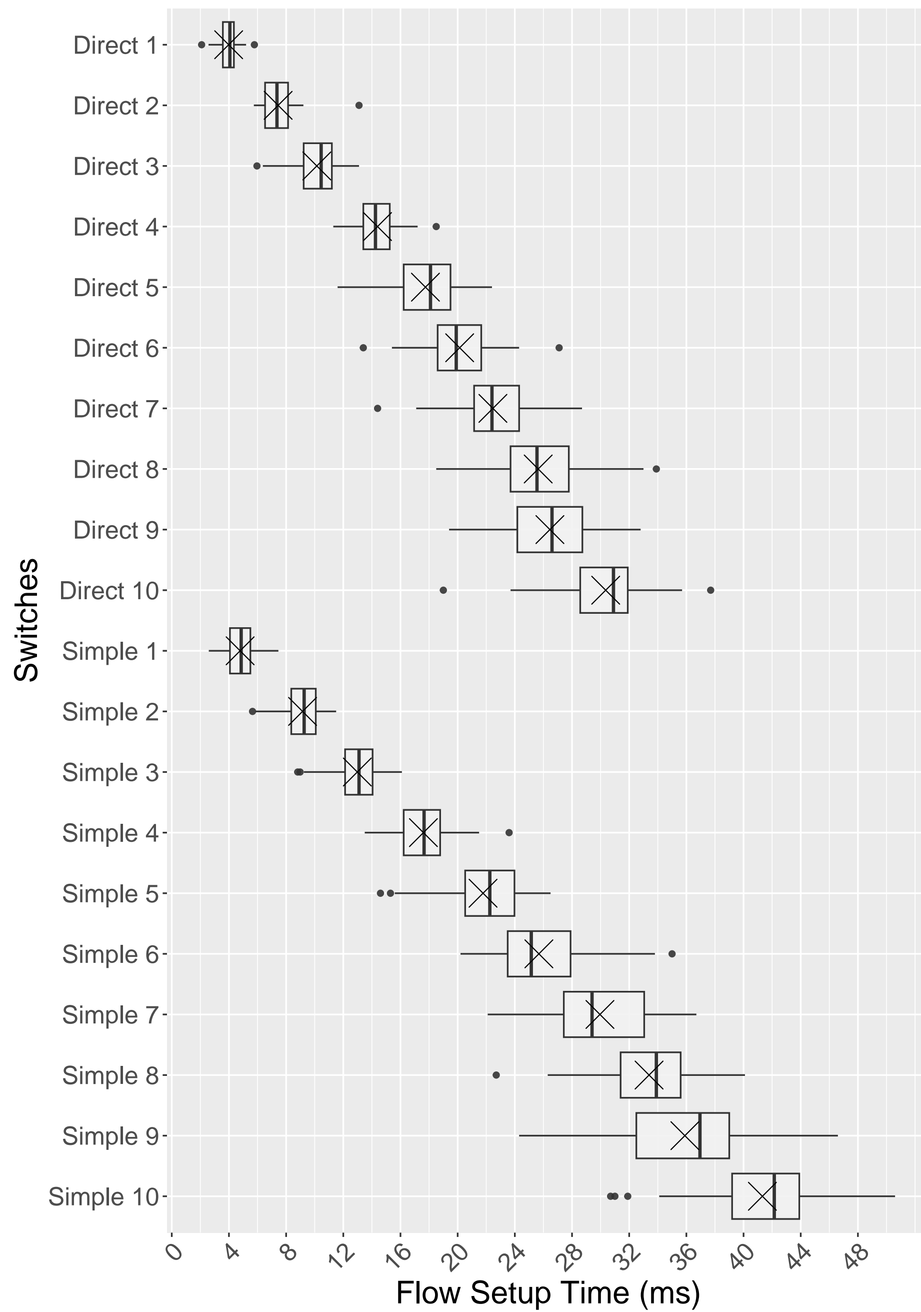


Figure 8.6: Flow setup time (ms) for increasing path lengths of 1 to 10, with a direct controller connection, and through a simple TCP proxy. X=mean. 50 repetitions.

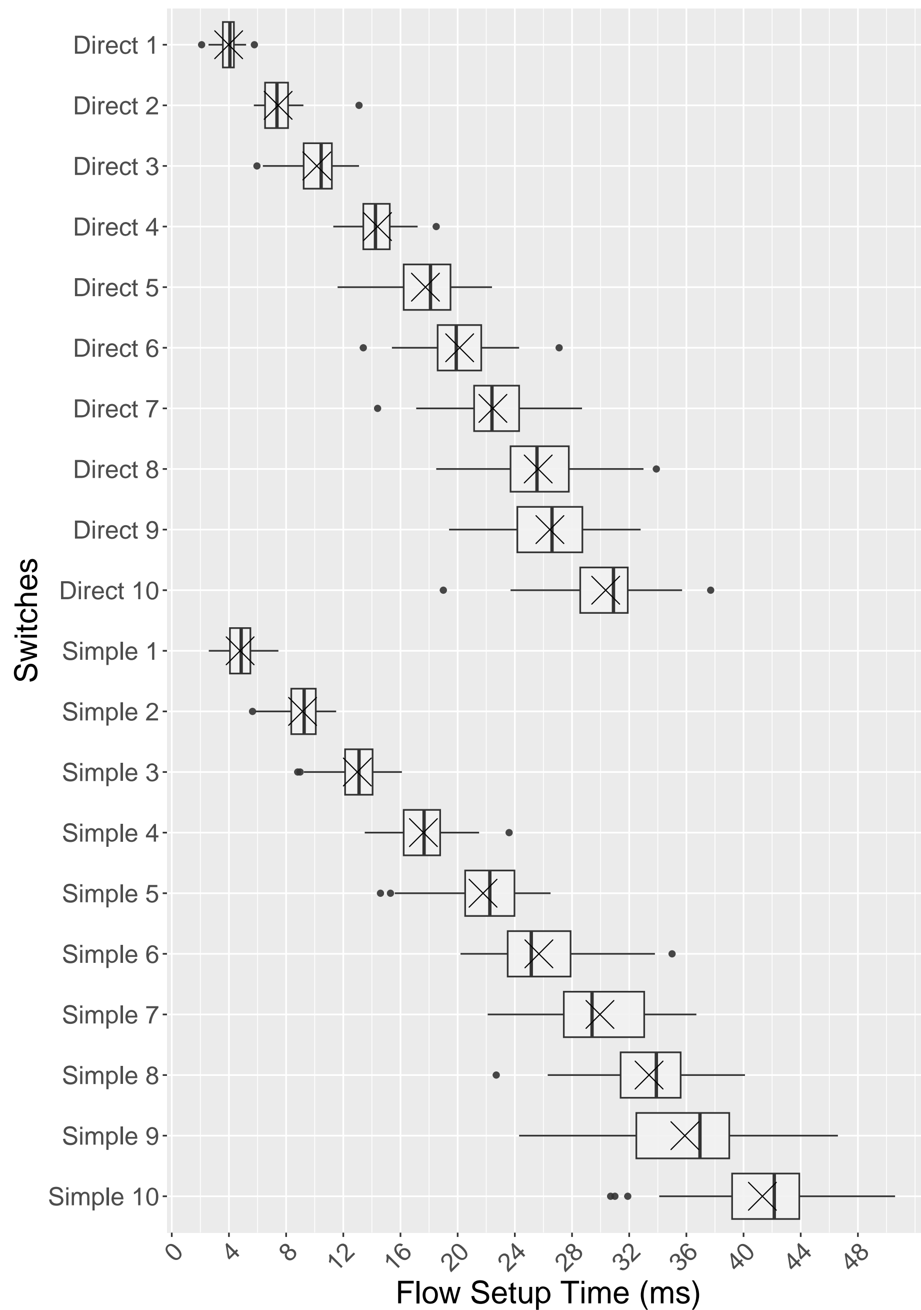


Figure 8.7: Flow setup time (ms) for increasing path lengths of 1 to 10, with SDTBFT and BFT-SMaRt, without using signatures. X=mean. 50 repetitions.

mance of SDBFT when using signed messages. Appendix C.2 Table C.9 presents full results and T-Test significance tests against the direct controller connection.

A single switch has a median latency of 10.5ms (μ=10.51ms, σ=1.54), increasing to 69.55ms (μ=69.54ms, σ=6.29) at 10 switches, both statistically significant with a very large effect size ((T-test: $t = -27.68683, df = 65.70923, p < 0.00001$, effect $= 5.54(VL)$) for 1 switch, (T-test: $t = -38.89269, df = 74.53187, p < 0.00001$, effect $= 7.78(VL)$) for 10 switches). Interestingly, increasing from 8 to 9 switches results in no statistically significant increase in flow setup time (T-test: $t = -0.4254, df = 96.3133, p = 0.6715$, effect $= 0.085(S)$). It is unclear why this occurs, as there are no extreme outliers in the results for either 8 or 9 switches, and both have similar variance of 4.73 and 4.14. Further testing may be needed to establish if this difference is truly consistent over a larger testing regime.

The results for BFT-SMaRt follow a clearer pattern. A single switch has a mean flow setup time of 30.05ms (μ=29.83ms, σ=2.38, median=30.05ms), increasing to 223ms (μ=223.58, σ=8.53) for 10 controllers, both statistically significant with a very large effect size when compared to the direct controller connection. ((T-test: $t = 74.23714, df = 56.16721, p < 0.00001$, effect $= 14.85(VL)$) for 1 switch, (T-test: $t = 149.14835, df = 63.65559, p < 0.00001$, effect $= 29.83(VL)$) for 10 switches). As in the unsigned case, this represents a significant increase on the direct controller connection. Appendix C.2 Table C.11 shows that in all cases, the results for BFT-SMaRt are statistically significantly higher than those when compared to SDBTF when using signatures, demonstrating clearly that, as with the unsigned case, SDBFT far outperforms BFT-SMaRt.

8.3.3 Discussion

In this test, I evaluate the performance of the SDBFT and BFT-SMaRt proxies when setting up multi-hop routes. I find that both systems scale as expected, with a stable increase in flow setup time with each additional switch that is introduced. When using unsigned messages, the flow setup time for a 10-hop path is almost three times as much when using BFT-SMaRt compared to SDBFT. This is also reflected in the signed results, with an approximate 2.5 times increase in flow setup time for a 10-hop path.

This result shows that the SDBFT proxy performs well when handling multi-hop routing. The difference between the direct controller connection and SDBFT

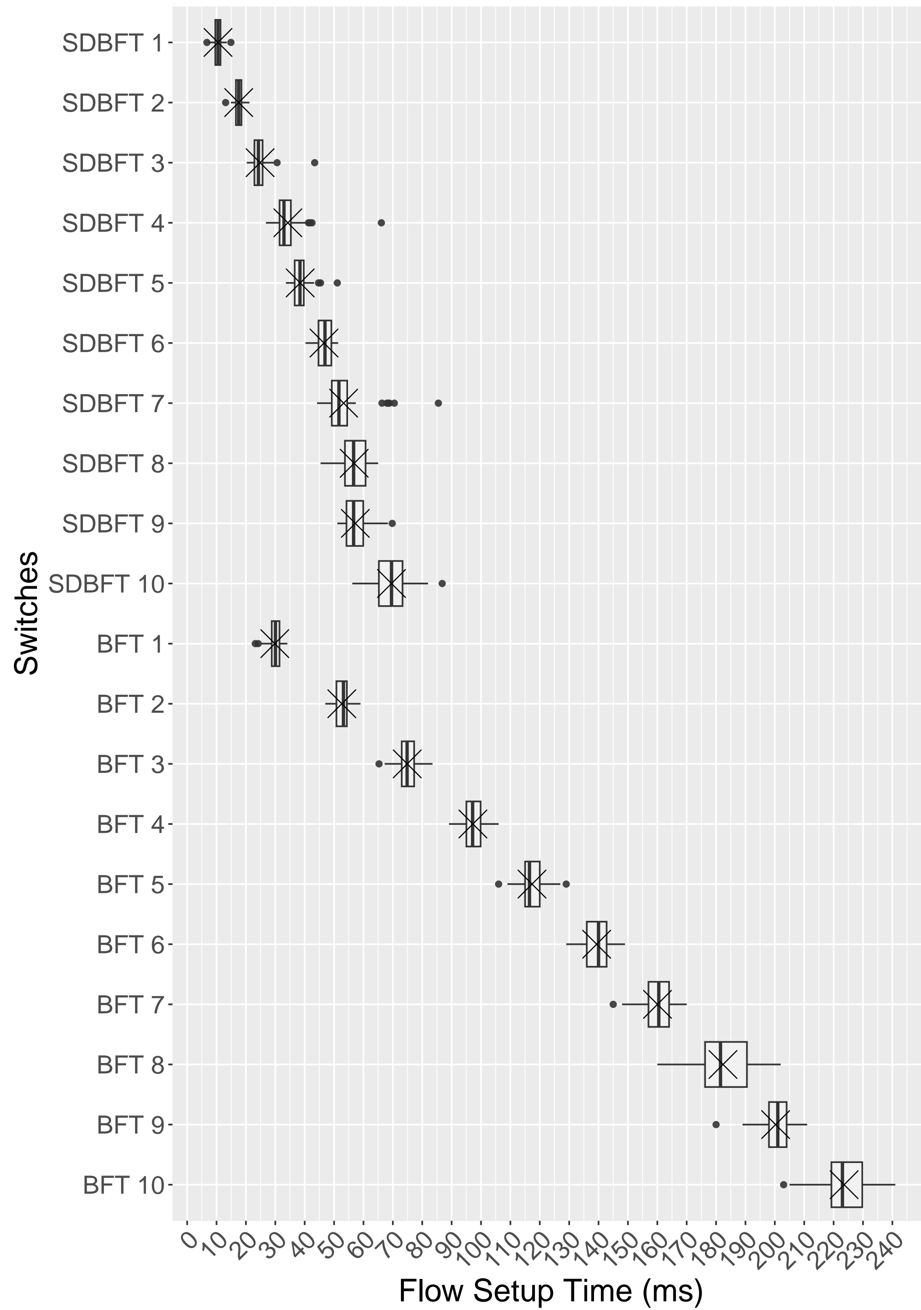


Figure 8.8: Flow setup time (ms) for increasing path lengths of 1 to 10, with SDTBFT and BFT-SMaRt, using signatures. X=mean. 50 repetitions.

is slightly greater than found in my baseline tests found in Section 8.2, with roughly twice as much of an increase in flow setup time when compared to the

baseline test. This same pattern is also found with when using BFT-SMaRt, so is potentially down to CPU differences in the host machine on which the experiments are run.

For both SDBFT and BFT-SMaRt, the effect sizes when compared to the direct connection, seen in Appendix C.2 Tables C.8 and C.9, do not increase or decrease with longer path lengths (though they do vary), an effect not seen with the simple TCP proxy where the effect size steadily increases with longer path lengths. This effect occurs both when using signatures, and without. In both cases median flow setup times increase steadily with each additional switch.

8.4 Failure operation

In this set of experiments, I aim to measure the performance of the SDBFT controller architecture whilst an attack occurs. More specifically, I measure the latency introduced when the failover handover operation occurs, and the performance of subsequent new flows using the expanded quorum of controllers.

8.4.1 Setup

I make use of the OVS-based setup described in 7.1.1. The network consists of four hosts (H1, H2, H4, H4) connected to a single switch (as shown in Figure 8.1). I load the AmplifiedDOS malicious application (as described in 4.4), with a single host being targeted (H4). I measure the time taken for flow setup under no attack, flow setup on the flow where the attack initially occurs, which includes the time taken to contact the backup controllers after the primaries, and then for the following flow setups using the expanded controller quorum, with the attack continuing. For SDBFT I measure performance using a primary quorum of 4 controllers, with 3 further controllers as backup. I compare this with the traditional BFT approach utilising 4 controllers (to handle a single faulty node). In this scenario, a single controller is *malicious*.

I then load the TargetDropper malicious application (described in Section 4.4), which installs an empty flow rule onto the switch with no actions, causing packets

on the target flow to be dropped by the switch. This application is used as it has a clear effect on the network, in that if the attack is successful, then the target flow will undergo a denial-of-service attack. In this test, I make 3 of the 4 controllers malicious. Each experiment is repeated 20 times.

8.4.2 Results

The results for a single failure are presented in Figure 8.9 and Table 8.3. As can be seen, there is a large increase in latency on the flow where the attack occurs, with a median flow setup time of 197.9ms (µ=193.98ms, σ=18.57), as the protocol contacts the secondary quorum. This then stabilises at 27.26ms for subsequent flows. This is a statistically significant increase over the setup time before the attack (Mann-Whitney U: $U = 62, z = -3.9133, p < 0.0001$, effect $= 0.59(L)$), however this increase in setup time is equivalent to the increase seen when increasing the controller count seen in Section 8.2. As expected, the BFT approach maintains its performance whilst under the presence of a single fault, with pre, during and post attack median flow setup times of 26.84ms, 26.4ms and 26.83ms respectively. The during and post attack values represent no statistically significant difference when compared to the pre attack time, with the flow setup time during attack having Mann-Whitney U results of $U = 207, z = 0.1738, p = 0.862$, effect $= 0.028(N)$, and the post-attack setup having a result of $U = 199, z = -0.9893, p = 0.9893$, effect $= 0.0021(N)$. Whilst this has a benefit on the first flow the fault occurs, it remains constant whilst SDBFT recovers performance close to normal levels for subsequent flows which is statistically the same as BFT-SMaRt (Mann-Whitney U: $U = 215, z = 0.388, p = 0.698$, effect $= 0.062(N)$).

Figure 8.10 presents the results for 3 out of 4 controllers becoming compromised, running the TargetDropper malicious application. In the case of SDBFT, on the flow where the attack occurs there is a median flow setup time of 230.48ms (µ=239.27ms, σ=23.05). This represents a statistically significant increase with large effect size on the single failure result (Mann-Whitney U: $U = 4, z = -6.3826, p < 0.0001$, effect $= 0.84(L)$). This increase is down to the time taken to send the request to the backup controllers and then reach the majority threshold

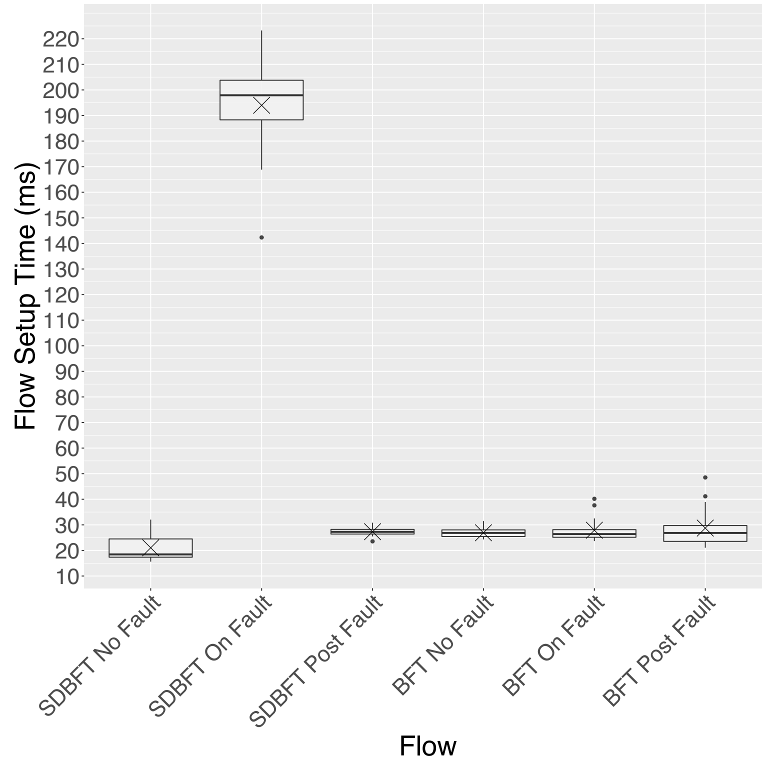


Figure 8.9: Fault recovery under SDBFT and BFT-SMaRt with a single fault. X=mean. 20 repetitions.

of $f + 1$ matching responses. In the single failure scenario, where only 1 of the $f+1$ primary controllers is faulty, and so f are correct, then only a single correct response from one backup is enough to reach that threshold. If, however, f of the $f + 1$ are faulty, then the responses from all f backups need to be collected to reach the threshold.

For the subsequent flows, this reduces to a median of 28.23ms (μ=30.07, σ=5.28), again close to the pre-attack flow setup time. Whilst the flow setup time is delayed, the flow is successfully setup and the hosts can communicate. Even though the majority of the primary quorum is compromised, the single benign controller differs in response, and so the failover protocol is triggered. The backup quorum is brought in, increasing the total number of controllers to 7, with a majority of 4 being correct (assuming that no controllers in the backup quorum are compromised), meaning the correct flow rule can be installed on the switch. In the case of BFT-SMaRt, however, where 4 controllers can only handle

Table 8.3: Mean, standard deviation and median (in milliseconds) of failure mode flow setup times before, during, and after attack, for a single faulty controller and 3 faulty controllers.

