Enhancing and Protecting Intrusion Detection Systems Using P4-Enabled Data Planes

Benjamin (Richard David) Lewis, BSc (Hons)

School of Computing and Communications

Lancaster University

A thesis submitted for the degree of

Doctor of Philosophy

October, 2023

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. This thesis does not exceed the maximum permitted word length of 80,000 words including appendices and footnotes, but excluding the bibliography. A rough estimate of the word count is 39000.

Benjamin (Richard David) Lewis

Enhancing and Protecting Intrusion Detection Systems Using P4-Enabled Data Planes

Benjamin (Richard David) Lewis, BSc (Hons).

School of Computing and Communications, Lancaster University

A thesis submitted for the degree of *Doctor of Philosophy.* October, 2023

Abstract

As computer networks have evolved to form the Internet, there has been an ever-growing attack surface, ready to be exploited by malicious actors. Computer networks are fundamental to daily life, with dependence on them further increasing every single day. The Internet is used to facilitate manufacturing, finance, critical infrastructure and global communication. Networks also serve as a fundamental attack surface, exposing users and devices to malicious actors, internally and externally. The cost of weak security can now prove to be enormous, in terms of material costs, as well as outages to service and production.

With the evolution of the uses of computer networks, with networks becoming more pervasive, there has been a need for more flexible and dynamic network management. To this end, the concept of Software-Defined Networking has evolved, taking the historically rigid realm of network management into open specifications and protocols. This paradigm shift from fixed-function to programmable platforms —referred to as *softwarisation*— has enabled innovation in both the management of networks, and how network devices process traffic. Network hardware can be involved not only in forwarding traffic, but also in actively determining how traffic is forwarded.

In this thesis, we explore the intersection of programmable control with programmable hardware. We examine how we can not only leverage existing technologies, but combine them to harness the benefits of distinct approaches. Building on this concept, we present a framework and prototype implementation to facilitate this combination with existing platforms. With the 4MIDable framework, we demonstrate how we can integrate existing network security appliances into emerging network architectures, disseminating their capability deeper into the network. We also show how programmable network infrastructure can be used to protect the network itself.

Publications

Lewis, B. *et al. Using p4 to enable scalable intents in software defined networks* in *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), 442–443

Lewis, B. *et al. P4ID: P4 enhanced intrusion detection* in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2019), 1–4

Lewis, B. *et al.* 4MIDable: Flexible Network Offloading For Security VNFs. *Journal of Network and Systems Management* **31,** 52 (2023)

Acknowledgements

Firstly, I would like to thank my supervisors, Nicholas Race and Matthew Broadbent for their support and guidance throughout my time at Lancaster, I would also like to express my gratitude to Charalampos Rotsos for the many hours endured supporting me.

I would also like to thank colleagues and friends from Lancaster, in no particular order, Ellie Davies, Lyn Hill, Nic Hart, Lyndon Fawcett and many more.

I would like to thank family, especially Marie & Norrie for pushing me this far, and the Dixons and extended family for their unwavering support throughout this long process.

Finally, I would like to extend my thanks to Ethan & family who have worked hard to support me every step of the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The modern world is fundamentally centred around network and Internet connectivity, from individuals on social media, to business and finance, healthcare, manufacturing and even critical infrastructure such as the power grid.

Not only are connected end-devices and networks vulnerable to attack, but also the connecting network infrastructure itself. Disruption can become the focus of an attack, aiming to force hosts or entire networks offline, through a process known as Denial-of-Service (DoS). These attacks have been growing in scale, with the largest recorded in 2021 involving 340 million packets per second of traffic directed towards a Microsoft Azure customer [150].

Given the scale and diversity of these attacks, the cost of poor network security has been estimated to range into billions of dollars, with individual targets experiencing costs in the tens, if not hundreds of millions of dollars. These costs not only account for direct damage to IT systems, but also consequential losses as a result of disruption and are still merely the financial cost. In the case of the WannaCry [95] attack, it is estimated that over 19000 patient appointments were cancelled as a direct result.

These threats and consequences highlight the importance of network security, as the damage was achieved and so prolific by spreading via vulnerable network hosts. To be able to mitigate against network problems, an understanding of the traffic transiting through them is critical. In some cases, this can be a trade-off between real-time visibility and the performance cost of such visibility.