			Mean	SD	Median
Single Fault	**SDBFT**	**Pre**	21.03	5.08	18.45
		Attack	193.98	18.58	197.92
		After	27.32	1.54	27.26
	BFT	**Pre**	26.90	1.83	26.84
		Attack	27.91	4.42	26.40
		After	28.73	7.23	26.83
Three Faults	**SDBFT**	**Pre**	23.59	7.16	21.80
		Attack	239.37	23.05	230.48
		After	30.07	5.28	28.23
	BFT (4 Nodes)	**Pre**	34.62	4.94	34.05
		Attack	27.83	3.32	27.69
		After	-	-	-
	BFT (10 Controllers)	**Pre**	30.50	2.61	30.44
		Attack	35.65	8.44	32.11
		After	44.08	41.26	29.81

a single faulty node, the 3 malicious nodes hold the majority and the malicious flow rule is installed onto the switch (this results in a flow setup time of 0ms as the ping fails).

BFT-SMaRt requires 10 replicas to handle 3 faulty replicas, and so I repeat the test for BFT-SMaRt with 10 controllers. As can be seen in Figure 8.10, 10 controllers are able to handle the 3 malicious controllers, and the flow is successfully setup.

8.4.3 Discussion

The results demonstrate the key tradeoff when utilising the SDBFT approach compared to a traditional BFT approach. As can be seen in Figure 8.9, whilst the flow on which the attack occurs experiences a far higher setup time than normal, this stabilised for all subsequent flows. Whilst the stable point still represents a slightly higher flow setup time due to the use of additional controllers, it remains at the same level as BFT-SMaRt approach, even when not under attack. Whilst BFT-SMaRt has minimal impact from the attack (for a single failure), this requires additional latency on every flow, even whilst not under

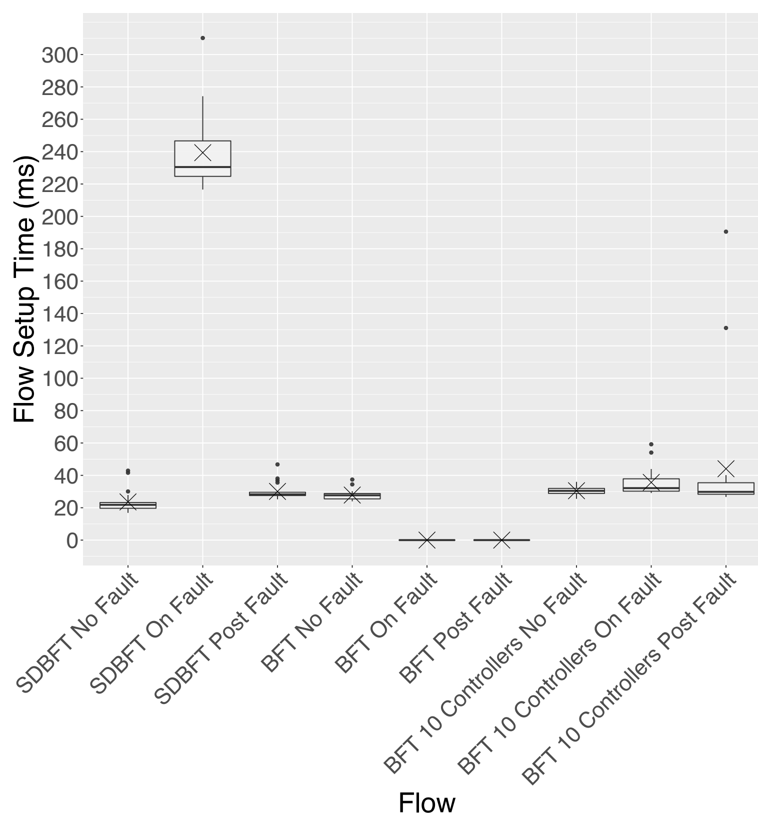


Figure 8.10: Fault recovery under SDBFT and BFT-SmaRt, 3 faults. X=mean. 20 repetitions.

attack/fault. This clearly demonstrates the intended tradeoff of SDBFT of a lower latency whilst not under attack, for a substantially higher latency on the flow where the attack occurs.

The second benefit of SDBFT is shown in the second test consisting of multiple malicious controllers. To handle the three faults, only 4 controllers are required, increasing to 7 once the attack occurs. Even whilst under attack, this is still lower than the minimum of 10 controllers required by BFT-SMaRt, which are required on all flows, even before the attack occurs. This reduction in the number of required replicas can provide much greater scalability.

8.5 High Throughput Benchmark

In this test, I measure the performance of the SDBFT system whilst under a high load by using the Cbench tool, as described in 7.3.2. This test represents a worst-case scenario, as the Cbench tool is designed to find the maximum load that a controller can handle. I expect that SDBFT will have a noticeable impact on the Cbench results, as a large amount of extra processing is applied to packets between the switch and controller. As I have shown in the baseline testing (Section 8.2), SDBFT can process switch requests with lower latency than the comparative BFT-SMaRt implementation, and so therefore I expect that SDBFT should also have noticeably better Cbench results than BFT-SMaRt. I test with both a single emulated switch, and sixteen. Utilising the latency mode of operation in Cbench with a single switch, where a single request is sent at a time, will reveal how well SDBFT and BFT-SMaRt can process messages in quick succession. Running Cbench with 16 switches in throughput mode represents a very high load on the controller, which will push the SDBFT and BFT-SMaRt protocols to their limits, and in particular measure how well they can handle multiple simultaneous requests.

8.5.1 Setup

I perform this test on a workstation machine — a Dell Precision T7610 workstation, equipped with two Intel Xeon CPU E5-2650 processors (8 core, 16 thread, 2.0-2.8 GHz) CPUs and 128GB RAM. The controllers, proxies and Cbench tool are all run on the host operating system (Ubuntu 18.04). The Cbench tool is run in both latency and throughput modes (as described in 7.3.2). I measure performance of a direct controller connection, through a simple TCP proxy, and through the SDBFT and BFT-SMaRt proxies both using 4 controllers. The Floodlight controller is using the LearningSwitch application for forwarding.

Cbench was run with both a single switch, and the default 16 switches. When 16 switches were in use, 16 instances of the simple TCP proxy, SDBFT proxy, or BFT-SMaRt client proxy were launched (one for each switch). For each test, Cbench was run once, with 80 loops. The first 20 are discarded as warmup. In

particular when proxying, it takes approximately 10-15 loops for the results to stabilise, so I ignore the first 20 loops to allow for variance in this. I then measure for 60 loops, which is six times as many loops as the default setting of 10, and represents a full minute of testing under load (each loop is 1 second). A delay of 5 seconds was applied between the beginning of the switch to controller handshake and the starting of testing to ensure the switch to controller connection is properly established.

When using ordered BFT-SMaRt with 16 switches in throughput mode, the BFT-SMaRt system fails due to message timeout, caused by an overloading of the BFT system. The default request timeout is 2 seconds. I have confirmed that the controller responds to all requests within 100ms of receiving them, which indicates this delay is rooted within the BFT protocol. To enable BFT-SMaRt to complete the test, I set the request timeout to 0, which disables timeouts. This allows the test to complete.

For each test, I present the min, max, mean and standard deviation in responses per second as reported by Cbench. For each test Cbench is run 3 times, and the result with the highest mean responses/second is reported.

8.5.2 Results

Single Switch I present the results of Cbench when run with a single emulated switch in Table 8.4. As can be seen in this table, a direct controller connection performs similarly when running in both latency and throughput modes, processing a mean of 33594 and 33812 requests/second respectively. The introduction of the simple TCP proxy actually improves the result in the latency tests, and only slightly decreases performance in the throughput test. It is unclear why it has this effect on the latency test. When the test was repeated, the result was similar.

When performing the latency test, both SDBFT and BFT-SMaRt perform noticeably worse than a direct controller connection. SDBFT when run with a single controller without using signatures, only experiences a small drop in performance to a mean 24874 responses/second. This decreases further to 16072 and 5619 responses/second for 4 and 10 controllers respectively. BFT-SMaRt,

Table 8.4: Cbench benchmarking result with a single virtual switch. N = number of controllers. S = Signed (Y/N) Values = responses/second (rounded to nearest whole integer)

			Latency				Throughput			
	N	S	Min	Max	Mean	SD	Min	Max	Mean	SD
Direct	1		24935	51149	33594	8133	24753	50838	33813	7579
Simple	1		25573	51153	38968	8563	22252	42980	32723	3646
SDBFT	1	N	21356	29387	24874	2339	994	43938	5220	9563
SDBFT	1	Y	1865	2231	2093	101	1531	2520	1948	215
SDBFT	4	N	5627	20441	16072	3822	723	3373	1326	617
SDBFT	4	Y	1820	2256	2084	108	1591	2342	1952	166
SDBFT	10	N	1425	6761	5619	1244	484	2585	993	437
SDBFT	10	Y	1804	2428	2162	121	1543	2356	1994	212
BFT (Ord)	4	N	1265	6303	1627	942	6901	10656	9094	952
BFT (Ord)	4	Y	684	2326	849	288	2131	2466	2337	75
BFT (Unor)	4	N	4021	9905	4746	981	9698	12828	11840	752
BFT (Unor)	4	Y	1178	2663	1298	252	2451	2871	2768	90
BFT (Ord)	10	N	364	2452	973	329	1283	10685	5070	1909
BFT (Ord)	10	Y	299	1032	597	150	651	1538	1310	189
BFT (Unor)	10	N	2508	4427	3016	305	5282	11700	8327	1214
BFT (Unor)	10	Y	540	2012	1317	202	2143	2372	2271	49

when using ordered messages, only manages 1627 responses/second when using 4 controllers, and 973 when using 10. I expect BFT-SMaRt to perform worse in this test, as it relies on the proxies processing responses quickly, and as I show in the baseline experiments BFT-SMaRt requires a higher flow setup time than SDBFT.

The throughput tests, however, see the reverse. In these tests, BFT-SMaRt outperforms SDBFT in almost every test. I believe this is largely down to the use of Netty sockets in BFT-SmaRt, compared to the standard Java sockets used in SDBFT. Netty sockets are non-blocking and are built upon listeners, rather than waiting and reading from a socket as is the case with standard Java sockets which are blocking. This allows BFT-SMaRt to receive packets and process them at a higher rate that the SDBFT proxy. This is confirmed with the results for the unordered BFT-SMaRt proxy, which whilst similar in operation to SDBFT, outperforms it.

Incorporating signatures has a major impact on the Cbench output. For example, whilst SDBFT with a single controller in the latency test processes 24874

responses/second, this drops to 2093 when using signatures. The reduction is less pronounced for 4 and 10 controllers. This is most likely explained by the extra packet processing required to parse and verify the signed messages. This is done in the same thread that reads from the socket, and so blocks subsequent reads until it is done. This could also explain why this effect is reduced for BFT-SMaRt, as the non-blocking sockets do not get restricted by the extra time. An interesting observation is that for SDBFT, when using 1, 4 or 10 controllers and signed messages, the performance does not decrease with an increased controller count. It is unclear why this is the case, however as is shown in the benchmark testing increasing the controller count for SDBFT has a minimal impact on performance. This similar performance in contrast to the unsigned version when there is a greater drop could again be down to the extra processing on signed messages stabilising the performance.

Conversely, when running in throughput mode, SDBFT performs better with 4 and 10 controllers when using signatures than without. This may be due to the generally slower communication when signatures are in use — the controller can send responses at a slower rate as it has to sign messages which prevents the proxy socket thread from becoming overloaded.

Sixteen Switches Table 8.5 presents the Cbench results when simulating 16 switches. As can be seen in the direct results, the controller processes 110532 and 294646 responses/second in the latency and throughput tests respectively. In the throughput test, 800 `PacketIn` messages are sent to the controller at a time, which explains why the mean responses/second are so high. The simple proxy causes a small reduction in responses in the latency test, but actually increases the number of responses in the throughput test.

The introduction of SDBFT with a single controller only has a small impact on the performance in latency mode, with just a 0.7% drop on the direct controller connection, and performs better that the simple proxy. Increasing to 4 and 10 controllers introduces a greater drop when compared to the single switch test, to 26207 and 10908 responses/second respectively, a 76.11% and 90.01% decrease. In the single switch test, increasing the controller count only resulted in a 35.39% and 77.4% drop. It is unclear exactly why this performance is worse, as the

Table 8.5: Cbench benchmarking result with sixteen virtual switches. N = number of controllers. S = Signed (Y/N) Values = responses/second (rounded to nearest whole integer)

			Latency				Throughput			
	N	S	Min	Max	Mean	S/D	Min	Max	Mean	S/D
Direct	1		83036	128736	110532	11050	273604	472480	294646	35283
Simple	1		83857	198516	94884	5696	240690	381428	330364	24747
SDBFT	1	N	84366	135646	109691	9807	6978	313665	140261	85698
SDBFT	1	Y	17573	20997	19643	856	6325	18149	13662	2459
SDBFT	4	N	21176	29555	26207	1651	3263	148405	32806	32575
SDBFT	4	Y	6299	7222	6759	204	2407	6214	4539	890
SDBFT	10	N	8506	12419	10908	841	0	89146	13228	16294
SDBFT	10	Y	2797	3597	3057	159	1547	3331	2562	433
BFT (Ord)	4	N	4488	7765	6975	619	1554	5829	4095	1045
BFT (Ord)	4	Y	4018	6892	5840	573	2739	6811	5552	989
BFT (Unor)	4	N	14104	20621	18228	1492	8326	26009	20707	4248
BFT (Unor)	4	Y	7753	11655	10301	810	10531	15651	13905	1451
BFT (Ord)	10	N	851	2883	2770	701	233	3981	1709	1078
BFT (Ord)	10	Y	805	3312	2034	567	462	3707	1960	800
BFT (Unor)	10	N	4147	7969	6039	679	2189	8894	6186	2112
BFT (Unor)	10	Y	2898	6516	5356	653	1217	7546	5959	1619

SDBFT proxy only represents a single switch. A potential explanation is that whilst in latency mode, Cbench will wait until it has received the response to the previous request before sending the next request. As previously discussed, the use of standardised sockets potentially causes congestion on the proxy. This, combined with congestion on the host machine caused by the traffic of 16 switches to 10 controllers, could cause sufficient delays in the final responses from the controllers being processed by the proxy and forwarded to the switch (as the response from all primary controllers needs to be processed before sending to the switch). This is less apparent in BFT-SMaRt, which is able to forward the response to the switch as soon as the threshold number of matching responses is reached.

BFT-SMaRT performs very poorly in latency mode when using ordered messages, being able to process just 6975 requests with 4 controllers, and 2770 with 10. I expected BFT-SMaRt to perform worse due to its slower response time as shown by the baseline testing. The reduction is consistent with the single switch test however, as for 16 switches 4 and 10 controllers experience a 93.69% and 97.49% drop over the direct connection, compared to 95.16% and 97.1% in the

single switch test.

When running in throughput mode, both SDBFT and BFT-SMaRt suffer from a heavily reduced performance when compared to the direct controller connection and simple TCP proxy. SDBFT with 4 controllers is only able to process 32806 requests/second, a 88% reduction when compared to the direct connection. For BFT-SMaRt, this reduction is 98.61%. For SDBFT, this slowdown is most likely due to threading issues as I have previously discussed. For BFT-SMaRt, as I mentioned in the setup description above, I had to disable request timeouts in the BFT protocol, as the protocol was becoming overloaded when running the throughput tests. A reason why SDBFT performs better than BFT-SMaRt in this case is that Cbench sends 800 `PacketIN` messages in a single request, which SDBFT is able to forward onto to the controller in a single request as it can receive responses to any requests it has processed and then process each message. In BFT-SMaRt, as the protocol requires a single response to a request, the BFT-SMaRt client proxy creates an individual request for every `PacketIN` message seen, even if multiple are sent by the switch in one request. This means that the server proxy does not need to wait for many responses from the controller before forwarding to the client, but for scenarios such as this test with a very large number of requests there is a large network overhead in sending all of the requests individually.

There is one anomaly in the SDBFT throughput testing with 10 controllers. In all of the throughput tests when not using signatures, on at least one loop of Cbench no responses were received, which causes a minimum value of zero. Of the 80 loops I run (discarding the first 20 as a warmup), in all cases this happens within the first 30, and so could possibly be caused by a slower warmup time due to the load.

As was the case with the single switch tests, introducing signatures causes a significant reduction in performance across the SDBFT tests for both latency and throughput. As I have found throughout this testing, this effect is less pronounced when using BFT-SMaRt.

8.5.3 Discussion

I have found that both the SDBFT and BFT-SMaRt proxies represent a substantial decrease in performance when tested with Cbench when compared to a direct controller connection. This is somewhat as expected, as the proxies introduce a large amount of extra processing onto requests which will have an impact on how many requests can be handled per second. This result indicates that consideration should be made when deploying the SDBFT proxy, especially in a scenario of high rate, diverse traffic that will require a high amount of requests to be made to the controller.

As I have shown, a limitation of the SDBFT implementation is the use of Java sockets for communication. When faced with very high loads, these sockets struggle to keep up with the rate of packets when compared to the Netty sockets used in BFT-SMaRt. In future iterations of the SDBFT proxy, a migration to Netty sockets, or similar, should be considered.

It is important to consider that Cbench results can be affected quite substantially with different setups, and represent a best case scenario. For example, I originally attempted to run Cbench on the OVS setup, however the results were substantially lower across all tests, including a direct connection, due to the controllers being located inside virtual machines. Because of this results from Cbench should only be taken as an estimate of peak performance.

8.6 Testing on physical switch

Our previous tests have all been performed using virtual switches. In this test, I aim to verify that the SDBFT proxy works with a commercial, physical SDN switch, along with physical hosts. I compare the performance of SDBFT with a simple TCP proxy, along with the BFT-SMaRt proxy.

8.6.1 Setup

For this test I use the physical network setup described in 7.1.3. I use two different topologies in this test, as shown in Figure 8.11. The first of these is a single

switch topology, using a single switch and four Raspberry Pi hosts connected to that switch. The second uses three switches in a line topology, with two RPis connected to the first switch, and then a single RPi to each of the other two switches. In this test, all of the required proxies and controllers are run on a single server. When using three switches, a proxy is deployed for each switch. In each test, I first ping between H2 and H3 to *warm up* the controller, and then measure the second flow setup time between H1 and H4. Each test is repeated 50 times, with the OpenFlow instance on the switch reset after each test, which requires the OpenFlow handshake to be run on each test. All protocols are run without using signature in this test. SDBFT is run with a single, 4 and 10 controllers, whilst BFT-SmaRt is run with 4 and 10 controllers.

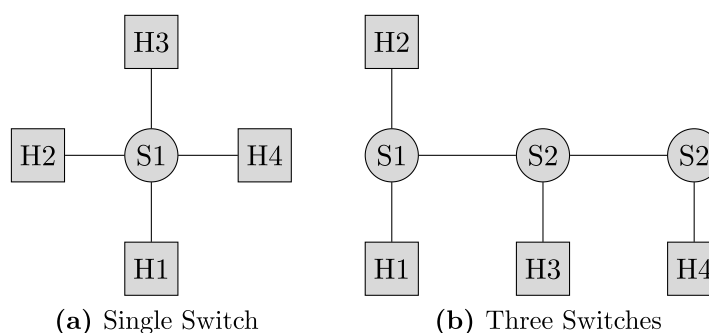


Figure 8.11: Topologies used for physical switch testing. X=mean.

8.6.2 Results

I found that when using the physical switches, all tests resulted in a larger amount of variance, and therefore greater proportion of, and proportionally larger, outliers in the flow setup time results, including when utilising a direct controller connection.

After investigation, I believe one of the primary reasons for large outliers is the `LinkDiscoveryManager` module within Floodlight, which send out regular LLDP floods out of all available network ports in order to identify connected devices. If this flood occurs around the same time that the ping test is occurring, then it will result in a larger flow setup time due to a delay caused by the Floodlight

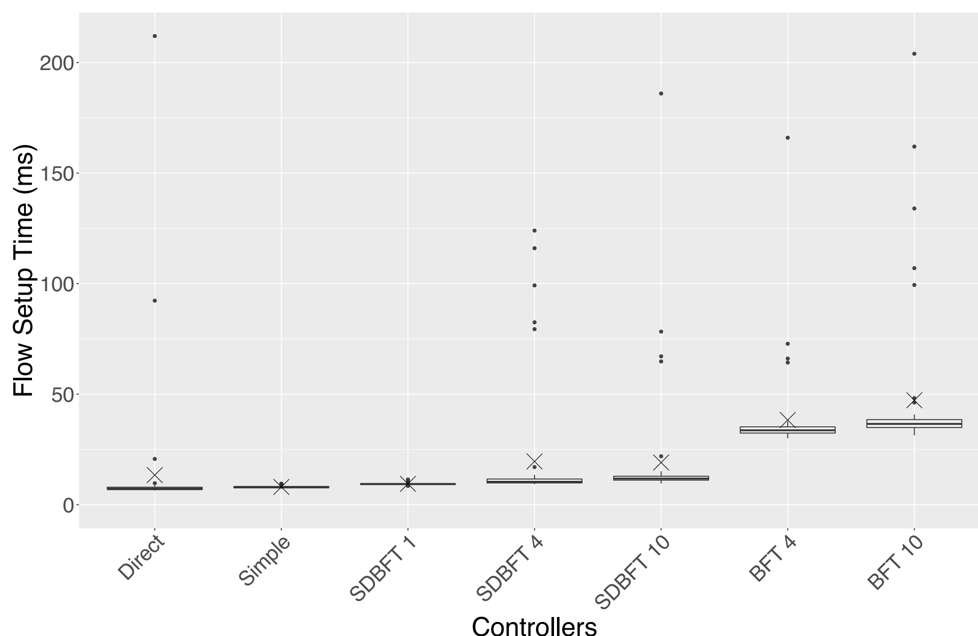


Figure 8.12: Testing with single physical switch. X=mean. 50 repetitions.

`TopologyManager` recomputing the network topology. Further variance is caused by some element of the physical setup causing additional latency which is not present in other tests, which are run on a single host machine. In some cases, examining network traffic reveals that the physical switches and hosts generate additional network traffic which virtual hosts do not, in particular ARP and LLDP packets, which also have to be processed by the controller. This also increases flow setup time if these occur during tests.

Single Switch The results for a single physical switch can be found in Figure 8.12 and Table 8.6.

A direct connection from the switch to the controller results in a median flow setup time of 7.28ms. Introducing the simple TCP proxy increases this slightly to 7.88ms, a statistically significant increase, though with a small effect size (Mann-Whitney U: $U = 572, z = -4.67075, p < 0.0001$, effect $= 0.47(S)$). This is a similar level to what I measured in baseline tests in Section 8.2, which also showed a small, but significant, increase in flow setup time with the introduction of the simple TCP proxy.

Table 8.6: Physical switch results and statistical significance tests. Mean and median in milliseconds. Mann-Whitney U tests against direct controller connection.

					Mann-Whitney U			
Switches	**Approach**	**Mean**	**S.D.**	**Med.**	**U**	**Z**	**P**	**Effect**
One	Direct	13.39	31.13	7.28	-	-	-	-
	Simple	8.02	0.52	7.88	572	-4.67075	<0.0001	0.47 (S)
	SDBFT 1	9.43	0.50	9.31	193.5	-7.28031	<0.0001	0.73 (M)
	SDBFT 4	19.56	27.77	10.35	152	-7.56644	<0.0001	0.76 (M)
	SDBFT 10	19.07	27.90	11.80	145	-7.61573	<0.0001	0.76 (M)
	BFT 4	38.30	20.27	33.65	2401	-7.93186	<0.0001	0.79 (M)
	BFT 10	47.24	34.06	36.55	2405	-7.95954	<0.0001	0.80 (L)
Three	Direct	48.55	67.96	18.60	-	-	-	-
	Simple	58.58	70.04	22.00	653.5	-4.32757	<0.0001	0.43 (S)
	SDBFT 1	61.23	69.87	25.40	525	-5.18758	<0.0001	0.51 (M)
	SDBFT 4	56.70	58.45	29.80	537	-5.10716	<0.0001	0.51 (M)
	SDBFT 10	83.18	87.21	30.10	480	-5.48852	<0.0001	0.54 (M)
	BFT 4	135.47	99.62	79.10	2283.5	-6.57614	<0.0001	0.65 (M)
	BFT 10	179.12	94.17	165.00	2363.5	-7.11158	<0.0001	0.70 (M)

The SDBFT proxy with a single controller increases the flow setup time to 9.31ms, a statistically significant increase over a direct connection (Mann-Whitney U: $U = 193.5, z = -7.28031, p < 0.0001$, effect $= 0.73(M)$), which is a comparative increase as observed in baseline testing (Section 8.2. Increasing to 4 and 10 controllers results in median flow setup times of 10.35ms and 11.7ms, a statistically significant increase with moderate effect size over the direct connection. Whilst the mean actually decreases when increasing from SDBFT with 4 to 10 controllers, this still provides a statistically significant increase with a small effect size (Mann-Whitney U: $U = 595.5, z = -4.7169, p < 0.0001$, effect $= 0.47(M)$). Examining Figure 8.12 reveals 5 outliers when using 4 controllers which increase the mean and median values for 4 controllers slightly.

For BFT-SMaRt, the median setup time for 4 controllers is 33.65, statistically significant increase with moderate effect size on the direct case (Mann-Whitney U: $U = 2401, z = -7.93186, p < 0.0001$, effect $= 0.79(M)$), increasing to 36.55ms for 10 controllers. Comparing SDBFT and BFT-SMaRt with 4 controllers shows a statistically significant different (Mann-Whitney U: $U = 250, z = -7.0281, p < 0.0001$, effect $= 0.7(M)$), demonstrating that SDBFT clearly outperforms the BFT-SMaRt approach.

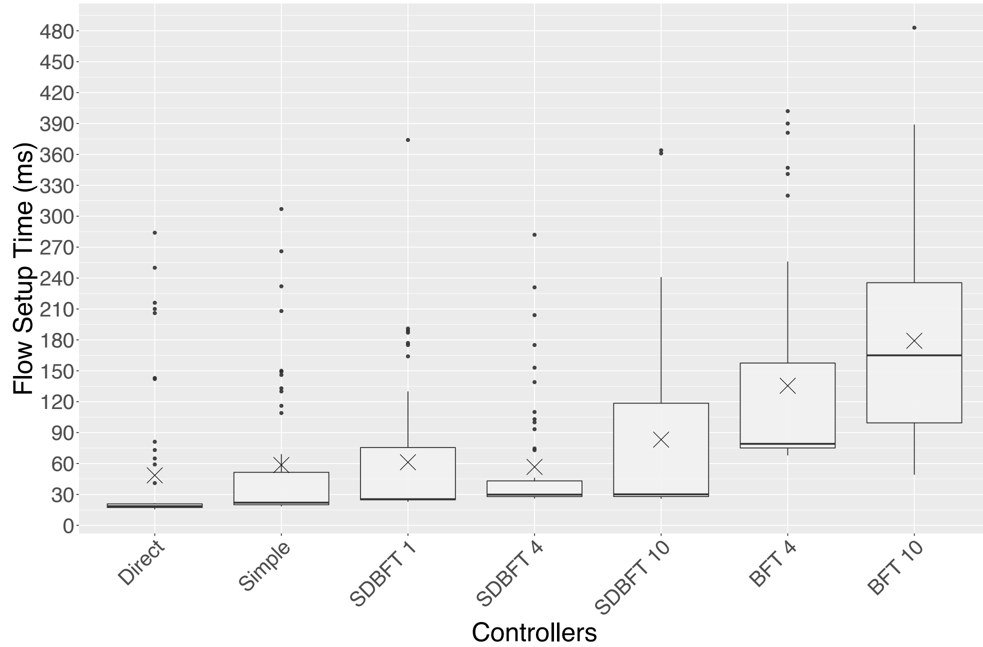


Figure 8.13: Testing with three physical switches, X=mean. 50 repetitions.

Three Switches The results for a three physical switches can be found in Figure 8.13 and Table 8.6. A direct switch to controller connection provides with a 3-hop path provides a median flow setup time of 18.6ms. The simple proxy increases this to 22ms, a statically significant increase, though with small effect size (Mann-Whitney U: $U = 653.5, z = -4.327571, p < 0.0001$, effect $= 0.43(S)$). This effect size is almost the same as in baseline tests (Section 8.2), where the tcp proxy resulted in a significant increase with effect size 0.46.

The SDBFT proxy with a single controller results in a flow setup time of 25.4ms, which increases to 29.8ms and 30.10ms for 4 and 10 controllers respectively. In this case, the increase from 4 to 10 controllers is not statistically significant (Mann-Whitney U: $U = 1212.5, z = -0.5857, p = 0.5581$, effect $= 0.058(N)$). As with the single switch example, there are a number of outliers visible in Figure 8.13, which could explain this lack of increase when increasing the controller count.

Using the BFT-SMaRt proxy results in median flow setup times of 79.10ms and 165.0ms for 4 and 10 controllers respectively. As shown in Table 8.6, both of these represent statistically significant increases over the direct controller connec-

tion. If I compare SDBFT and BFT-SMaRt with 4 controllers, as with the single switch case this represents a statistically significant increase with moderate effect size, again showing that SDBFT outperforms BFT-SMaRt (Mann-Whitney U: $U = 387, z = -6.1108, p < 0.0001$, effect $= 0.61(M)$).

8.6.3 Discussion

The aim of this test was to demonstrate the SDBFT proxy in use with a physical testbed, using a physical, commercial SDN switch, as well as physical hosts. The test demonstrates that the proxy works well with physical devices — the SDBT proxy successfully provides control to both a single switch, and multiple switches. In both cases, the SDBFT proxy outperforms the equivalent BFT-SMaRt deployment, with even a 10 controller SDBFT deployment resulting in a lower flow setup time that a BFT-SMaRt deployment with just 4 controllers.

8.7 Deployment of Physical Proxy

In Section 8.6 I demonstrated that the SDBFT proxy works with a physical SDN switch. However, in that test the proxy was deployed on the same server as the controllers, with a network between the switch and the proxy. In this test, I explore the feasibility of deploying the proxy on a physical device as a hardware proxy, connected directly to the switch. This is a potential solution to applying the SDBFT protocol to older devices without modifying the device firmware.

8.7.1 Setup

I use a single switch, using the same topology as seen in Figure 8.11a, however I use one of the four Raspberry Pis as the proxy, resulting in the topology seen in Figure 8.14. The proxy is launched on the RPi over SSH. In this test, I perform two pings between H1 and H2, and record the flow setup time of the second ping. Note that as the control machine can no longer access the remote management interface of the switch, the starting and stopping of the OpenFlow instance on the switch is performed through the Pi. In a real-world deployment, the hardware

proxy can also proxy these connections. Each test is repeated 20 times. The `LearningSwitch` Floodlight application was used for routing. SDBFT was run with 4 controllers. Each test was repeated 20 times.

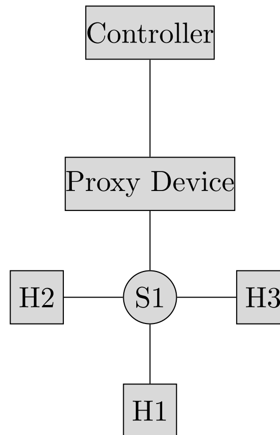


Figure 8.14: Hardware proxy topology

8.7.2 Results

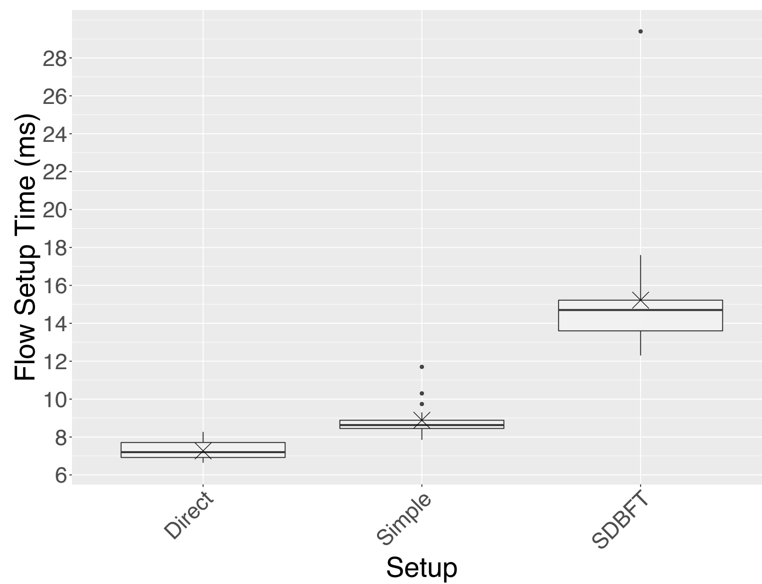


Figure 8.15: A direct switch to controller connection, and through the simple and SDBFT proxies running on a Raspberry Pi. X=mean. 20 repetitions.

A direct connection from the switch to the controller has a median flow setup time of 7.27ms. Introducing the Pi based proxy, running the simple TCP proxy, results in a median flow setup time of 8.63ms (μ=8.89ms, σ=0.86), a statistically significant increase (with large effect size) of 18.7% (Mann-Whitney U: $U = 72, z = -8.2188, p < 0.0001$, effect $= 0.81(L)$). This is a larger increase of introducing the simple TCP proxy, with a larger effect size, than when the proxy was deployed on the same host as the controllers in the physical switch tests (Section 8.6). The extra latency is due to the extra physical hop on the network introduced by the RPi, as well as a lower CPU clock speed of the RPi compared to the server-grade machine the physical switch tests were performed on.

The median flow setup time for SDBFT is 14.7ms (μ=15.22, σ=3.56), a statistically significant increase with moderate effect size on the direct controller connection (Mann-Whitney U: $U = 59, z = -5.7589, p < 0.0001$, effect $= 0.68(M)$). This is a 102% increase over the direct connection, compared to 60.3% for SDBFT with 4 controllers in the baseline test, and 42.17% in the physical switch tests. This larger increase compared to the simple proxy is caused by CPU differences between the 2 tests. The Raspberry Pi 4 Model B uses a Quad core Cortex-A72 (ARM V8) CPU with a clock speed of 1.5GHz, compared to the server on which the baselines were run, which uses 4 AMD Opteron 6376 16 core 2.3GHz CPUs.

8.7.3 Discussion

This test aimed to demonstrate the effectiveness of the SDBFT proxy when deployed as a physical device, using a Raspberry Pi to host the proxy. The results clearly show that this is a viable option, even with the relatively cheap hardware provided by the Raspberry Pi. The newer generation Raspberry Pis, which feature quad core CPUs, can handle the SDBFT proxy, which is a heavily threaded application, well, with only a small increase in latency when compared to baseline tests performed on a single machine.

8.8 Network Traffic Load

One of the advantages of the SDBFT system is that only two rounds of communication are required for a client to receive a response from a server, when compared the the 3 BFT rounds used by BFT-SMaRt. In this test, I run each protocol a number of times and capture the network packets sent between the SDBFT proxy and the controllers (in the case of SDBFT), and the number of packets between the BFT-SMaRt client proxy and server proxies, and the communication between the server proxies, in the case of BFT-SMaRt. I also measure the number of packets sent during a normal, direct connection between the switch and controller.

For a direct connection, I would expect that there would be two `PacketIn` messages from the switch to the controller, one for each direction of the flow. I would expect two responses from the controller for each `PacketIn`: a `PacketOUT` and a `FLowMod`, for a total of 6 messages. When using SDBFT with 4 controllers, this should increase to 24. When using BFT-SMaRt, I expect these 24 packets, plus the communication between the replicas, which should equal 32 transmissions per request.

8.8.1 Setup

For this test I use the OVS testbed described in 7.1.1. I use a single switch topology, with two hosts. I then perform 2 pings between the pair of hosts. I start a tcpdump capture before the second ping, and terminate as soon as the ping is complete. I then use tshark to output the packet counts per TCP port. For SDBFT, I count packets where the source or destination port are the OpenFlow port used by the controllers (on the OVS setup all controllers use port 6653, so I count all packets to this port number). For BFT-SMaRt, I count packets between any of the assigned replica ports defined in the `hosts.config` file, as well as the port assigned to the server proxy for non-bft communication (which is set in the server proxy configuration file). This allows me to collect traffic both from the client to the replicas, and between the replicas. I repeat these tests 20 times each.

Table 8.7: Packet Counts

Controller	Min	Max	Median	Mean	Std. Dev.
Direct	33	40	40	38.35	2.16
SDBFT 4	95	147	106	111.6	15.95
SDBFT 10	231	346	272	278.4	38.91
BFT 4	203	232	210.5	213.5	8.68
BFT 10	1770	2165	2033	2027.9	70.46

The network capture is captured on the `any` interface. There are a large number of TCP retransmissions and duplicate ACKs within the capture files. These are caused by the Wireshark capture collecting messages across the different virtual switches, which represent multiple interfaces in the capture. I cleanup all files using tshark to remove these retransmissions and duplicates, by using a `not tcp.analysis.retransmission && not tcp.analysis.duplicate_ack` filter.

8.8.2 Results

The results of this test are presented in Table 8.7. For a direct connection, the median number of packets is 40 (μ=38.35, σ=2.15). This is substantially higher than my estimate of 6, however analysis of the packet capture reveals that in that approximately 7 second period that packet capture was live for, 26 OpenFlow packets were captured. This includes two echo requests from the controller to the switch (which generate 4 packets total with the responses), an extra pair of `PacketIN` messages for a flow from one of the virtual hosts to the switch, and a number of `PacketOUT` messages sent by the controller as part of host discovery.

When using SDBFT with 4 controllers, the number of packets increases to 106, a 165% increase in the number of packets. This is roughly as expected. Whilst a direct connection requires 6 packets to be sent between the switch and controllers, using SDBFT with 4 controllers multiplies this by 4, meaning a single flow setup should instead take 24 packets. Increasing to 10 controllers should increase the number of packets required to 60, there is an average of 272 packets in out test, an increase of 580%.

BFT-SMaRt with 4 controllers generates on average 210.5 packets, a 426.25% increase on a direct connection. If I look at the BFT smart protocol [22], there

are two broadcast rounds each requiring n^2 packets to be sent (where n is the number of replicas), as well as n messages from the client to the replicas in the initial request, $n-1$ messages sent from the leader to the other replicas in the prepare stage, and a further n messages when the replicas respond to the client. For a 4 replica setup, a single request then generates 43 messages, which is itself a 616% increase over a direct connection, and so the packet counts are in line with this.

BFT-SMaRt with 10 controllers represents a much larger increase in packet counts, with a median of 2033 packets. This is an increase of 4982.5% over a direct connection. BFT-SMaRt with 10 controllers should generate 239 packets per request (largely caused by two broadcast rounds of 100 packets each), which is a 2883% increase over a direct connections 6 packets. It is not entirely clear where the extra messages come from, though they are all sent between the replicas through the BST-SMaRt connections. Through analysing the packet captures it is unclear exactly what causes these extra messages, but it is worth considering that a single extra request passing through the BFT system generates 239 extra messages, even if no response is received, which can quickly bring the total number of packets up, an effect which is heavily reduced when using SDBFT or a direct connection.

8.8.3 Discussion

In this test I aim to get an estimation of how much extra network load is introduced through the use of the SDBFT and BFT-SMaRt protocols. I find that SDBFT performs roughly as expected, with a roughly 3.5x increase in the number of packet when using 4 controllers, and 8.5x when using 10 controllers. As the SDBFT proxy broadcasts messages to all replicas, this is roughly as expected. The variance is caused by traffic noise — captures contain all packets between the proxy and switch, which includes packets not part of the flow setup. I show that BFT-SMaRt, as expected, results in a far greater increase in packet counts due to the repeated broadcasts.

8.9 Conclusion

In this chapter I aimed to evaluate the performance of the implementation of the SDBFT protocol, and provide a comparison with both the traditional SDN model of a direct switch to single controller connection, as well as to the comparative BFT implementation built using BFT-SMaRt.

In Section 8.2 I showed that SDBFT performs well on a small scale topology, providing substantially faster flow setup times when compared to BFT-SMaRt, and close to the traditional model. I show that SDBFT scales well with an increasing number of controllers forming the primary quorum, and show the impact of introducing signed messages into the protocol. I expanded this testing to multi-hop paths in Section 8.3.

Section 8.4 evaluated SDBFT when under attack. In these tests, I showed that whilst SDBFT suffers a substantial increase in flow setup time on the initial flow that is attacked, once the failover has been initialised the system returns to close to normal performance. I also demonstrated that, when 3 out of 4 controllers are compromised, the SDBFT protocol is able to identify a fault has occurred and successfully trigger the failover protocol. This confirms that SDBFT can operate with $f + 1$ replicas, compared to the $3f + 1$ required by traditional BFT algorithms such as BFT-SMaRt.

in Section 8.5 I used the Cbench controller benchmarking tool to perform high throughput testing of the SDBFT proxy implementation. I find that, whilst it generally performs better than BFT-SMaRt, that there are potential implementation issues caused by the use of Java sockets and threading rather than a more efficient alternative. These particular results can help lead future development of the SDBFT implementation.

I also demonstrated SDBFT deployed against physical switches (Section 8.6, and show that SDBFT is able to outperform BFT-SMaRt and operate with a small extra amount of latency when compared to a direct controller connection. Further, I found that SDBFT performs well when deployed on a Raspberry PI as a hardware proxy, as shown in Section 8.7.

Finally, when comparing the network loads caused by the SDBFT and BFT-SMaRt protocols, I find that SDBFT performs as expected, with a substantial increase in packet counts when compared to a direct controller connection, but substantially less that when using BFT-SMaRt, in particular when using 10 controllers. These results were presented in Section 8.8.