One of the most common and least-intrusive mechanisms of network protection is the firewall. These are usually situated at points between networks as a dedicated middlebox, either between local networks, or between a local network and the Internet. Middleboxes[15] take their name from being placed in the 'network path' whilst not performing normal routing. firewalls, as an example of a middlebox, monitor connections between hosts, either permitting or denying traffic. In some cases, rules will be more specific, defining the behaviours associated with traffic, such as maximum bandwidth. Traffic can be loosely identified, as different types of traffic are usually associated with well-known ports, but malicious traffic will often also use these ports, in an attempt to avoid detection.

Deep Packet Inspection (DPI) is a more resource-intensive approach by which not only the packet headers are inspected, but also the packet payloads. Packets can either be compared against known rules, such as those specifying attack identifiers, or compared against a known baseline, with anomalous traffic outside that baseline being flagged. Like a firewall, an Intrusion Detection System (IDS) must be situated in the traffic flow, or receive a mirrored copy of traffic. However, inspecting the packet payload as well adds computational complexity.

In both cases, the middleboxes can only monitor traffic that passes through them. As a result, they are typically sited as points between networks, such as between a local trusted network, and untrusted wider network. This limits their inspection only to traffic crossing between networks. Local traffic, for example, may be beyond the monitoring scope, and thus, malicious behaviour there cannot be detected. It is also important to note that any action taken by an IDS is reactive, the traffic is already in transit or has been received by the host by the time an alert is generated. The action taken can range from storing alerts for human operators to generating new forwarding rules, to block, limit or re-direct traffic.

To counter this, IDS can act as an intermediary, as an Intrusion Prevention System (IPS), where each packet is inspected before being passed through to the network. However, this adds latency to packet processing and the IPS must be sized appropriately for the largest traffic bursts, to avoid disrupting traffic flow.

Flow monitoring is a lightweight alternative to intrusion detection and instead focuses on monitoring which flows are present in a network, passing this information to a collector. A flow can be defined as a number of fields, including source and destination addresses, flow lifetime and the amount of traffic associated with each flow.

By trading fidelity for significantly reduced resource requirements, both in terms of processing and storage, it can be deployed within network switches themselves, rather than being entirely reliant on middleboxes.

Flow monitoring offers the ability to view network trends and behaviour patterns, such as large consumers of bandwidth, or a sudden large number of new connections. It can offer greater visibility as it can be deployed to a wider range of devices in a network. When scaling to larger networks, this monitoring can still be resource intensive for switches, which have limited on-board compute resources.

In either case, these typically require dedicated appliances or network hardware with monitoring capability. Their deployment and placement must be planned to be able to capture network data most effectively. With a move towards more diverse and divergent network technologies, where the network can be dynamically reconfigured, this can pose challenges for effective placement. These new architectures also provide a new level of capability within network switches and routers, which itself can be used to deploy functionality historically constrained to middleboxes.

1.1 Software Defined Networking for Defence

Software Defined Networking (SDN) is a relatively recent network paradigm, separating the control and data planes of a network. The control plane makes decisions on how traffic should be forwarded, using a wider context of the network, taking into account policies and routing information. With this context, and feedback from the data plane, the control plane generates forwarding rules to be installed into the data plane. OpenFlow [92], first published in 2008, is a popular realisation of the SDN concept, with adoption in both academia and industry. One of the key benefits of Software Defined Networking (SDN) is to de-couple physical and logical network topologies, offering a more flexible and dynamic network architecture. This flexibility can conflict with the static placement of monitoring appliances and applications such as middleboxes.

Whilst the control plane enables flexibility, especially in OpenFlow, the data plane operates at high speeds with more limited behaviour. High-speed devices typically follow a match-action architecture, where the data plane takes an action on a packet based on the packet or series of packets meeting specific criteria. These devices are programmed by the control plane, and provide feedback to the data plane in the form of statistics and even routing entire packets to the control plane where necessary.

By applying SDN to network security a number of benefits can be realised. Software Defined Networking (SDN) enables functionalities traditionally restricted to dedicated appliances to instead be deployed throughout the network into SDN-capable devices, from the core to the edge. This can ease the restrictions in function placement, as well as enabling the network to perform a greater number of security related functions.