Chapter 9

Conclusion

In this thesis I propose a solution to the problem of malicious controllers in software-defined networks. Through the centralisation of the network control plane into an SDN controller running as software, a single point of failure is introduced which can cause disruption to the network either through genuine fault, or through compromise of the SDN controller. A compromised SDN controller can gain large amounts of, if not total, control of the underlying network, including the routing and other network functions such as firewalls. The actions of a compromised controller within the network manifest as byzantine (or arbitrary) faults within the network control.

The proposed protocol, Software-Defined Byzantine Fault prevenTing control (SDBFT), solves this problem by introducing replication to the SDN control plane and the application of an efficient byzantine fault-preventing (fault detecting with recovery) protocol in order to prevent malicious controllers from installing malicious flow rules onto switches. Compared to previous solutions that utilise traditional byzantine-fault tolerant algorithms such as PBFT, requiring $3f + 1$ controllers and multiple rounds of communication to handle f faulty (or malicious) controllers, SDBFT requires only $2f+1$ controllers, utilising only $f+1$ under normal operations, with a further f controllers brought in as backup when a failure occurs, operating over just 2 rounds of communication. Experimental evaluation shows that it is able to operate with a minimal impact of flow setup time when compared against a normal direct controller connection, and provides

much lower flow setup times when compared to approaches utilising traditional BFT algorithms.

9.1 Thesis Contributions

9.1.1 Exploration of Attack Capabilities From a Compromised SDN Controller

I provided a threat model of the attacker who would compromise an SDN controller, including the types of attacker, the method of compromising the SDN controller and the goals of the attacker.

I then examined a number of attacks that can be launched from a malicious controller to achieve the goals of the attacker. These include both new attacks and existing attacks from the literature. To the best of my knowledge, this is the widest-ranging exploration of practical attacks that can be launching from a compromised SDN controller.

9.1.1.1 Practical Demonstration of Attacks

In order to practically demonstrate the impact of the described attacks, I implemented the discussed attacks through a set of malicious applications for the Floodlight SDN controller, and a virtual network built upon the Mininet platform. I demonstrated a number of attacks that can introduce additional latency into the network, including by directing all traffic for a target flow through the SDN controller, and by directing traffic over a non-optimal routes in the network. I also demonstrated denial-of-service attacks including a simple sinkhole attack, and a resource-consumption denial-of-service attack which duplicates flow in the the network to cause debilitating network congestion.

9.1.1.2 Impact of Attacks on Industrial Control Systems (ICS)

In order to demonstrate the impact of the attacks on a real-world scenario, I also measured the impact of the attacks on the use case of ICS. ICS is a particularly

interesting use case due to the reliance on real-time protocols for communication between devices, which provides susceptibility to attacks which can introduce additional latency into the network without blocking communication. Whilst the use of SDN (and programmable networks in general) is currently limited within ICS environments, it is increasingly being proposed as a future direction for ICS environments due to the additional control provided by SDN.

I showed that the real-time Siemens Profinet industrial protocol is very susceptible to attack, with only the small amount of latency of requiring all packets to pass through the controller enough to break the real-time properties of the communication, causing devices to disconnect. For the non real-time protocols of S7Comm, Modbus and Ethernet/IP, introducing extra latency through the SDN controller does not cause a failure in communication, but does cause problems with the underlying physical process due to the additional latency.

I also showed that a compromised SDN controller can be used to facilitate person-in-the-middle attacks on the industrial protocols.

To the best of my knowledge, this is the first exploration of the impact of a compromised SDN controller on ICS networks. This work has been published as "Controller-in-the-Middle: Attacks on Software Defined Networks in Industrial Control Systems" [87].

9.1.2 Design of a Consensus-Based Distributed Controller Architecture to Prevent Malicious Insiders

To prevent an compromised SDN controller from carrying out malicious control actions within a network, I designed a consensus-based byzantine fault-tolerant protocol, Software-Defined Byzantine Fault Tolerant control (SDBFT), which is able to prevent malicious control with a minimal impact on performance.

Existing approaches for providing byzantine-fault tolerance to the SDN control plane utilise traditional BFT algorithms derived from the PBFT algorithm by Liskov and Castro [41]. Such algorithms require a minimum of $3f + 1$ controllers to handle f faulty (or malicious) controllers, and require multiple rounds

of communication to operate. The SDBFT protocol only requires $2f + 1$ controllers ($f+1$ during normal operation with a further f when a fault is detected), allowing more controllers to become faulty yet detected, and just two rounds of communication in the normal case by relaxing the requirement to handle faulty nodes in the normal operation of the protocol. This reduction in the required number of controllers results in an overall reduction in the number of replica controllers required in the system. Instead, if there is any disagreement between the set of controllers, then a backup quorum of controllers is incorporated and the majority response is taken, only requiring one additional round of communication. This works on the assumption that for the majority of time, the SDN control architecture would not be compromised and so reducing the complexity during normal operation to provide more efficient network control is worthwhile with the tradeoff of higher latency on the occurrence of a fault.

I expanded the SDBFT protocol with signatures in order to provide authentication of controllers and non-repudiation, allowing the verification of malicious controllers and preventing packet tampering on the switch to controller connection. I also described a method for state synchronisation between controllers in order to ensure backup controllers have a consistent view of the network state.

9.1.2.1 Implementing and Evaluating the SDBFT Architecture

I implemented the SDBFT protocol as a Java-based proxy to enable testing with a variety of switch types. This proxy, which sits between the switch and controllers, operates the SDBFT protocol without having to modify the switch itself. I evaluated the performance of SDBFT in comparison with a direct switch to controller connection, a simple TCP proxy and an implementation of a traditional BFT based approach utilising the BFT-SMaRt Java library. This utilised three test environments — a simulated network based upon the Mininet network simulation tool, a virtualised environment using OpenVSwitch virtual switches, and a physical testbed using Dell OpenNetworking hardware SDN switches.

On testing with a single virtual switch, I showed that the SDBFT protocol, with a single controller and no signature used, resulted in only a small increase in flow setup time over a direct controller connection, a latency which could

be reduced if the SDBFT protocol was implemented natively on a switch, as I demonstrated that the simple act of introducing a proxy between the switch and controller is responsible for a large proportion of the additional flow setup time. Utilising 4 and 10 controllers with the SDBFT protocol results in great. but manageable, additional flow setup time over a direct controller connection. Comparatively, the BFT-SMaRt implementation with 4 and 10 controllers resulted in a a much larger increase in flow setup time over the direct connection, significantly more than the SDB FT protocol. I also showed that, as expected, introducing signatures into the protocol increases latency, resulting in 3x additional latency over the unsigned SDBFT.

When tested with multi-hop paths of increasing lengths from 1 to 10 switches in the Mininet environment, I show that the SDBFT protocol results in a small additional flow setup time for a 1-hop and 10-hop path over the direct connection respectively, with BFT-SMaRt resulting in statistically significantly higher flow setup times than SDBFT. Both approaches show a reduced impact on flow setup time with longer path lengths.

When testing under failure, I demonstrated the intended performance of the SDBFT protocol in that a large amount of latency is introduced on setting up the flow where the flow occurs due tot he second round of communication with the backup controllers, which drops back to normal levels for subsequent flows as the backup controllers are utilised in the same round of communication as the primary set. As expected, the BFT-SMaRt implementation handles the faults with no impact on performance, however this is still represents a greater flow setup time that the SDNFT protocol. I also demonstrated that the SDBFT protocol was able to successfully operate with all but one of the primary controllers becoming malicious, whereas in the same situation the BFT-SMaRt approach results in malicious flow rules being installed within the network.

I performed a high throughput benchmark of the SDBFT protocol to measure performance under a high network load. I showed that both the SDBFT and BFT-SMaRt protocols resulted in a major drop performance when compared to a direct controller connection, which indicates that consideration should be given to the use of fault-tolerant approaches in highly dynamic networks which require a large amount of controller input to switches.

9.1.3 Research Impact

Firstly, this thesis demonstrates the need to consider insider threats within SDN, and in particular the issue of compromised controllers. With the increasing move to SDN across various use cases and the centralisation of network control and functions into the SDN controller comes a tempting target for attackers. I demonstrate the impact of these attacks through practical experimentation, clearly showing the damage that can arise from a compromised SDN controller.

Secondly, I propose a solution to the problem of compromised SDN controllers that is successfully able to prevent a compromised controller from influencing the operation of a network. In particular, the proposed solution, SDBFT, is able to handle a greater number of faulty nodes with fewer replicas compared to previous works which utilise traditional byzantine fault tolerant algorithms. Through extensive experimentation I demonstrate that my approach is able to operate with a minimal impact on flow setup time compared to existing work, and is certainly feasible for use in real-world networks. This is backed by a complete implementation of the protocol as a network proxy which allows the SDBFT protocol to be applied to existing switch hardware without modification to the switch itself.

9.1.4 Summary

In summary, this thesis has made the following contributions:

- Explored the practical impact of a compromised SDN controller on networks.
- Provided the first exploration of the impact of malicious SDN controllers in the context of ICS.
- Proposed the design of an efficient protocol for providing byzantine fault tolerance in the SDN control plane.
- Realising the designed protocol as a prototype implementation consisting of a Java-based network proxy and a modified version of the Floodlight SDN controller, allowing testing of the protocol with multiple types of SDN switches.

- Performed extensive evaluation of the prototype implementation on multiple types of SDN network, providing a comparison against the traditional SDN model, and a comparative system built using a traditional byzantine fault-tolerant approach.

9.2 Future Work

9.2.1 Proactive Control

In this work I have focussed on the reactive SDN control model, in which the switch sends a request to the controllers which then generate a response. The alternative to reactive control is proactive, where the controllers push updates to switch without a corresponding request from the switch. This reduces the load on the controller. In reality, many deployments utilise a hybrid approach where proactive rules are installed for known devices and routes, with new, previously unseen devices then handled in a reactive manner.

The current SDBFT protocol relies on a reactive model in order to provide fault tolerance, and controller responses are mapped to a particular switch request. The switch triggers an event, and the controller is expected to respond in a short period. Part of the challenge of a proactive mode is that proactive controller commands may not occur from all controllers simultaneously (this partially depends on the level of determinism in the set of applications).

The actual act of matching controller commands is straight forward, as if all controller commands match then the logic for testing for consensus is the same as testing the responses to a switch request. The difficulty is that with the current SDBFT approach, backup controllers do not communicate with switches and so if there is disagreement and the backups need to be contacted, there is no switch request to be forwarded. This means that a protocol needs to be developed that allows the backups to generate a proactive command on demand. This could take the form of the switch forwarding the received set of proactive commands to the backups, which then analyse the commands and pick one of the received ones to use.

9.2.2 Controller Verification

Currently in the case where a controller exhibits faulty/malicious behaviour, the extended SDBFT protocol is used until an administrator can investigate and remove or repair the faulty controller.

A more efficient approach could be to perform an automated verification of the switch-controller communication in order to identify the faulty or malicious controller, and remove them from the set of controllers until they can be repaired, allowing a return to the standard protocol. This could be done by the primary and backup set of controllers for a switch, on observing a fault, sending a report to the complete pool of controllers which can then perform verification over the controller actions in order to identify the faulty controllers. This information can be forwarded to the switch which can then sever connections to the controllers which are identified as malicious.

This has the challenge of ensuring that the pool of controllers has sufficient information in order to be able to verify controller responses. For example, controllers in the wider pool may not have up-to-date information about the complete state of the network outside of the switches that they control. The decision made by a controller can change based upon the information it has, and so verification can only occur if the verifier is making its judgement based upon the same information the suspect controller had at the time it made its decision.

9.2.3 Anonymous Information Sharing

In the current implementation of the SDBFT protocol, controllers within the network can observe the identities of controllers within the primary and backup quorums for a given switch by observing the consistency protocol used to share network updates between controllers. This could potentially allow an attacker who is in the network to learn which controller should be targeted in order to affect the operation of a particular target switch.

Future versions of the SDBFT system could be expanded to allow the sharing of the information required to maintain consistency (switch and host connectivity

information) in an anonymous fashion to prevent the controller from being identified. This comes with the additional challenge in authenticating this information without revealing the source controller in order to prevent a malicious insider from publishing false updates to the network.

9.2.4 Native Implementation of SDBFT

The current implementation of the SDBFT protocol relies on a Java-based TCP proxy running separately from the switch itself. Whilst this made the implementation much simpler for testing the core protocol, and also more versatile for testing in different scenarios, this has an impact on the performance of the protocol due to the extra networking and packet processing overhead. The next step is the implementation of SDBFT is to produce a native implementation on an SDN switch, moving the packet processing required for SDBFT onto the switch itself, removing the need for the proxy and the additional packet processing required for it. A logical first step for this could be to implement the protocol into the OpenVSwitch virtual switch environment as this is an open source project and runs on general purpose hardware.

9.3 Reproducibility

It is important to support reproducibility within scientific research. To allow this, the code required to reproduce the results of this work will be released onto GitHub[1].

- The documented source code of the implemented proxy, which includes the implementation of the following:
  - Simple TCP proxy
  - The SDBFT proxy
  - The BFT-SMaRt proxy (client and server)
  - Configuration files for each type of testing

[1] `https://github.com/josephgardiner/sdbft`

- An implementation of the Floodlight controller, with SDBFT modifications included and a set of malicious applications and suitable configuration files
- The Mininet and bash scripts used for testing
- Details on recreating the setup for the Industrial Control Systems testbed used in Chapter 4.5, including project files where possible.

This code is provided with instruction, and will allow third parties to replicate the experiments described in this thesis, including the comparative tests, with minimal effort. The Mininet software used for testing is readily available and well documented for installation and operation. Instructions on how to reproduce the virtual environment used in baseline testing are also provided.

9.4 Concluding Remarks

In this thesis I designed an efficient, byzantine fault-tolerant protocol, SDBFT, for preventing the actions of compromised controllers in software-defined networks. Through the use of a prototype proxy-based implementation, I was able to extensively evaluate the performance of the proposed approach and show favourable performance against the existing approach taken by the literature, and only a minimal drop in network performance compared to the traditional SDN control model.

The SDBFT protocol can be applied to existing SDN networking hardware through the use of a network proxy implementation, or could be incorporated into future SDN switches natively with a smaller impact on network performance. Whilst the system would not work for all network types, in particular highly dynamic, high throughput networks which require extensive amounts of controller interaction, the proposed approach could prove valuable in securing software-defined networks.

Appendix A

Implementation

A.1 SDBFT Proxy Configuration

Listing A.1 presents an example configuration file for the SDBFT proxy. The configuration is used by both the SDBFT and BFT-SMaRt implementations, which are both included within the same project. The “version” option specifies which version of the proxy should be used, selected from '“simple” (for a simple TCP proxy), ”unsigned” and ”signed” (for the SDBFT proxy) or ”bftclient” and “bftserver” (for the BFT-SMaRt proxy). Note that as the configuration file is used for multiple variations of the proxy, not all options are required.

```
version=unsigned
localport=55413
controllers=[\
{"id": 1, "ip": "127.0.0.1", "port":6653},\
{"id": 2, "ip": "127.0.0.1", "port":6654},\
{"id": 3, "ip": "127.0.0.1", "port":6655},\
{"id": 4, "ip": "127.0.0.1", "port":6656}\
]
loadbackups=false
backupcontrollers=[\
{"id": 1, "ip": "127.0.0.1", "port":6657},\
```

```
{"id": 2, "ip": "127.0.0.1", "port":6658},\
{"id": 3, "ip": "127.0.0.1", "port":6659},\
{"id": 4, "ip": "127.0.0.1", "port":6660}\
]
signaturetype=SHA256withRSA
signaturekey=RSA
keysize=512
siglength=64
ofversion=1.0
batchacks=True
startxid=600000
bftid=12
bftclientid=1111
```

Listing A.1: Example SDBFT configuration file

Appendix B

Evaluation Setup

B.1 OVS Test Launch Script

Listing B.2 shows an example bash script for launching four instances of the Floodlight controller on four VMs, starting the SDBFT proxy, setting the controller of the OVS bridge to the proxy and performing ping tests between cirrOS hosts, logging the outputs to a text file.

```
#!/bin/bash
ssh netsec@192.168.122.142 "cd finalfloodlight ; java -jar  -server
    -XX:+UseCompressedOops  target/floodlight.jar -cf
    target/bin/floodlightdefault.properties &" &
ssh netsec@192.168.122.143 "cd finalfloodlight ; java -jar  -server
    -XX:+UseCompressedOops  target/floodlight.jar -cf
    target/bin/floodlightdefault.properties &" &
ssh netsec@192.168.122.144 "cd finalfloodlight ; java -jar  -server
    -XX:+UseCompressedOops  target/floodlight.jar -cf
    target/bin/floodlightdefault.properties &" &
ssh netsec@192.168.122.145 "cd finalfloodlight ; java -jar  -server
    -XX:+UseCompressedOops  target/floodlight.jar -cf
    target/bin/floodlightdefault.properties &" &
sleep 5
```

```
java -jar -XX:+UseCompressedOops  -XX:+UseNUMA  proxy.jar proxy.properties &
sleep 5
ovs-vsctl set-controller br1 tcp:127.0.0.1:55413
sleep 12
ssh cirros@172.16.1.174 "ping -c 4 172.16.1.175"
sleep 6
ssh cirros@172.16.1.171 "ping -c 4 172.16.1.173" >> "test$1.txt"
sleep 6
ssh cirros@172.16.1.171 "ping -c 4 172.16.1.173" >> "test$1.txt"
ssh netsec@192.168.122.142 "pkill -9 java"
ssh netsec@192.168.122.143 "pkill -9 java"
ssh netsec@192.168.122.144 "pkill -9 java"
ssh netsec@192.168.122.145 "pkill -9 java"
pkill -9 java
```

Listing B.1: Bash script for launching controllers, proxy and performing ping tests in OVS setup

B.2 Mininet Python Configuration Example

The Python example code presented in Listing B.2 creates a mininet network consisting of a single switch, connecting to a single controller. IPv6 is disabled on the switch due to limited support by the Floodlight controller. Two hosts are added, with the default IP addresses `10.0.0.1` and `10.0.0.2`, and are connected to the switch Once the network has been built, host 1 pings host 2, with the output printed to a file. The script waits for 6 seconds (to allow the flow rule to expire on the virtual switch), and then repeats the ping, printing the output to the same file. The network is then shut down.

```
from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch, OVSKernelSwitch,
    RemoteController
from mininet.link import TCLink
```

```
from mininet.cli import CLI
from mininet.log import setLogLevel
import sys
import time

def multiControllerNet(num):
    "Create a network from semi-scratch with multiple controllers."

    net = Mininet( controller=RemoteController, switch=OVSSwitch,
    build=False, autoSetMacs = True, autoStaticArp = True)

    print "*** Creating (reference) controllers"
    c1 = net.addController('c1', controller=RemoteController,
ip="127.0.0.1", port=55413)

    print "*** Creating switches"
    s1 = net.addSwitch( 's1' , protocols=["OpenFlow13"])

    s1.cmd("sysctl -w net.ipv6.conf.all.disable_ipv6=1")
    s1.cmd("sysctl -w net.ipv6.conf.default.disable_ipv6=1")
    s1.cmd("sysctl -w net.ipv6.conf.lo.disable_ipv6=1")

    print "*** Creating hosts"

    h1 = net.addHost( 'h1' )
    h2 = net.addHost( 'h2' )

    net.addLink( s1, h1 )
    net.addLink( s1, h2 )

    print "*** Starting network"
    net.build()
```

```
    c1.start()
    s1.start( [ c1 ] )
    print "***Setup Done"
    time.sleep(25)
    print "*** Testing network"
  #  net.pingAll()
    f = open("results/1/test" + num + ".txt", "a")

    res1=h1.cmd('ping -c 4 10.0.0.2')
    print(res1)
    f.write(res1)
    time.sleep(6)
    res2= h1.cmd('ping -c 4 10.0.0.2')
    print(res2)
    f.write(res2)
    f.close()

    print "*** Stopping network"
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )  # for CLI
    multiControllerNet(sys.argv[1])
```

Listing B.2: Mininet Python Configuration, Single Switch to Single Controller

B.3 Bash Launch Script

Listing B.3 shows the bash script used to launch four Floodlight instances, the SDBFT proxy and the mininet python script found in Appendix B.2.

```
#!/bin/bash
mn -c
```

```
java -jar  -server -XX:+UseCompressedOops
    singlehopcontrollers/floodlight1/target/floodlight.jar -cf
    singlehopcontrollers/floodlight1/target/bin/floodlightdefault.properties &
java -jar  -server -XX:+UseCompressedOops
    singlehopcontrollers/floodlight2/target/floodlight.jar -cf
    singlehopcontrollers/floodlight2/target/bin/floodlightdefault.properties &
java -jar  -server -XX:+UseCompressedOops
    singlehopcontrollers/floodlight3/target/floodlight.jar -cf
    singlehopcontrollers/floodlight3/target/bin/floodlightdefault.properties &
java -jar  -server -XX:+UseCompressedOops
    singlehopcontrollers/floodlight4/target/floodlight.jar -cf
    singlehopcontrollers/floodlight4/target/bin/floodlightdefault.properties &
sleep 5
java -jar -XX:+UseCompressedOops  -XX:+UseNUMA  proxy.jar proxy.properties &

sleep 5
python 1hop.py $1
pkill -9 java
sleep 2
```

Listing B.3: Bash script for launching controllers and mininet (run as root)

Appendix C

Evaluation

This appendix contains tables containing the complete results of various experiments performed in Chapter 8, including the results of various statistical significance tests. Effect size classified according to Cohen [53, 222]:

Effect Size	**Classification**
$<$0.2	None (S)
0.2-0.49	Small (S)
0.5-0.79	Moderate (M)
0.8-1.29	Large (L)
$>$1.3	Very Large (VL)

C.1 Baseline Results

Table C.1: Baseline results without using signatures. Mann-Whitney U test against direct controller. Mean, SD and median results in milliseconds. N = Number of controllers

					Mann-Whitney U			
Approach	**N**	**Mean**	**SD**	**Median**	**U**	**Z**	**P**	**Effect**
Direct	1	13.53	5.87	12.18	-	-	-	-
Simple	1	15.63	7.60	13.98	578	-4.62923	<0.0001	0.46 (S)
SDBFT	1	16.05	2.01	15.75	2286	-7.13857	<0.0001	0.71 (M)
	2	18.97	3.75	17.96	2356	-7.62112	<0.0001	0.76 (M)
	3	20.06	3.71	19.21	2372	-7.73142	<0.0001	0.77 (M)
	4	22.37	7.29	19.53	2379	-7.77982	<0.0001	0.78 (M)
	5	23.42	6.25	19.95	2387	-7.83485	<0.0001	0.78 (M)
	6	25.16	9.26	20.90	2396	-7.89687	<0.0001	0.79 (M)
	7	26.37	8.44	22.91	2404	-7.95202	<0.0001	0.80 (L)
	8	26.40	6.50	23.53	2412	-8.00717	<0.0001	0.80 (L)
	9	26.80	6.95	26.09	2415	-8.02785	<0.0001	0.80 (L)
	10	30.00	8.32	28.92	2424	-8.08990	<0.0001	0.81 (L)
BFT Ordered	4	43.70	25.03	38.50	46	-8.29671	<0.0001	0.83 (L)
	5	51.94	34.09	42.12	41	-8.33118	<0.0001	0.83 (L)
	6	66.00	52.75	42.96	31	-8.40012	<0.0001	0.84 (L)
	7	96.51	71.33	71.61	13	-8.52421	<0.0001	0.85 (L)
	8	94.75	67.26	70.44	17	-8.49663	<0.0001	0.85 (L)
	9	61.63	39.40	48.07	30	-8.40701	<0.0001	0.84 (L)
	10	66.27	38.98	53.14	21	-8.46906	<0.0001	0.85 (L)
BFT Unordered	4	26.61	2.49	26.39	79	-8.06922	<0.0001	0.81 (L)
	5	27.15	5.47	26.30	129	-7.72453	<0.0001	0.77 (M)
	6	26.69	4.19	27.26	120	-7.78657	<0.0001	0.78 (M)
	7	28.11	2.13	27.92	63	-8.17952	<0.0001	0.82 (L)
	8	27.89	2.62	27.61	69	-8.13815	<0.0001	0.81 (L)
	9	28.83	2.09	28.42	58	-8.21399	<0.0001	0.82 (L)
	10	30.45	8.72	29.50	100	-7.92445	<0.0001	0.79 (M)

Table C.2: Baseline Mann-Whitney U test results comparing increasing controller counts, no signatures. Numbers in test columns represent controller count.

		Mann-Whitney U			
Test 1	**Test 2**	**U**	**Z**	**P**	**Effect**
SDBFT 4	SDBFT 10	2002	-5.18080	<0.0001	0.52 (M)
BFT Ordered 4	BFT Ordered 10	2332	-7.45567	<0.0001	0.75 (M)
BFT Unordered 4	BFT Unordered 10	2183	-6.42849	<0.0001	0.64 (M)
SDBFT 1	SDBFT 2	404	-5.82874	<0.0001	0.58 (M)
SDBFT 2	SDBFT 3	890.5	-2.47489	0.01333	0.25 (S)
SDBFT 3	SDBFT 4	1122	-0.87898	0.37941	0.09 (N)
SDBFT 4	SDBFT 5	886	-2.50595	0.01221	0.25 (S)
SDBFT 5	SDBFT 6	1095	-1.06510	0.28683	0.11 (N)
SDBFT 6	SDBFT 7	1048	-1.38910	0.16480	0.14 (N)
SDBFT 7	SDBFT 8	1154	-0.65836	0.51031	0.07 (N)
SDBFT 8	SDBFT 9	1270.5	-0.13788	0.89034	0.01 (N)
SDBFT 9	SDBFT 10	1547	-2.04402	0.04095	0.20 (S)
BFT Ordered 4	BFT Ordered 5	350	-6.20099	<0.0001	0.62 (M)
BFT Ordered 5	BFT Ordered 6	1131	-0.81692	0.41398	0.08 (N)
BFT Ordered 6	BFT Ordered 7	831	-2.88506	0.00391	0.29 (S)
BFT Ordered 7	BFT Ordered 8	1271	-0.14132	0.88761	0.01 (N)
BFT Ordered 8	BFT Ordered 9	1596	-2.38181	0.01723	0.24 (S)
BFT Ordered 9	BFT Ordered 10	1561	-2.14053	0.03231	0.21 (S)
BFT Unordered 4	BFT Unordered 5	1101	-1.02373	0.30596	0.10 (N)
BFT Unordered 5	BFT Unordered 6	1157	-0.63768	0.52368	0.06 (N)
BFT Unordered 6	BFT Unordered 7	905	-2.37492	0.01755	0.24 (S)
BFT Unordered 7	BFT Unordered 8	1364	-0.78245	0.43395	0.08 (N)
BFT Unordered 8	BFT Unordered 9	898	-2.42318	0.01539	0.24 (S)
BFT Unordered 9	BFT Unordered 10	1562.5	-2.15088	0.03149	0.22 (S)

Table C.3: BFT-SMaRt statistical significant test against SDBFT, no signatures.

	Controllers						
Mann-Whitney U	4	5	6	7	8	9	10
U	143	27	72	119	20	56	113
Z	-7.62815	-8.42772	-8.11747	-7.79346	-8.47595	-8.22777	-7.83483
P	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
Effect	0.76 (M)	0.84 (L)	0.81 (L)	0.78 (L)	0.85 (L)	0.82 (L)	0.78 (M)

Table C.4: Baseline results using SHA512 with RSA signatures. Mann-Whitney U test against direct controller. Mean, SD and median results in milliseconds. N = number of controllers

					Mann-Whitney U			
Approach	**N**	**Mean**	**SD**	**Median**	**U**	**Z**	**P**	**Effect**
Direct	1	13.53	5.87	12.18	-	-	-	-
SDBFT	1	32.02	4.19	31.18	2450	-8.26914	<0.0001	0.83 (L)
	2	34.09	5.67	32.53	2452	-8.28292	<0.0001	0.83 (L)
	3	36.52	6.50	34.37	2452	-8.28292	<0.0001	0.83 (L)
	4	37.08	6.64	34.07	2451	-8.27606	<0.0001	0.83 (L)
	5	42.19	17.21	37.65	2457	-8.31739	<0.0001	0.83 (L)
	6	40.59	18.88	36.14	2407	-7.97270	<0.0001	0.80 (L)
	7	45.65	17.25	43.96	2459	-8.33118	<0.0001	0.83 (L)
	8	41.47	7.31	40.25	2457	-8.31739	<0.0001	0.83 (L)
	9	46.48	23.30	43.23	2461	-8.34497	<0.0001	0.83 (L)
	10	54.16	31.74	47.37	2464	-8.36565	<0.0001	0.84 (L)
BFT Ordered	4	71.66	43.39	56.77	5	-8.57936	<0.0001	0.86 (L)
	5	82.27	54.20	58.74	2	-8.60004	<0.0001	0.86 (L)
	6	79.62	47.80	62.71	1	-8.60693	<0.0001	0.86 (L)
	7	96.37	58.00	81.61	0	-8.61383	<0.0001	0.86 (L)
	8	90.30	62.37	69.19	54	-8.24156	<0.0001	0.82 (L)
	9	65.91	26.83	60.97	1	-8.60693	<0.0001	0.86 (L)
	10	64.83	8.07	63.18	0	-8.61383	<0.0001	0.86 (L)
BFT Unordered	4	42.12	5.29	41.16	49	-8.27606	<0.0001	0.83 (L)
	5	42.83	4.85	41.90	47	-8.28982	<0.0001	0.83 (L)
	6	41.06	6.47	41.52	99	-7.93134	<0.0001	0.79 (M)
	7	42.89	2.68	42.15	49	-8.27603	<0.0001	0.83 (L)
	8	45.30	5.17	44.27	45	-8.30361	<0.0001	0.83 (L)
	9	44.35	2.97	43.97	48	-8.28295	<0.0001	0.83 (L)
	10	44.57	4.88	44.02	48	-8.28295	<0.0001	0.83 (L)

Table C.5: Baseline Results Using SHA256 with RSA Signatures. Mann-Whitney U test against direct controller. Mean, SD and median results in milliseconds.

					Mann-Whitney U			
Approach	**N**	**Mean**	**SD**	**Median**	**U**	**Z**	**P**	**Effect**
Direct	1	13.53	5.87	12.18	-	-	-	-
SDBFT	1	24.05	4.76	22.35	2406	-7.96583	<0.0001	0.80 (L)
	2	27.90	6.21	25.06	2417	-8.04164	<0.0001	0.80 (L)
	3	27.63	6.30	24.92	2419	-8.05543	<0.0001	0.81 (L)
	4	26.78	5.01	24.74	2418	-8.04853	<0.0001	0.80 (L)
	5	28.42	5.50	26.39	2421	-8.06922	<0.0001	0.81 (L)
	6	31.34	7.58	28.93	2429	-8.12437	<0.0001	0.81 (L)
	7	32.01	6.63	30.50	2432	-8.14505	<0.0001	0.81 (L)
	8	33.90	7.49	31.16	2444	-8.22777	<0.0001	0.82 (L)
	9	34.40	7.45	32.64	2447	-8.24848	<0.0001	0.82 (L)
	10	34.66	7.14	32.22	2446	-8.24156	<0.0001	0.82 (L)
BFT Ordered	4	56.69	5.06	56.03	1	-8.60693	<0.0001	0.86 (L)
	5	93.53	65.34	63.34	1	-8.60693	<0.0001	0.86 (L)
	6	117.38	66.21	102.28	0	-8.61383	<0.0001	0.86 (L)
	7	60.55	17.18	55.38	3	-8.59315	<0.0001	0.86 (L)
	8	69.14	23.44	61.28	2	-8.60004	<0.0001	0.86 (L)
	9	64.95	16.78	60.66	0	-8.61383	<0.0001	0.86 (L)
	10	63.66	5.87	63.64	0	-8.61383	<0.0001	0.86 (L)
BFT Unordered	4	43.64	4.54	42.70	45	-8.30361	<0.0001	0.83 (L)
	5	42.66	1.98	42.39	50	-8.26916	<0.0001	0.83 (L)
	6	42.37	1.80	42.28	50	-8.26914	<0.0001	0.83 (L)
	7	43.66	3.06	43.01	49	-8.27603	<0.0001	0.83 (L)
	8	42.92	6.58	43.49	99	-7.93134	<0.0001	0.79 (M)
	9	45.27	5.37	44.52	48	-8.28292	<0.0001	0.83 (L)
	10	45.30	2.43	44.99	46	-8.29674	<0.0001	0.83 (L)

Table C.6: Baseline results using SHA512 with RSA signatures. Mann-Whitney U test against equivalent unsigned approach.

		Mann-Whitney U			
Approach	**Controllers**	**U**	**Z**	**P**	**Effect**
SDBFT	1	0	-8.61385	<0.0001	0.86 (L)
	2	53	-8.24846	<0.0001	0.82 (L)
	3	26	-8.43459	<0.0001	0.84 (L)
	4	230	-7.02840	<0.0001	0.70 (M)
	5	135	-7.68319	<0.0001	0.77 (M)
	6	307	-6.49743	<0.0001	0.65 (M)
	7	185	-7.33847	<0.0001	0.73 (M)
	8	112	-7.84172	<0.0001	0.78 (M)
	9	111	-7.84861	<0.0001	0.78 (M)
	10	136	-7.67627	<0.0001	0.77 (M)
BFT Ordered	4	148	-7.59354	<0.0001	0.76 (M)
	5	251	-6.88348	<0.0001	0.69 (M)
	6	627	-4.29140	<0.0001	0.43 (S)
	7	1081	-1.16161	0.2454	0.12 (N)
	8	1155	-0.65147	0.5147	0.07 (N)
	9	593	-4.52579	<0.0001	0.45 (M)
	10	706	-3.74679	0.00018	0.37 (M)
BFT Unordered	4	2500	-8.61385	<0.0001	0.86 (L)
	5	2452	-8.28292	<0.0001	0.83 (L)
	6	2450	-8.26914	<0.0001	0.83 (L)
	7	2500	-8.61383	<0.0001	0.86 (L)
	8	2500	-8.61383	<0.0001	0.86 (L)
	9	2500	-8.61385	<0.0001	0.86 (L)
	10	2402	-7.93826	<0.0001	0.79 (M)

Table C.7: Baseline results using SHA256 with RSA signatures. Mann-Whitney U test against equivalent unsigned approach.

		Mann-Whitney U			
Approach	**Controllers**	**U**	**Z**	**P**	**Effect**
SDBFT	1	46	-8.29676	<0.0001	0.83 (L)
	2	169	-7.44877	<0.0001	0.74 (M)
	3	192	-7.29021	<0.0001	0.73 (M)
	4	517	-5.04981	<0.0001	0.50 (M)
	5	641	-4.19490	<0.0001	0.42 (S)
	6	567	-4.70503	<0.0001	0.47 (S)
	7	620	-4.33966	<0.0001	0.43 (S)
	8	571	-4.67746	<0.0001	0.47 (S)
	9	583	-4.59474	<0.0001	0.46 (S)
	10	824	-2.93332	0.0034	0.29 (S)
BFT Ordered	4	162	-7.49703	<0.0001	0.75 (M)
	5	198	-7.24885	<0.0001	0.72 (M)
	6	396	-5.88388	<0.0001	0.59 (M)
	7	1639	-2.67825	0.0074	0.27 (M)
	8	1321	-0.48601	0.6270	0.05 (N)
	9	578	-4.62920	<0.0001	0.46 (M)
	10	732	-3.56755	0.0004	0.36 (M)
BFT Unordered	4	2500	-8.61383	<0.0001	0.86 (L)
	5	2450	-8.26916	<0.0001	0.83 (L)
	6	2500	-8.61383	<0.0001	0.86 (L)
	7	2500	-8.61383	<0.0001	0.86 (L)
	8	2447	-8.24846	<0.0001	0.82 (L)
	9	2500	-8.61383	<0.0001	0.86 (L)
	10	2400	-7.92447	<0.0001	0.79 9M)

C.2 Multi-hop Path Test Results

Table C.8: Multi-hop path tests without using signatures. T-test significance test against direct controller connection. Mean, SD and median results in milliseconds.

					T-Test			
Approach	**Hops**	**Mean**	**SD**	**Median**	**T**	**DF**	**P**	**Effect**
Direct	1	3.98	0.64	4.07	-	-	-	-
	2	7.44	1.23	7.36	-	-	-	-
	3	10.15	1.62	10.45	-	-	-	-
	4	14.39	1.49	14.25	-	-	-	-
	5	17.75	2.39	18.10	-	-	-	-
	6	20.14	2.57	19.9	-	-	-	-
	7	22.47	2.62	22.4	-	-	-	-
	8	25.63	3.05	25.55	-	-	-	-
	9	26.45	3.26	26.6	-	-	-	-
	10	30.36	3.34	30.9	-	-	-	-
Simple	1	4.78	1.06	4.86	-4.57884	81.01979	<0.0001	0.92 (L)
	2	9.15	1.41	9.26	-6.47428	96.17939	<0.0001	1.29 (L)
	3	12.98	1.62	13.10	-8.73321	97.99922	<0.0001	1.75 (VL)
	4	17.60	2.10	17.65	-8.81824	88.23011	<0.0001	1.76 (VL)
	5	21.78	2.89	22.25	-7.59963	94.66134	<0.0001	1.52 (VL)
	6	25.68	3.35	25.15	-9.28525	91.75634	<0.0001	1.86 (VL)
	7	29.95	3.63	29.40	-11.82885	89.25281	<0.0001	2.37 (VL)
	8	33.39	3.68	33.90	-11.46863	94.66938	<0.0001	2.29 (VL)
	9	35.89	4.75	36.95	-11.58802	86.86495	<0.0001	2.32 (VL)
	10	41.32	4.19	42.15	-14.46773	93.32191	<0.0001	2.89 (VL)
SDBFT	1	7.81	1.10	7.89	-21.25165	79.09807	<0.0001	4.25 (VL)
	2	13.10	1.72	12.90	-18.93358	88.55849	<0.0001	3.79 (VL)
	3	18.57	1.73	18.45	-25.10514	97.61997	<0.0001	5.02 (VL)
	4	25.31	2.84	25.10	-24.10981	74.05099	<0.0001	4.82 (VL)
	5	29.66	2.72	29.35	-23.27545	96.40587	<0.0001	4.66 (VL)
	6	36.79	6.41	35.90	-17.05230	64.31690	<0.0001	3.41 (VL)
	7	41.63	7.05	40.45	-18.02573	62.32604	<0.0001	3.61 (VL)
	8	44.37	6.21	43.80	-19.15106	71.30473	<0.0001	3.83 (VL)
	9	46.73	3.32	46.65	-30.79620	97.97104	<0.0001	6.16 (VL)
	10	53.72	6.84	53.65	-21.70866	71.08825	<0.0001	4.34 (VL)
BFT	1	21.95	1.44	22.10	80.34816	67.76206	<0.0001	16.07 (VL)
	2	36.85	1.35	36.85	114.12296	97.11770	<0.0001	22.82 (VL)
	3	51.35	2.27	51.80	104.25377	88.69011	<0.0001	20.85 (VL)
	4	68.02	3.26	68.50	105.81683	68.52997	<0.0001	21.16 (VL)
	5	82.84	3.79	82.80	102.65045	82.58556	<0.0001	20.53 (VL)
	6	95.97	3.00	96.40	135.78274	95.67701	<0.0001	27.16 (VL)
	7	109.82	4.46	110.00	119.36269	79.27073	<0.0001	23.87 (VL)
	8	121.86	4.99	122.00	116.44010	81.13481	<0.0001	23.29 (VL)
	9	135.60	3.96	136.00	-150.42344	94.55750	<0.0001	30.08 (VL)
	10	151.76	7.80	152.00	101.15677	66.34967	<0.0001	20.23 (VL)

Table C.9: Multi-hop path test using signatures. T-test significance test against direct controller connection. Mean, SD and median results in milliseconds.

					T-Test			
Approach	**Hops**	**Mean**	**SD**	**Median**	**T**	**DF**	**P**	**Effect**
SDBFT	1	10.51	1.54	10.50	-27.68683	65.70923	<0.0001	5.54 (VL)
	2	17.54	1.56	17.55	-36.04592	92.88393	<0.0001	7.21 (VL)
	3	24.82	3.41	24.25	-27.49073	70.19136	<0.0001	5.50 (VL)
	4	34.26	5.79	32.95	-23.49598	55.42899	<0.0001	4.70 (VL)
	5	38.47	3.31	38.40	-35.91880	89.19606	<0.0001	7.18 (VL)
	6	46.57	3.12	46.90	-46.23749	94.43678	<0.0001	9.25 (VL)
	7	53.25	7.80	51.65	-26.46955	59.95856	<0.0001	5.29 (VL)
	8	56.85	4.73	56.65	-39.24091	83.73440	<0.0001	7.85 (VL)
	9	57.23	4.14	56.65	-41.28726	92.95098	<0.0001	8.26 (VL)
	10	69.54	6.29	69.55	-38.89269	74.53187	<0.0001	7.78 (VL)
BFT	1	29.83	2.38	30.05	74.23714	56.16721	<0.0001	14.85 (VL)
	2	52.64	2.60	53.20	111.33972	69.81660	<0.0001	22.27 (VL)
	3	74.90	3.47	74.85	119.52754	69.51201	<0.0001	23.91 (VL)
	4	97.24	3.70	97.15	146.90353	64.41852	<0.0001	29.38 (VL)
	5	117.36	4.89	116.50	129.51607	71.17541	<0.0001	25.90 (VL)
	6	139.36	5.01	140.00	149.78734	73.05438	<0.0001	29.96 (VL)
	7	160.06	5.64	160.50	156.49082	69.27583	<0.0001	31.30 (VL)
	8	182.36	8.92	181.50	117.52917	60.27968	<0.0001	23.51 (VL)
	9	200.26	6.14	201.00	176.80821	74.65966	<0.0001	35.36 (VL)
	10	223.58	8.53	223.00	149.14835	63.65559	<0.0001	29.83 (VL)

Table C.10: Multi-hop path test without using signatures. T-test significance test comparing SDBFT to BFt-SMaRt.

		T-Test			
Test 1	**Test 2**	**T**	**DF**	**P**	**Effect**
SDBFT 1	BFT 1	55.07926	91.52218	<0.0001	11.02 (VL)
SDBFT 2	BFT 2	76.85285	92.72016	<0.0001	15.37 (VL)
SDBFT 3	BFT 3	81.13188	91.47875	<0.0001	16.23 (VL)
SDBFT 4	BFT 4	69.89536	96.14613	<0.0001	13.98 (VL)
SDBFT 5	BFT 5	80.55168	88.83049	<0.0001	16.11 (VL)
SDBFT 6	BFT 6	59.14316	69.52158	<0.0001	11.83 (VL)
SDBFT 7	BFT 7	57.81489	82.83928	<0.0001	11.56 (VL)
SDBFT 8	BFT 8	68.79486	93.62073	<0.0001	13.76 (VL)
SDBFT 9	BFT 9	-121.62148	95.11430	<0.0001	24.32 (VL)
SDBFT 10	BFT 10	66.81835	96.34406	<0.0001	13.36 (VL)

Table C.11: Multi-hop path test using signatures. T-test significance test comparing SDBFT to BFt-SMaRt.

		T-Test			
Test 1	**Test 2**	**T**	**DF**	**P**	**Effect**
SDBFT 1	BFT 1	48.28170	83.89063	<0.0001	9.66 (VL)
SDBFT 2	BFT 2	81.99545	80.21435	<0.0001	16.40 (VL)
SDBFT 3	BFT 3	72.82507	97.96825	<0.0001	14.57 (VL)
SDBFT 4	BFT 4	64.79139	83.28631	<0.0001	12.96 (VL)
SDBFT 5	BFT 5	94.54526	86.12298	<0.0001	18.91 (VL)
SDBFT 6	BFT 6	111.14289	82.10120	<0.0001	22.23 (VL)
SDBFT 7	BFT 7	78.51023	89.24015	<0.0001	15.70 (VL)
SDBFT 8	BFT 8	87.88877	74.49681	<0.0001	17.58 (VL)
SDBFT 9	BFT 9	136.63902	85.92238	<0.0001	27.33 (VL)
SDBFT 10	BFT 10	102.73519	90.15558	<0.0001	20.55 (VL)

References

[1] A. E. Abbadi and S. Toueg. “Maintaining Availability in Partitioned Replicated Databases”. In: *ACM Trans. Database Syst.* 14.2 (1989), 264–290. ISSN: 0362-5915. DOI: 10.1145/63500.63501. URL: https://doi.org/10.1145/63500.63501 (page 53).

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. “Fault-Scalable Byzantine Fault-Tolerant Services”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles.* SOSP ’05. Brighton, United Kingdom: Association for Computing Machinery, 2005, 59–74. ISBN: 1595930795. DOI: 10.1145/1095810.1095817. URL: https://doi.org/10.1145/1095810.1095817 (page 53).

[3] A. R. Abdou, P. C. Van Oorschot, and T. Wan. “Comparative analysis of control plane security of SDN and conventional networks”. In: *IEEE Communications Surveys and Tutorials* 20.4 (2018), pp. 3542–3559. ISSN: 1553877X. DOI: 10.1109/COMST.2018.2839348 (page 58).

[4] D. Agrawal and A. El Abbadi. “An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion”. In: *ACM Trans. Comput. Syst.* 9.1 (1991), 1–20. ISSN: 0734-2071. DOI: 10.1145/103727.103728. URL: https://doi.org/10.1145/103727.103728 (page 53).

[5] M. K. Aguilera, W. Chen, and S. Toueg. “Failure Detection and Consensus in the Crash-Recovery Model”. In: *Distrib. Comput.* 13.2 (2000), 99–125. ISSN: 0178-2770. DOI: 10.1007/s004460050070. URL: https://doi.org/10.1007/s004460050070 (page 53).

[6] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani. “Securing software defined networks: taxonomy, requirements, and open issues”. In: *IEEE*

Communications Magazine 53.4 (2015), pp. 36–44. DOI: 10.1109/MCOM.2015.7081073 (page 102).

[7] E. Al-Shaer and S. Al-Haj. "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures". In: *ACM Workshop on Assurable and Usable Security Configuration*. SafeConfig '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, 37–44. ISBN: 9781450300933. DOI: 10.1145/1866898.1866905. URL: https://doi.org/10.1145/1866898.1866905 (pages 3, 59, 102).

[8] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. "Network configuration in a box: towards end-to-end verification of network reachability and security". In: *2009 17th IEEE International Conference on Network Protocols*. 2009, pp. 123–132. DOI: 10.1109/ICNP.2009.5339690 (page 59).

[9] T. R. Alves, M. Buratto, F. M. de Souza, and T. V. Rodrigues. "OpenPLC: An open source alternative to automation". In: *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. 2014, pp. 585–589. DOI: 10.1109/GHTC.2014.6970342 (page 112).

[10] R. Anderson. *Reaching Ultra Low Latency in Trading Infrastructure*. https://ffnews.com/thought-leader/reaching-ultra-low-latency-in-trading-infrastructure/. Accessed: 2022-7-18. 2021 (page 4).

[11] N. Anerousis, P. Chemouil, A. A. Lazar, N. Mihai, and S. B. Weinstein. "The Origin and Evolution of Open Programmable Networks and SDN". In: *IEEE Communications Surveys Tutorials* 23.3 (2021), pp. 1956–1971. DOI: 10.1109/COMST.2021.3060582 (page 18).

[12] M. Antikainen, T. Aura, and M. Särelä. "Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch". In: *Secure IT Systems: 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*. Ed. by K. Bernsmed and S. Fischer-Hübner. Cham: Springer International Publishing, 2014, pp. 229–244. ISBN: 978-3-319-11599-3. DOI: 10.1007/978-3-319-11599-3_14. URL: http://dx.doi.org/10.1007/978-3-319-11599-3{_}14 (pages 58, 85).

[13] M. Aslan and A. Matrawy. "Adaptive consistency for distributed SDN controllers". In: *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*. 2016, pp. 150–157. DOI: 10.1109/NETWKS.2016.7751168 (page 151).

[14] A. Atlas, J. M. Halpern, S. Hares, D. Ward, and T. Nadeau. *An Architecture for the Interface to the Routing System*. RFC 7921. June 2016. DOI: 10.17487/RFC7921. URL: https://www.rfc-editor.org/info/rfc7921 (page 29).

[15] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2 (page 36).

[16] F. Bannour, S. Souihi, and A. Mellouk. "Distributed SDN Control: Survey, Taxonomy, and Challenges". In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 333–354. DOI: 10.1109/COMST.2017.2782482 (pages 124, 148, 151).

[17] BBC News. *Colonial hack: How did cyber-attackers shut off pipeline?* https://www.bbc.co.uk/news/technology-57063636. Accessed: 2022-4-7. 2021 (page 108).

[18] BBC News. *Hacker tries to poison water supply of Florida city.* https://www.bbc.co.uk/news/world-us-canada-55989843 . Accessed: 2022-4-7. 2021 (page 108).

[19] K. Benton, L. J. Camp, and C. Small. "OpenFlow Vulnerability Assessment". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: Association for Computing Machinery, 2013, 151–152. ISBN: 9781450321785. DOI: 10.1145/2491185.2491222. URL: https://doi.org/10.1145/2491185.2491222 (page 24).

[20] P. Berde et al. "ONOS: Towards an Open, Distributed SDN OS". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620744. URL: http://doi.acm.org/10.1145/2620728.2620744 (pages 33, 41, 66).

[21] D. J. Bernstein, T. Lange, and P. Schwabe. "The Security Impact of a New Cryptographic Library". In: *Progress in Cryptology – LATINCRYPT 2012*. Ed. by A. Hevia and G. Neven. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 159–176. ISBN: 978-3-642-33481-8 (page 65).

[22] A. Bessani, J. Sousa, and E. E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43 (pages 55, 71, 152, 154, 156, 170, 232).

[23] A. Bessani, M. Santos, J. a. Felix, N. Neves, and M. Correia. "On the Efficiency of Durable State Machine Replication". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, 169–180 (page 55).

[24] *BFT-SMaRt*. https://github.com/bft-smart (pages 156, 170).

[25] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. "An architecture for active networking". In: *International Conference on High Performance Networking*. Springer. 1997, pp. 265–279 (page 18).

[26] E. Biham, S. Bitan, A. Carmel, A. Dankner, U. Malin, and A. Wool. "Rogue7: Rogue Engineering-Station attacks on S7 Simatic PLCs". In: *Black Hat 2019*. 2019 (page 111).

[27] L. Bilge and T. Dumitraş. "Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, 833–844. ISBN: 9781450316514. DOI: `10.1145/2382196.2382284`. URL: `https://doi.org/10.1145/2382196.2382284` (page 3).

[28] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. "Deconstructing paxos". In: *ACM SIGACT News* 34.1 (2003), p. 47. ISSN: 01635700. DOI: `10.1145/637437.637447`. URL: `http://portal.acm.org/citation.cfm?doid=637437.637447` (page 42).

[29] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (2014), 87–95. ISSN: 0146-4833. DOI: `10.1145/2656877.2656890`. URL: `https://doi.org/10.1145/2656877.2656890` (page 18).

[30] F. Botelho, A. Bessani, F. M. Ramos, and P. Ferreira. "On the design of practical fault-tolerant SDN controllers". In: *Proceedings - 2014 3rd European Workshop on Software-Defined Networks, EWSDN 2014* (2014), pp. 73–78. DOI: `10.1109/EWSDN.2014.25` (page 68).

[31] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani. "On the feasibility of a consistent and fault-tolerant data store for SDNs". In: *Proceedings - 2013 2nd European Workshop on Software Defined Networks, EWSDN 2013* (2013), pp. 38–43. DOI: `10.1109/EWSDN.2013.13` (page 68).

[32] F. Botelho, T. A. Ribeiro, P. Ferreira, F. M. V. Ramos, and A. Bessani. "Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane". In: *2016 12th European Dependable Computing Conference (EDCC)*. 2016, pp. 169–180. DOI: 10.1109/EDCC.2016.12 (page 152).

[33] S. Boukria, M. Guerroumi, and I. Romdhani. "BCFR: Blockchain-based Controller Against False Flow Rule Injection in SDN". In: *Proceedings - IEEE Symposium on Computers and Communications* 2019-June (2019). ISSN: 15301346. DOI: 10.1109/ISCC47284.2019.8969780 (pages 59, 60).

[34] G. Bracha and S. Toueg. "Asynchronous consensus and broadcast protocols". In: *Journal of the ACM* 32.4 (1985), pp. 824–840. ISSN: 00045411. DOI: 10.1145/4221.214134. URL: http://portal.acm.org/citation.cfm?doid=4221.214134 (page 43).

[35] C. Braz, A. Seffah, and D. M'Raihi. "Designing a Trade-Off Between Usability and Security: A Metrics Based-Model". In: *Human-Computer Interaction – INTERACT 2007*. Ed. by C. Baranauskas, P. Palanque, J. Abascal, and S. D. J. Barbosa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 114–126. ISBN: 978-3-540-74800-7 (page 8).

[36] Cabinet Office. *Public Summary of Sector Security and Resilience Plans 2017*. 2017 (page 103).

[37] C. Cachin. "Yet Another Visit to Paxos". In: 2010 (page 55).

[38] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-15259-7 (page 41).

[39] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. "Ethane: Taking Control of the Enterprise". In: *SIGCOMM Comput. Commun. Rev.* 37.4 (2007), 1–12. ISSN: 0146-4833. DOI: 10.1145/1282427.1282382. URL: https://doi.org/10.1145/1282427.1282382 (pages 19, 57).

[40] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. Mckeown, and S. Shenker. "SANE: A Protection Architecture for Enterprise Networks". In: *Usenix Security Symposium*. 2006 (pages 19, 57).

[41] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Operating Systems Design and Implementation*. OSDI. 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296824 (pages 7, 49, 51, 71, 123, 238).

[42] M. Castro and B. Liskov. “Proactive Recovery in a Byzantine-Fault-Tolerant System”. In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000 (page 52).

[43] M. Castro, R. Rodrigues, and B. Liskov. “BASE: Using Abstraction to Improve Fault Tolerance”. In: *ACM Trans. Comput. Syst.* 21.3 (2003), 236–269. ISSN: 0734-2071. DOI: 10.1145/859716.859718. URL: https://doi.org/10.1145/859716.859718 (page 52).

[44] V. Cerf and R. Kahn. “A Protocol for Packet Network Intercommunication”. In: *IEEE Transactions on Communications* 22.5 (1974), pp. 637–648. DOI: 10.1109/TCOM.1974.1092259 (page 16).

[45] T. D. Chandra and S. Toueg. “Unreliable failure detectors for reliable distributed systems”. In: *Journal of the ACM* 43.2 (1996), pp. 225–267. ISSN: 00045411. DOI: 10.1145/226643.226647. arXiv: arXiv:1011.1669v3. URL: http://portal.acm.org/citation.cfm?doid=226643.226647 (page 43).

[46] B. Chandrasekaran, B. Tschaen, and T. Benson. “Isolating and tolerating SDN application failures with LegoSDN”. In: *Symposium on Software Defined Networking (SDN) Research, SOSR 2016* (2016). DOI: 10.1145/2890955.2890965. URL: http://dx.doi.org/10.1145/2890955.2890965 (page 62).

[47] S. Chaudhuri. “More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems”. In: *Inf. Comput.* 105.1 (1993), pp. 132–158. ISSN: 0890-5401. DOI: 10.1006/inco.1993.1043. URL: http://dx.doi.org/10.1006/inco.1993.1043 (page 42).

[48] M. Cheminod, L. Durante, L. Seno, F. Valenza, A. Valenzano, and C. Zunino. “Leveraging SDN to improve security in industrial networks”. In: *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS* (2017). DOI: 10.1109/WFCS.2017.7991960 (page 107).

[49] S. Cheung, M. Ammar, and M. Ahamad. “The grid protocol: a high performance scheme for maintaining replicated data”. In: *IEEE Transactions on Knowledge and Data Engineering* 4.6 (1992), pp. 582–592. DOI: 10.1109/69.180609 (page 53).

[50] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. "Attested Append-Only Memory: Making Adversaries Stick to Their Word". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, 189–204. ISBN: 9781595935915. DOI: 10.1145/1294261.1294280. URL: https://doi.org/10.1145/1294261.1294280 (page 54).

[51] Cisco. *Introduction to EIRGP*. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/13669-1.html. Accessed: 2022-2-4 (page 17).

[52] Cisco. *Introduction to IRGP*. https://www.cisco.com/c/en/us/support/docs/ip/interior-gateway-routing-protocol-igrp/26825-5.html . Accessed: 2022-2-4 (page 16).

[53] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988 (pages 193, 253).

[54] M. Conti, F. De Gaspari, and L. V. Mancini. "A Novel Stealthy Attack to Gather SDN Configuration-Information". In: *IEEE Transactions on Emerging Topics in Computing* 8.2 (2020), pp. 328–340. ISSN: 21686750. DOI: 10.1109/TETC.2018.2806977 (page 24).

[55] M. Correia, N. Neves, and P. Verissimo. "How to tolerate half less one Byzantine nodes in practical distributed systems". In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004*. 2004, pp. 174–183. DOI: 10.1109/RELDIS.2004.1353018 (page 54).

[56] V. T. Costa and H. M. K. Costa. "Vulnerability Study of FlowVisor-based Virtualized Network Environments". In: *2nd Workshop Network Virtualization Intelligence Future Internet, Rio de Janeiro, Brazil* 6994 (2013), p. 6994. URL: http://www.gta.ufrj.br/fits, (pages 58, 82).

[57] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, 177–190. ISBN: 1931971471 (page 54).

[58] F. Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Commun. ACM* 34.2 (1991), 56–78. ISSN: 0001-0782. DOI: 10.1145/102792.102801. URL: https://doi.org/10.1145/102792.102801 (page 39).

[59] H. Cui, G. O. Karame, F. Klaedtke, and R. Bifulco. "On the Fingerprinting of Software-Defined Networks". In: *IEEE Transactions on Information Forensics and Security* 11.10 (2016), pp. 2160–2173. ISSN: 15566013. DOI: 10.1109/TIFS.2016.2573756 (page 24).

[60] M. C. Dacier, H. Konig, R. Cwalinski, F. Kargl, and S. DIetrich. "Security Challenges and Opportunities of Software-Defined Networking". In: *IEEE Security and Privacy* 15.2 (2017), pp. 96–100. ISSN: 15584046. DOI: 10.1109/MSP.2017.46 (pages 3, 23).

[61] T. Das, V. Sridharan, and M. Gurusamy. "A Survey on Controller Placement in SDN". In: *IEEE Communications Surveys Tutorials* 22.1 (2020), pp. 472–503. DOI: 10.1109/COMST.2019.2935453 (page 34).

[62] R. De Prisco et al. "On k-set consensus problems in asynchronous systems". In: *IEEE Transactions on Parallel and Distributed Systems* (2001), pp. 7–21. ISSN: 10459219. DOI: 10.1109/71.899936 (page 43).

[63] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, 205–220. ISBN: 9781595935915. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281 (page 152).

[64] S. Deng, X. Gao, Z. Lu, and X. Gao. "Packet injection attack and its defense in software-defined networks". In: *IEEE Transactions on Information Forensics and Security* 13.3 (2018), pp. 695–705. ISSN: 15566013. DOI: 10.1109/TIFS.2017.2765506 (pages 24, 63).

[65] A. Derhab, M. Guerroumi, M. Belaoued, and O. Cheikhrouhou. "BMC-SDN: Blockchain-Based Multicontroller Architecture for Secure Software-Defined Networks". In: *Wireless Communications and Mobile Computing* 2021 (2021). ISSN: 15308677. DOI: 10.1155/2021/9984666 (pages 59, 60).

[66] A. Derhab, M. Guerroumi, A. Gumaei, L. Maglaras, M. A. Ferrag, M. Mukherjee, and F. A. Khan. "Blockchain and Random Subspace Learning-Based IDS for SDN-Enabled Industrial IoT Security". In: *Sensors* 19.14 (2019). DOI: 10.3390/s19143119. URL: https://www.mdpi.com/1424-8220/19/14/3119 (page 107).

[67] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (1959), 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390 (pages 17, 107).

[68] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch, and H. Raymond Strong. "An efficient algorithm for byzantine agreement without authentication". In: *Information and Control* 52.3 (1982), pp. 257–274. ISSN: 00199958. DOI: 10.1016/S0019-9958(82)90776-8. URL: http://www.sciencedirect.com/science/article/pii/S0019995882907768 (page 43).

[69] P. Dong, X. Du, H. Zhang, and T. Xu. "A detection method for a novel DDoS attack against SDN controllers by vast new low-traffic flows". In: *2016 IEEE International Conference on Communications, ICC 2016* (2016). DOI: 10.1109/ICC.2016.7510992 (page 24).

[70] S. Dong, K. Abbas, and R. Jain. "A Survey on Distributed Denial of Service (DDoS) Attacks in SDN and Cloud Computing Environments". In: *IEEE Access* 7 (2019), pp. 80813–80828. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2922196 (page 24).

[71] J. M. Dover. *A denial of service attack against the Open Floodlight SDN controller*. Tech. rep. Dover Networks LLC, 2017. URL: https://silo.tips/download/a-denial-of-service-attack-against-the-open-floodlight-sdn-controller (page 24).

[72] J. M. Dover. *A switch table vulnerability in the Open Floodlight SDN controller*. Tech. rep. Dover Networks LLC, 2017. URL: https://silo.tips/download/a-switch-table-vulnerability-in-the-open-floodlight-sdn-controller (page 24).

[73] Z. Drias, A. Serhrouchni, and O. Vogel. "Analysis of cyber security for industrial control systems". In: *2015 International Conference on Cyber Security of Smart Cities, Industrial Control System and Communications (SSIC)*. 2015, pp. 1–8. DOI: 10.1109/SSIC.2015.7245330 (page 104).

[74] L. Dridi and M. F. Zhani. "SDN-Guard: DoS Attacks Mitigation in SDN Networks". In: *Proceedings - 2016 5th IEEE International Conference on Cloud Networking, CloudNet 2016* (2016), pp. 212–217. DOI: 10.1109/CLOUDNET.2016.9 (page 24).

[75] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona. "The real Byzantine Generals". In: *The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576)*. Vol. 2. 2004, pp. 6.D.4–61. DOI: 10.1109/DASC.2004.1390734 (page 45).

[76] K. ElDefrawy and T. Kaczmarek. "Byzantine Fault Tolerant Software-Defined Networking (SDN) Controllers". In: *Computer Software and Applications Conference (COMPSAC)*. 2016, pp. 208–213 (pages 8, 71, 75, 157, 170).

[77] R. Enns, M. Björklund, A. Bierman, and J. Schönwälder. *Network Configuration Protocol (NETCONF)*. RFC 6241. June 2011. DOI: 10.17487/RFC6241. URL: https://www.rfc-editor.org/info/rfc6241 (page 29).

[78] D. Erickson. "The Beacon Openflow Controller". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: Association for Computing Machinery, 2013, 13–18. ISBN: 9781450321785. DOI: 10.1145/2491185.2491189. URL: https://doi.org/10.1145/2491185.2491189 (page 33).

[79] H. Farhady, H. Lee, and A. Nakao. "Software-Defined Networking: A survey". In: *Computer Networks* 81 (2015), pp. 79–95. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2015.02.014. URL: https://www.sciencedirect.com/science/article/pii/S1389128615000614 (page 20).

[80] N. Feamster, J. Rexford, and E. Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (2014), 87–98. ISSN: 0146-4833. DOI: 10.1145/2602204.2602219. URL: https://doi.org/10.1145/2602204.2602219 (pages 18–20).

[81] D. Ferguson, A. Lindem, and J. Moy. *OSPF for IPv6*. RFC 5340. July 2008. DOI: 10.17487/RFC5340. URL: https://www.rfc-editor.org/info/rfc5340 (page 17).

[82] M. Fitzi and J. A. Garay. "Efficient player-optimal protocols for strong and differential consensus". In: *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. PODC '03. New York, NY, USA: ACM, 2003, pp. 211–220.

ISBN: 1-58113-708-7. DOI: 10.1145/872035.872066. URL: http://doi.acm.org/10.1145/872035.872066 (page 44).

[83] P. Fonseca, R. Bennesby, E. Mota, and A. Passito. "A replication component for resilient OpenFlow-based networking". In: *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012* (2012), pp. 933–939. DOI: 10.1109/NOMS.2012.6212011 (page 68).

[84] 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. 9. ACM PUB27 New York, NY, USA, 2011, pp. 279–291. ISBN: 9781450308656. DOI: 10.1145/2034574.2034812. URL: https://dl.acm.org/doi/abs/10.1145/2034574.2034812 (page 22).

[85] S. Frey, Y. Elkhatib, A. Rashid, K. Follis, J. Vidler, N. Race, and C. Edwards. "It Bends But Would It Break? Topological Analysis of BGP Infrastructures in Europe". In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 2016, pp. 423–438. DOI: 10.1109/EuroSP.2016.39 (page 18).

[86] H. Garcia-Molina and D. Barbara. "How to Assign Votes in a Distributed System". In: *J. ACM* 32.4 (1985), 841–860. ISSN: 0004-5411. DOI: 10.1145/4221.4223. URL: https://doi.org/10.1145/4221.4223 (page 53).

[87] J. Gardiner, A. Eiffert, P. Garraghan, N. Race, S. Nagaraja, and A. Rashid. "Controller-in-the-Middle: Attacks on Software Defined Networks in Industrial Control Systems". In: *Proceedings of the 2th Workshop on CPS&IoT Security and Privacy*. CPSIoTSec '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 63–68. ISBN: 9781450384872. DOI: 10.1145/3462633.3483979. URL: https://doi.org/10.1145/3462633.3483979 (pages 11, 58, 103, 238).

[88] J. Gardiner and S. Nagaraja. "On the Security of Machine Learning in Malware C&C Detection: A Survey". In: *ACM Comput. Surv.* 49.3 (2016). ISSN: 0360-0300. DOI: 10.1145/3003816. URL: https://doi.org/10.1145/3003816 (page 61).

[89] GENI. *Campus Openflow Topology*. https://groups.geni.net/geni/wiki/OpenFlow/CampusTopology. Accessed: 2022-2-13 (page 20).

[90] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. "NOX: Towards an Operating System for Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.3 (2008), pp. 105–110. ISSN: 0146-4833. DOI: 10.1145/1384609.1384625. URL: http://doi.acm.org/10.1145/1384609.1384625 (page 33).

[91] P. Göransson, C. Black, and T. Culver. "Chapter 4 - How SDN Works". In: *Software Defined Networks (Second Edition)*. Ed. by P. Göransson, C. Black, and T. Culver. Second Edition. Boston: Morgan Kaufmann, 2017, pp. 61–88. ISBN: 978-0-12-804555-8. DOI: https://doi.org/10.1016/B978-0-12-804555-8.00004-1. URL: https://www.sciencedirect.com/science/article/pii/B9780128045558000041 (page 27).

[92] A. Hahn. "Operational Technology and Information Technology in Industrial Control Systems". In: *Cyber-security of SCADA and Other Industrial Control Systems*. Ed. by E. J. M. Colbert and A. Kott. Cham: Springer International Publishing, 2016, pp. 51–68. ISBN: 978-3-319-32125-7. DOI: 10.1007/978-3-319-32125-7_4. URL: https://doi.org/10.1007/978-3-319-32125-7_4 (page 109).

[93] L. Han, Z. Li, W. Liu, K. Dai, and W. Qu. "Minimum Control Latency of SDN Controller Placement". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 2175–2180. DOI: 10.1109/TrustCom.2016.0334 (page 35).

[94] B. Heller, R. Sherwood, and N. McKeown. "The Controller Placement Problem". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, 7–12. ISBN: 9781450314770. DOI: 10.1145/2342441.2342444. URL: https://doi.org/10.1145/2342441.2342444 (page 34).

[95] C. L. Hendrick. *Routing Informaion Protocol*. RFC 1058. June 1988. DOI: 10.17487/RFC1058. URL: https://www.rfc-editor.org/info/rfc1058 (page 16).

[96] M. Herlihy. "A Quorum-Consensus Replication Method for Abstract Data Types". In: *ACM Trans. Comput. Syst.* 4.1 (1986), 32–53. ISSN: 0734-2071. DOI: 10.1145/6306.6308. URL: https://doi.org/10.1145/6306.6308 (page 53).

[97] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack. "Uncover Security Design Flaws Using The STRIDE Approach". In: *MSDN Magazine* (2006) (page 58).

[98] S. Hong, L. Xu, H. Wang, and G. Gu. "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures". In: *Network and Distributed System Security (NDSS) Symposium.* NDSS '15. San Diego, CA, USA: Internet Society, 2015. ISBN: 1-891562-38-X. DOI: 10.14722/ndss.2015.23283. URL: http://dx.doi.org/10.14722/ndss.2015.23283 (pages 24, 58, 81).

[99] HP News. *HP Launches Industry's First SDN App Store, Unleashing New Wave of Networking Innovations.* https://www.hp.com/us-en/hp-news/press-release.html?id=1798074#.YcG9RS-l2Rs. Accessed: 2021-12-21. Sept. 25, 2014 (page 23).

[100] Z. Hu, M. Wang, X. Yan, Y. Yin, and Z. Luo. "A comprehensive security architecture for SDN". In: *2015 18th International Conference on Intelligence in Next Generation Networks, ICIN 2015* (2015), pp. 30–37. DOI: 10.1109/ICIN.2015.7073803 (page 85).

[101] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference.* USENIXATC'10. Boston, MA: USENIX Association, 2010, p. 11 (pages 66, 70).

[102] M. Hurfin and M. Raynal. "Simple and fast asynchronous consensus protocol based on a weak failure detector". In: *Distributed Computing* 12.4 (1999), pp. 209–223. ISSN: 01782770. DOI: 10.1007/s004460050067. URL: http://dx.doi.org/10.1007/s004460050067 (page 43).

[103] A. Hussein, I. H. Elhajj, A. Chehab, and A. Kayssi. "SDN security plane: An architecture for resilient security services". In: *Proceedings - 2016 IEEE International Conference on Cloud Engineering Workshops, IC2EW 2016* (2016), pp. 54–59. DOI: 10.1109/IC2EW.2016.15 (page 24).

[104] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. "B4: Experience with a Globally-Deployed Software Defined Wan". In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), 3–14. ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. URL: https://doi.org/10.1145/2534169.2486019 (pages 2, 20).

[105] Y. Jiménez, C. Cervelló-Pastor, and A. J. García. "On the controller placement for designing a distributed SDN control layer". In: *2014 IFIP Networking Conference.* 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857117 (page 34).

[106] H. Jo, J. Nam, and S. Shin. “NOSArmor: Building a Secure Network Operating System”. In: *Security and Communication Networks* 2018 (2018). ISSN: 19390122. DOI: 10.1155/2018/9178425 (page 63).

[107] E. Jonsson and T. Olovsson. “On the Integration of Security and Dependability in Computer Systems”. In: 1992 (pages 35, 37).

[108] R. Kandoi and M. Antikainen. “Denial-of-service attacks in OpenFlow SDN networks”. In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015* (2015), pp. 1322–1326. DOI: 10.1109/INM.2015.7140489 (page 24).

[109] N. Katta, H. Zhang, M. Freedman, and J. Rexford. “Ravana: Controller Fault-tolerance in Software-defined Networking”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR ’15. New York, NY, USA: ACM, 2015, 4:1–4:12. ISBN: 978-1-4503-3451-8. DOI: 10.1145/2774993.2774996. URL: http://doi.acm.org/10.1145/2774993.2774996 (pages 68, 69).

[110] M. B. Kelley. *The Stuxnet Attack On Iran’s Nuclear Plant Was ’Far More Dangerous’ Than Previously Thought*. Ed. by B. Insider. https://www.businessinsider.com/stuxnet-was-far-more-dangerous-than-previous-thought-2013-11?r=US&IR=T . Accessed: 2021-07-18. 2013 (pages 4, 108).

[111] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. “VeriFlow: Verifying Network-Wide Invariants in Real Time”. In: *Workshop on Hot Topics in Software Defined Networks*. HotSDN ’12. Helsinki, Finland: Association for Computing Machinery, 2012, 49–54. ISBN: 9781450314770. DOI: 10.1145/2342441.2342452. URL: https://doi.org/10.1145/2342441.2342452 (pages 3, 59, 102).

[112] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. “VeriFlow: Verifying Network-Wide Invariants in Real Time”. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid (pages 3, 59).

[113] L. Kleinrock. “Information Flow in Large Communication Nets, Ph.D. Thesis Proposal”. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, 1961. URL: http://www.cs.ucla.edu/~lk/LK/Bib/REPORT/PhD/ (page 15).

[114] R. Klöti, V. Kotronis, and P. Smith. "OpenFlow: A security analysis". In: *Proceedings - International Conference on Network Protocols, ICNP* (2013). ISSN: 10921648. DOI: 10.1109/ICNP.2013.6733671 (pages 24, 58).

[115] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. "Onix: A Distributed Control Platform for Large-Scale Production Networks". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, 351–364 (pages 20, 33, 152).

[116] I. Koren and C. M. Krishna. In: *Fault-Tolerant Systems (Second Edition)*. Ed. by I. Koren and C. M. Krishna. Second Edition. San Francisco (CA): Morgan Kaufmann, 2021. ISBN: 978-0-12-818105-8. DOI: https://doi.org/10.1016/B978-0-12-818105-8.00011-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128181058000115 (page 38).

[117] R. Kotla and M. Dahlin. "High throughput Byzantine fault tolerance". In: *International Conference on Dependable Systems and Networks, 2004*. 2004, pp. 575–584. DOI: 10.1109/DSN.2004.1311928 (page 52).

[118] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *ACM Trans. Comput. Syst.* 27.4 (2010). ISSN: 0734-2071. DOI: 10.1145/1658357.1658358. URL: https://doi.org/10.1145/1658357.1658358 (page 52).

[119] D. Kreutz, F. M. Ramos, and P. Verissimo. "Towards Secure and Dependable Software-Defined Networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: Association for Computing Machinery, 2013, 55–60. ISBN: 9781450321785. DOI: 10.1145/2491185.2491199. URL: https://doi.org/10.1145/2491185.2491199 (pages 24, 65, 78, 80).

[120] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. ISSN: 15582256. DOI: 10.1109/JPROC.2014.2371999. arXiv: 1406.0440 (page 20).

[121] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. ISSN: 15582256. DOI: 10.1109/JPROC.2014.2371999. arXiv: 1406.0440 (page 65).

[122] D. Kreutz, J. Yu, P. Esteves-Verissimo, C. Magalhaes, and F. M. Ramos. "The KISS principle in software-defined networking: A framework for secure communications". In: *IEEE Security and Privacy* 16.5 (2018), pp. 60–70. ISSN: 15584046. DOI: 10.1109/MSP.2018.3761717 (page 65).

[123] D. Kreutz, J. Yu, F. M. V. Ramos, and P. Esteves-Verissimo. "ANCHOR: Logically Centralized Security for Software-Defined Networks". In: *ACM Trans. Priv. Secur.* 22.2 (2019). ISSN: 2471-2566. DOI: 10.1145/3301305. URL: https://doi.org/10.1145/3301305 (page 8).

[124] M. Krotofil, K. Kursawe, and D. Gollmann. "Securing Industrial Control Systems". In: *Security and Privacy Trends in the Industrial Internet of Things*. Ed. by C. Alcaraz. Cham: Springer International Publishing, 2019, pp. 3–27. ISBN: 978-3-030-12330-7. DOI: 10.1007/978-3-030-12330-7_1. URL: https://doi.org/10.1007/978-3-030-12330-7_1 (page 104).

[125] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563 (page 49).

[126] L. Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. ISSN: 07342071. DOI: 10.1145/279227.279229. URL: http://portal.acm.org/citation.cfm?doid=279227.279229 (pages 41, 50, 68).

[127] L. Lamport. "Paxos Made Simple". In: *ACM SIGACT News* 32.4 (2001), pp. 51–58. ISSN: 01635700. DOI: 10.1145/568425.568433. URL: http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf (page 42).

[128] L. Lamport. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103. ISSN: 01782770. DOI: 10.1007/s00446-006-0005-x. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=64624 (page 42).

[129] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. ISSN: 01640925. DOI: 10.1145/357172.357176. arXiv: arXiv:1011.1669v3. URL: http://portal.acm.org/citation.cfm?doid=357172.357176 (pages 42, 45, 48, 49).

[130] B. Lantz, B. Heller, and N. McKeown. "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". In: Jan. 2010, p. 19. DOI: 10.1145/1868447.1868466 (pages 10, 71, 88, 180).

[131] J. C. Laprie. "Dependability: Basic Concepts and Terminology". In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese.* Ed. by J. C. Laprie. Vienna: Springer Vienna, 1992, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1. URL: https://doi.org/10.1007/978-3-7091-9170-5_1 (pages 35, 37).

[132] J.-C. Laprie. "Dependable Computing: Concepts, Limits, Challenges". In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing.* FTCS'95. Pasadena, California: IEEE Computer Society, 1995, 42–54. ISBN: 0818671467 (pages 35, 36, 38).

[133] S. Lee et al. "The Smaller, the Shrewder: A Simple Malicious Application Can Kill an Entire SDN Environment". In: *International Workshop on Security in Software Defined Networks & Network Function Virtualization.* SDN-NFV Security '16. ACM, 2016, pp. 23–28. ISBN: 978-1-4503-4078-6. DOI: 10.1145/2876019.2876024. URL: http://doi.acm.org/10.1145/2876019.2876024 (page 86).

[134] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff. "A Brief History of the Internet". In: *SIGCOMM Comput. Commun. Rev.* 39.5 (2009), 22–31. ISSN: 0146-4833. DOI: 10.1145/1629607.1629613. URL: https://doi.org/10.1145/1629607.1629613 (page 15).

[135] 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 - HotSDN '12* (2012). DOI: 10.1145/2342441 (page 65).

[136] H Li, P Li, S Guo, and A Nayak. "Byzantine-Resilient Secure Software-Defined Networks with Multiple Controllers in Cloud". In: *IEEE Transactions on Cloud Computing* 2.4 (2014), pp. 436–447. ISSN: 2168-7161. DOI: 10.1109/TCC.2014.2355227 (pages 8, 70, 75, 123, 147, 154).

[137] H. Li, P. Li, S. Guo, and S. Yu. "Byzantine-resilient secure software-defined networks with multiple controllers". In: *2014 IEEE International Conference on Communications (ICC)*. 2014, pp. 695–700. DOI: 10.1109/ICC.2014.6883400 (pages 8, 70, 75, 147, 154).

[138] J. C. R. Licklider. "Man-Computer Symbiosis". In: *IRE Transactions on Human Factors in Electronics* HFE-1.1 (1960), pp. 4–11. DOI: 10.1109/THFE2.1960.4503259 (page 15).

[139] S. Liu and B. Li. "On scaling software-Defined Networking in wide-area networks". In: *Tsinghua Science and Technology* 20.3 (2015), pp. 221–232. DOI: 10.1109/TST.2015.7128934 (page 20).

[140] B. Lokesh and N. Rajagopalan. "A Blockchain-based security model for SDNs". In: *Proceedings of CONECCT 2020 - 6th IEEE International Conference on Electronics, Computing and Communication Technologies* (2020). DOI: 10.1109/CONECCT50063.2020.9198337 (pages 59, 60).

[141] K. Lougheed and Y. Rekhter. *Border Gateway Protocol (BGP)*. RFC 1105. June 1989. DOI: 10.17487/RFC1105. URL: https://www.rfc-editor.org/info/rfc1105 (page 17).

[142] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann. "JURY: Validating controller actions in software-defined networks". In: *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 109–120. ISBN: 9781467388917. DOI: 10.1109/DSN.2016.19 (page 66).

[143] D. Malkhi and M. Reiter. "Byzantine Quorum Systems". In: *Distrib. Comput.* 11.4 (1998), 203–213. ISSN: 0178-2770. DOI: 10.1007/s004460050050. URL: https://doi.org/10.1007/s004460050050 (page 53).

[144] H. Mann and D. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50 –60. DOI: 10.1214/aoms/1177730491. URL: https://doi.org/10.1214/aoms/1177730491 (page 192).

[145] J. P. Martin and L. Alvisi. "Fast Byzantine consensus". In: *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006), pp. 202–215. ISSN: 15455971. DOI: 10.1109/TDSC.2006.35. URL: http://dx.doi.org/10.1109/TDSC.2006.35 (page 42).

[146] R. Masoudi and A. Ghaffari. "Software defined networks: A survey". In: *Journal of Network and Computer Applications* 67 (2016), pp. 1–25. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.03.016. URL: https://www.sciencedirect.com/science/article/pii/S1084804516300297 (page 20).

[147] S. Matsumoto, S. Hitz, and A. Perrig. "Fleet: Defending SDNs from malicious administrators". In: *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: ACM, 2014, pp. 103–108. ISBN: 9781450329897. DOI: 10.1145/2620728.2620750. URL: http://dx.doi.org/10.1145/2620728.2620750. (pages 73, 81).

[148] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746 (pages 2, 19, 25).

[149] M. McQueen, T. McQueen, W. Boyer, and M. Chaffin. "Empirical Estimates and Observations of 0Day Vulnerabilities". In: *2009 42nd Hawaii International Conference on System Sciences*. 2009, pp. 1–12. DOI: 10.1109/HICSS.2009.186 (page 3).

[150] J. M. McQuillan, I. Richer, and E. C. Rosen. "The New Routing Algorithm for the ARPANET". In: *IEEE Transactions on Communications* 28.5 (1980), pp. 711–719. DOI: 10.1109/TCOM.1980.1094721 (page 17).

[151] J. M. McQuillan, I. Richer, E. C. Rosen, and D. Bert-sekas. "ARPANET Routing Algorithm Improvements: 2nd Semiannual Technical Report". In: *BBN Report* 3940 (1978) (page 17).

[152] J. Medved, R. Varga, A. Tkacik, and K. Gray. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 2014, pp. 1–6. DOI: 10.1109/WoWMoM.2014.6918985 (page 33).

[153] J. van der Merwe, S. Rooney, L. Leslie, and S. Crosby. "The Tempest-a practical framework for network programmability". In: *IEEE Network* 12.3 (1998), pp. 20–28. DOI: 10.1109/65.690958 (page 19).

[154] R. M. Metcalfe and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks". In: *Commun. ACM* 19.7 (1976), 395–404. ISSN: 0001-0782. DOI: 10.1145/360248.360253. URL: https://doi.org/10.1145/360248.360253 (page 16).

[155] N. Meulen. "DigiNotar: Dissecting the First Dutch Digital Disaster". In: *Journal of Strategic Security* 6 (June 2013), pp. 46–58. DOI: 10.5038/1944-0472.6.2.4 (page 65).

[156] O. Michel and E. Keller. "SDN in wide-area networks: A survey". In: *2017 Fourth International Conference on Software Defined Systems (SDS)*. 2017, pp. 37–42. DOI: 10.1109/SDS.2017.7939138 (page 20).

[157] D. L. Mills. *Exterior Gateway Protocol formal specification*. RFC 904. Apr. 1984. DOI: 10.17487/RFC0904. URL: https://www.rfc-editor.org/info/rfc904 (page 17).

[158] P. M. Mohan et al. "Primary-Backup Controller Mapping for Byzantine Fault Tolerance in Software Defined Networks". In: *IEEE Global Communications Conference*. 2017, pp. 1–7 (pages 8, 72, 75, 76, 148, 154).

[159] P. M. Mohan, T. Truong-Huu, and M. Gurusamy. "Byzantine-Resilient Controller Mapping and Remapping in Software Defined Networks". In: *IEEE Transactions on Network Science and Engineering* 7.4 (2020), pp. 2714–2729. DOI: 10.1109/TNSE.2020.2981521 (pages 72, 75, 76, 148, 154).

[160] I. Moise. "Efficient agreement protocols in asynchronous distributed systems". In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 2022–2025. ISBN: 9780769543857. DOI: 10.1109/IPDPS.2011.367 (page 42).

[161] J. Moy. *OSPF specification*. RFC 1131. Oct. 1989. DOI: 10.17487/RFC1131. URL: https://www.rfc-editor.org/info/rfc1131 (page 17).

[162] J. Moy. *OSPF Version 2*. RFC 2328. Apr. 1998. DOI: 10.17487/RFC2328. URL: https://www.rfc-editor.org/info/rfc2328 (page 17).

[163] L. F. Müller, R. R. Oliveira, M. C. Luizelli, L. P. Gaspary, and M. P. Barcellos. "Survivor: An enhanced controller placement strategy for improving SDN survivability". In: *2014 IEEE Global Communications Conference*. 2014, pp. 1909–1915. DOI: 10.1109/GLOCOM.2014.7037087 (page 35).

[164] M. Naor and A. Wool. "The load, capacity and availability of quorum systems". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 214–225. DOI: 10.1109/SFCS.1994.365692 (page 53).

[165] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. "The Cost of the "S" in HTTPS". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, 133–140. ISBN: 9781450332798. DOI: 10.1145/2674005.2674991. URL: https://doi.org/10.1145/2674005.2674991 (pages 8, 65).

[166] G. Neiger. "Distributed Consensus Revisited". In: *Inf. Process. Lett.* 49.4 (1994), pp. 195–201. ISSN: 0020-0190. DOI: 10.1016/0020-0190(94)90011-6. URL: http://dx.doi.org/10.1016/0020-0190(94)90011-6 (page 44).

[167] L. H. Newman. *The Infrastructure Mess Causing Countless Internet Outages*. https://www.wired.com/story/bgp-route-leak-internet-outage/. Accessed: 2022-2-4 (page 18).

[168] T. H. Nguyen and M. Yoo. "Analysis of link discovery service attacks in SDN controller". In: *International Conference on Information Networking* (2017), pp. 259–261. ISSN: 19767684. DOI: 10.1109/ICOIN.2017.7899515 (pages 24, 81).

[169] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks". In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634. DOI: 10.1109/SURV.2014.012214.00180 (page 20).

[170] Open Networking Foundation. *ONF Overview*. Available at www.opennetworking.org/about/onfoverview (page 41).

[171] Open Networking Foundation. *The Openflow Switch Specification 1.1*. Available at https://www.opennetworking.org/. 2011 (pages 25, 40).

[172] Open Networking Foundation. *The Openflow Switch Specification 1.2*. Available at https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf. 2011 (page 40).

[173] Open Networking Foundation. *Software-Defined Networking: The New Norm for Networks*. Tech. rep. ONF, 2012. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf (page 20).

[174] Open Networking Foundation. *The Openflow Switch Specification 1.3*. Available at https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf. 2012 (pages 25–27).

[175] Open Networking Foundation. *The Openflow Switch Specification 1.5.1*. Available at https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. 2015 (pages 25, 40).

[176] OpenVSwitch. http://openvswitch.org/ (page 33).

[177] D. Oran. *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142. Feb. 1990. DOI: 10.17487/RFC1142. URL: https://www.rfc-editor.org/info/rfc1142 (page 17).

[178] B. Palinckx. *EternalBlue: A retrospective on one of the biggest Windows exploits ever*. https://www.loginradius.com/blog/engineering/eternal-blue-retrospective/ . Accessed: 2023-12-15. 2020 (page 3).

[179] M. Paliwal, D. Shrimankar, and O. Tembhurne. “Controllers in SDN: A Review Report”. In: *IEEE Access* 6 (2018), pp. 36256–36270. DOI: 10.1109/ACCESS.2018.2846236 (page 32).

[180] D. Palmer. *Ransomware gangs now have industrial targets in their sights. That raises the stakes for everyone*. Ed. by ZDNet. https://www.zdnet.com/article/ransomware-gangs-now-have-industrial-targets-in-their-sights-that-raises-the-stakes-for-everyone/ . Accessed: 2021-07-18. 2013 (page 108).

[181] M Pease, R Shostak, and L Lamport. “Reaching Agreement in the Presence of Faults”. In: *J. ACM* 27.2 (1980), pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: http://doi.acm.org/10.1145/322186.322188 (pages 42, 46).

[182] B. Pfaff and B. Davie. *The Open vSwitch Database Management Protocol.* RFC 7047. Dec. 2013. DOI: 10.17487/RFC7047. URL: https://www.rfc-editor.org/info/rfc7047 (page 29).

[183] K. Phemius and M. Bouet. "OpenFlow: Why latency does matter". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013).* 2013, pp. 680–683 (page 34).

[184] G. Pickett. "Abusign Software Defined Networks". Blackhat Europe. 2014 (page 24).

[185] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. "A Security Enforcement Kernel for OpenFlow Networks". In: *Hot Topics in Software Defined Networks.* HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, 121–126. ISBN: 9781450314770. DOI: 10.1145/2342441.2342466 (pages 62, 82, 102).

[186] Pox SDN Controller. https://github.com/noxrepo/pox (page 33).

[187] Project Floodlight. *Floodlight.* http://www.projectfloodlight.org/floodlight/ (pages 10, 33, 66, 68, 111, 157).

[188] Project Floodlight. *Floodlight Applications.* https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343528/Applications (page 23).

[189] Project Floodlight. *Loxigen.* https://github.com/floodlight/loxigen (pages 158, 159).

[190] C. Qi, J. Wu, H. Hu, G. Cheng, W. Liu, J. Ai, and C. Yang. "An intensive security architecture with multi-controller for SDN". In: *Proceedings - IEEE INFOCOM* 2016-September (2016), pp. 401–402. ISSN: 0743166X. DOI: 10.1109/INFCOMW.2016.7562109 (pages 73, 75, 154).

[191] A. Rashid, J. Gardiner, B. Green, and B. Craggs. "Everything Is Awesome! Or Is It? Cyber Security Risks in Critical Infrastructure". In: *Critical Information Infrastructures Security: 14th International Conference, CRITIS 2019, Linköping, Sweden, September 23–25, 2019, Revised Selected Papers.* Linköping, Sweden: Springer-Verlag, 2019, 3–17. ISBN: 978-3-030-37669-7. DOI: 10.1007/978-3-030-37670-3_1. URL: https://doi.org/10.1007/978-3-030-37670-3_1 (page 106).

[192] Real Games. *FactoryIO.* https://factoryio.com (page 109).

[193] M. H. Rehmani, A. Davy, B. Jennings, and C. Assi. "Software Defined Networks-Based Smart Grid Communication: A Comprehensive Survey". In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2637–2670. DOI: `10.1109/COMST.2019.2908266` (page 107).

[194] C. Röpke and T. Holz. "SDN rootkits: Subverting network operating systems of software-defined networks". In: *International Workshop on Recent Advances in Intrusion Detection (RAID).* Vol. LNCS 9404. Springer, Cham, 2015, pp. 339–356. ISBN: 9783319263618. DOI: `10.1007/978-3-319-26362-5_16` (pages 24, 81).

[195] C. Röpke and T. Holz. "Preventing Malicious SDN Applications From Hiding Adverse Network Manipulations". In: *Workshop on Security in Softwarized Networks: Prospects and Challenges.* SecSoN '18. Budapest, Hungary: Association for Computing Machinery, 2018, 40–45. ISBN: 9781450359122. DOI: `10.1145/3229616.3229620`. URL: `https://doi.org/10.1145/3229616.3229620` (page 102).

[196] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. "OFLOPS: An Open Framework for OpenFlow Switch Evaluation". In: *Passive and Active Measurement.* Ed. by N. Taft and F. Ricciato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–95. ISBN: 978-3-642-28537-0 (pages 63, 185).

[197] Ryu SDN Controller. `https://ryu-sdn.org` (page 33).

[198] V. Saini, Q. Duan, and V. Paruchuri. "Threat Modeling Using Attack Trees". In: *Journal of Computing Sciences in Colleges* 23 (Apr. 2008) (page 58).

[199] E. Sakic, N. Deriĉ, and W. Kellerer. "MORPH: An Adaptive Framework for Efficient and Byzantine Fault-Tolerant SDN Control Plane". In: *IEEE Journal on Selected Areas in Communications* 36.10 (2018), pp. 2158–2174. DOI: `10.1109/JSAC.2018.2869938` (pages 72, 75).

[200] E. Sakic, F. Sardis, J. W. Guck, and W. Kellerer. "Towards adaptive state consistency in distributed SDN control plane". In: *2017 IEEE International Conference on Communications (ICC).* 2017, pp. 1–7. DOI: `10.1109/ICC.2017.7997164` (page 151).

[201] B. Salisbury. *TCAMs and OpenFlow - What Every SDN Practitioner Must Know.* `https://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/` . Accessed: 16-11-2021. 2012 (page 27).

[202] S. Samonas and D. Coss. "The CIA strikes back: Redefining confidentiality, integrity and availability in security." In: *Journal of Information System Security* 10.3 (2014) (page 37).

[203] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (1990), 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167 (pages 49, 68).

[204] S. Scott-Hayward. "Design and deployment of secure, robust, and resilient SDN controllers". In: *1st IEEE Conference on Network Softwarization (NetSoft)*. Institute of Electrical and Electronics Engineers (IEEE), 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7258233 (page 61).

[205] S. Scott-Hayward, C. Kane, and S. Sezer. "OperationCheckpoint: SDN application control". In: *International Conference on Network Protocols, ICNP*. IEEE Computer Society, 2014, pp. 618–623. ISBN: 9781479962044. DOI: 10.1109/ICNP.2014.98 (pages 24, 64).

[206] S. Scott-Hayward, S. Natarajan, and S. Sezer. "A survey of security in software defined networks". In: *IEEE Communications Surveys and Tutorials* 18.1 (2016), pp. 623–654. ISSN: 1553877X. DOI: 10.1109/COMST.2015.2453114 (pages 3, 58, 78).

[207] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. "SDN security: A survey". In: *SDN4FNS 2013 - 2013 Workshop on Software Defined Networks for Future Networks and Services* (2013). DOI: 10.1109/SDN4FNS.2013.6702553 (pages 3, 57, 78).

[208] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. "SDN security: A survey". In: *SDN4FNS 2013 - 2013 Workshop on Software Defined Networks for Future Networks and Services*. Institute of Electrical and Electronics Engineers (IEEE), 2013, pp. 1–7. ISBN: 9781479927814. DOI: 10.1109/SDN4FNS.2013.6702553 (page 79).

[209] *SDX Central*. https://www.sdxcentral.com/networking/sdn/definitions/cisco-opflex/. Accessed: 2022-2-13 (page 29).

[210] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. "Are we ready for SDN? Implementation challenges for software-defined networks". In: *IEEE Communications Magazine* 51.7 (2013), pp. 36–43. ISSN: 01636804. DOI: 10.1109/MCOM.2013.6553676 (pages 3, 57, 78).

[211] A. Shaghaghi, M. A. Kaafar, R. Buyya, and S. Jha. “Software-Defined Network (SDN) Data Plane Security: Issues, Solutions and Future Directions”. In: *Handbook of Computer Networks and Cyber Security: Principles and Paradigms* (2018), pp. 341–387. DOI: 10.1007/978-3-030-22277-2_14. arXiv: 1804.00262. URL: https://arxiv.org/abs/1804.00262v1 (page 58).

[212] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (1979), 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176 (page 74).

[213] S. S. Shapiro and M. Wilk. “An Analysis of Variance Test for Normality (Complete Samples)”. In: *Biometrika* 52.3/4 (1965), pp. 591–611. ISSN: 00063444. URL: http://www.jstor.org/stable/2333709 (visited on 08/06/2022) (page 192).

[214] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar. “Carving Research Slices out of Your Production Networks with OpenFlow”. In: *SIGCOMM Comput. Commun. Rev.* 40.1 (2010), 129–130. ISSN: 0146-4833. DOI: 10.1145/1672308.1672333. URL: https://doi.org/10.1145/1672308.1672333 (page 19).

[215] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar. *FlowVisor: A Network Virtualization Layer.* Tech. rep. 2009, p. 15. arXiv: arXiv:1011.1669v3. URL: http://openflowswitch.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf http://www.openflow.org/downloads/../openflow-tr-2009-1-flowvisor.pdf%5Cnhttp://www.techrepublic.com/whitepapers/flowvisor-a-network-virtualization-layer/2382721 (pages 19, 58, 82, 157).

[216] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. “Rosemary: A robust, secure, and high-performance network operating system”. In: *Proceedings of the ACM Conference on Computer and Communications Security* (2014), pp. 78–89. ISSN: 15437221. DOI: 10.1145/2660267.2660353 (pages 61, 84, 86, 87).

[217] E. G. d. Silva, L. A. D. Knob, J. A. Wickboldt, L. P. Gaspary, L. Z. Granville, and A. E. Schaeffer-Filho. “Capitalizing on SDN-based SCADA systems: An anti-

eavesdropping case-study". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)* (2015), pp. 165–173 (page 107).

[218] E. G. d. Silva, A. Silva, J. Wickboldt, P. Smith, L. Granville, and A. Schaeffer-Filho. "A One-Class NIDS for SDN-Based SCADA Systems". In: June 2016, pp. 303–312 (page 107).

[219] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher. *OpFlex Control Protocol*. Internet-Draft draft-smith-opflex-00. Work in Progress. Internet Engineering Task Force. 21 pp. URL: `https://datatracker.ietf.org/doc/html/draft-smith-opflex-00` (page 29).

[220] J. Sousa and A. Bessani. "From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation". In: *2012 Ninth European Dependable Computing Conference*. 2012, pp. 37–48. DOI: `10.1109/EDCC.2012.32` (page 55).

[221] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. "Flexible, Wide-Area Storage for Distributed Systems with WheelFS". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, 43–58 (page 66).

[222] G. Sullivan and R. Feinn. "Using Effect Size—or Why the P Value Is Not Enough". In: *Journal of graduate medical education* 4 (Sept. 2012), pp. 279–82. DOI: `10.4300/JGME-D-12-00156.1` (pages 193, 253).

[223] D. Tatang, F. Quinkert, J. Frank, C. Röpke, and T. Holz. "SDN-GUARD: Protecting SDN controllers against SDN rootkits". In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2017* 2017-January (2017), pp. 297–302. DOI: `10.1109/NFV-SDN.2017.8169856` (page 64).

[224] D. Tennenhouse and D. J. Wetherall. "Towards an Active Network Architecture". In: *Computer Communication Review* 26 (1996), pp. 5–18 (page 18).

[225] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. "A survey of active network research". In: *IEEE Communications Magazine* 35.1 (1997), pp. 80–86. DOI: `10.1109/35.568214` (page 18).

[226] P. Thambidurai and Y. keun Park. "Interactive consistency with multiple failure modes". In: *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*. 1988, pp. 93–100. DOI: `10.1109/RELDIS.1988.25784` (page 53).

[227] R. J. Thomas and T. Chothia. "Learning from Vulnerabilities - Categorising, Understanding and Detecting Weaknesses in Industrial Control Systems". In: *Computer Security*. Ed. by S. Katsikas, F. Cuppens, N. Cuppens, C. Lambrinoudakis, C. Kalloniatis, J. Mylopoulos, A. Antón, S. Gritzalis, W. Meng, and S. Furnell. Cham: Springer International Publishing, 2020, pp. 100–116. ISBN: 978-3-030-64330-0 (page 105).

[228] R. J. Thomas, J. Gardiner, T. Chothia, E. Samanis, J. Perrett, and A. Rashid. "Catch Me If You Can: An In-Depth Study of CVE Discovery Time and Inconsistencies for Managing Risks in Critical Infrastructures". In: *Proceedings of the 2020 Joint Workshop on CPS&IoT Security and Privacy*. CPSIOTSEC'20. Virtual Event, USA: Association for Computing Machinery, 2020, 49–60. ISBN: 9781450380874. DOI: 10.1145/3411498.3419970. URL: https://doi.org/10.1145/3411498.3419970 (page 105).

[229] A. Tootoonchian and Y. Ganjali. "HyperFlow: A Distributed Control Plane for OpenFlow". In: *Internet Network Management Conference on Research on Enterprise Networking*. USENIX, 2010, p. 3. URL: http://dl.acm.org/citation.cfm?id=1863133.1863136 (page 66).

[230] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi. "Cross-App Poisoning in Software-Defined Networking". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), p. 16. DOI: 10.1145/3243734. URL: https://doi.org/10.1145/3243734.3243759 (pages 24, 81).

[231] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE Transactions on Computers* 62.1 (2013), pp. 16–30. DOI: 10.1109/TC.2011.221 (page 54).

[232] A. S. Wazan, R. Laborde, F. Barrere, A. Benzekri, and D. W. Chadwick. "PKI Interoperability: Still an Issue? A Solution in the X.509 Realm". In: *Information Assurance and Security Education and Training*. Ed. by R. C. Dodge and L. Futcher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 68–82. ISBN: 978-3-642-39377-8 (page 65).

[233] J. Weekes. "Towards smarter SDN switches: revisiting the balance of intelligence in SDN networks". English. PhD thesis. Lancaster University, Sept. 2019. DOI: 10.17635/lancaster/thesis/727 (page 29).

[234] J. Weekes and S. Nagaraja. “Controlling Your Neighbour’s Bandwidth for Fun and for Profit”. In: *Security Protocols XXV*. Ed. by F. Stajano, J. Anderson, B. Christianson, and V. Matyáš. Cham: Springer International Publishing, 2017, pp. 214–223. ISBN: 978-3-319-71075-4 (pages 24, 29).

[235] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang. “Towards a secure controller platform for OpenFlow applications”. In: *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. 2013, pp. 171–172. ISBN: 9781450320566. DOI: 10.1145/2491185.2491212 (pages 24, 64).

[236] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen. “SDNShield: Reconciliating configurable application permissions for SDN App markets”. In: *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016* (2016), pp. 121–132. DOI: 10.1109/DSN.2016.20 (page 64).

[237] T. J. Williams. “The Purdue Enterprise Reference Architecture”. In: *Comput. Ind.* 24.2–3 (Sept. 1994), 141–158. ISSN: 0166-3615. DOI: 10.1016/0166-3615(94)90017-5. URL: https://doi.org/10.1016/0166-3615(94)90017-5 (pages 104, 105).

[238] L. Yang, T. A. Anderson, R. Gopal, and R. Dantu. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746. Apr. 2004. DOI: 10.17487/RFC3746. URL: https://www.rfc-editor.org/info/rfc3746 (page 19).

[239] S. Yang, L. Cui, Z. Chen, and W. Xiao. “An Efficient Approach to Robust SDN Controller Placement for Security”. In: *IEEE Transactions on Network and Service Management* 17.3 (2020), pp. 1669–1682. DOI: 10.1109/TNSM.2020.2994837 (page 35).

[240] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu. “Software-Defined Wide Area Network (SD-WAN): Architecture, Advances and Opportunities”. In: *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. 2019, pp. 1–9. DOI: 10.1109/ICCCN.2019.8847124 (page 20).

[241] K.-P. Yee. “Aligning Security and Usability”. In: *IEEE Security and Privacy* 2.5 (2004), 48–55. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.64. URL: https://doi.org/10.1109/MSP.2004.64 (page 8).

[242] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. "Separating Agreement from Execution for Byzantine Fault Tolerant Services". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, 253–267. ISBN: 1581137575. DOI: 10.1145/945445.945470. URL: https://doi.org/10.1145/945445.945470 (page 52).

[243] C. Yoon and S. Lee. "Attacking SDN infrastructure: Are we ready for the next gen networking". Blackhat 2016. 2016 (page 24).

[244] P. Zhang, H. Wang, C. Hu, and C. Lin. "On denial of service attacks in software defined networks". In: *IEEE Network* 30.6 (2016), pp. 28–33. ISSN: 08908044. DOI: 10.1109/MNET.2016.1600109NM (page 24).

[245] H. Zhou, C. Wu, C. Yang, P. Wang, Q. Yang, Z. Lu, and Q. Cheng. "SDN-RDCD: A real-time and reliable method for detecting compromised SDN Devices". In: *IEEE/ACM Transactions on Networking* 26.5 (2018), pp. 2048–2061. ISSN: 10636692. DOI: 10.1109/TNET.2018.2859483 (pages 67, 139).

[246] Q. Zhu, C. Rieger, and T. Başar. "A hierarchical security architecture for cyber-physical systems". In: *2011 4th International Symposium on Resilient Control Systems.* 2011, pp. 15–20. DOI: 10.1109/ISRCS.2011.6016081 (pages 104, 105).