1.2 Deficiencies of SDN for Network Defence

Until recently, the focus of SDN was enabling a programmable control plane. One of the mechanisms used to enable deployment in hardware was to focus on the functionality exposed by existing switches, with incremental revisions from that point. A fixed function data plane limits the scope of new behaviour that can be introduced without the control plane being directly involved in the processing. Invoking the control plane in this way has a latency and throughput cost, which grows with traffic volume. This mechanism of invoking the control plane can also be abused in a Denial-of-Service (DoS) attack, where an attacker directs a large volume of traffic to the controller. This can result in either the switch's channel to the controller being overwhelmed, or, in some cases, the controller itself will be unable to process traffic.

Fixed function data planes, such as those offered by OpenFlow, can also suffer from fragmentation between different versions of the specification. Network hardware can therefore constrain capability further in some cases.

1.3 Recent Advances in SDN

More recently, there has been a focus shift towards enabling programmable data planes, not least in the wake of research into programmable switch chips that have performance equal to, or greater than fixed function data planes [10]. Programmable Protocol-Independent Packet Processors (P4) [11], originally published in 2014, proposes domain-specific language and compiler for representing network hardware in a high-level language. Using an open-source, high-level language significantly reduces the barrier to entry for defining network behaviour. P4 has since evolved, with support from a number of hardware [149] and software [103] platforms. The P4Runtime [110] has also been introduced, providing an alternative to the OpenFlow protocol for programming flexible behaviour. P4 enables data plane behaviour to be adapted for network and application requirements, with support for new protocols, metadata collection and stateful processing. The ability to hold state allows the data plane to make decisions autonomously, without the overhead of invoking the control plane.

1.4 Thesis Contributions

This thesis proposes the 4MIDable framework, designed to facilitate the deployment of security applications targeting programmable data planes. We focus on bringing existing middleboxes, appliances and host-based applications into the control plane. This serves to give the controller a greater context and understanding of network behaviour, through the detection and mitigation of network threats.

Our approach is designed around P4 programmable data planes and enabling their capability to be exposed and customised to suit individual applications.

We introduce a dedicated *Control* layer, which is designed to promote interoperability with both existing SDN controllers and bespoke *monitoring agents* that monitor the output of security applications. These agents can process static configurations, as well as dynamic output. The *Management Layer* allows monitoring agents to specify bespoke data plane configurations, which can offer additional monitoring functionality within the data plane itself. This can add application-layer detection, stateful processing and more. Delegating processing into the data plane reduces the work that the controller has to perform, and provides switches themselves with greater autonomy.

With this, we propose that we can harness the small units of computation available for each packet processed through the data plane as a viable alternative and complementary mechanism to relying on the control plane for packet processing and decision-making. We show, in the realisation of our designs, how we can achieve significant reductions in traffic reaching appliances, through the use of data plane offload. We also show this computational capability being used to protect the control plane against attack.

In summary, these are the contributions and aims of this thesis:

1. **Design of a security-focused controller framework for programmable data planes** This framework is built to allow the deployment of monitoring agents, running locally as part of the controller, or remotely as a part of middleboxes or end-hosts. This design will allow for a distributed approach, designed for resilience and scalability, through replicated state between con-

troller instances. The framework is focused on the flexibility of data planes, with the scope for applications to specify data plane behaviour, with a process in the control plane to allow this.

2. **Realisation of a security focused framework for P4** The designed framework is realised in a prototype implementation: 4MIDable, highlighting the benefits and viability of the designed approach, and this framework is evaluated against the goals of the thesis. The evaluation demonstrates that this architecture and implementation of it provides a viable alternative to contemporary monitoring approaches.

3. **Autonomous protection of network control infrastructure** Through the use of 4MIDable, we show that it is possible to use data plane based processing to provide a level of automatic protection against overload and malicious traffic targeting the control plane and its interface with the data plane.

1.5 Thesis Structure

The thesis is divided into six individual chapters. Following this introductory chapter, we move into the evolution of networking and move towards softwarisation of networks. Chapter 2 discusses and motivates the shift towards programmable data planes for implementing network security. This chapter details the evaluation of SDN and how security use-cases have been applied to the paradigm, reviewing contemporary work as well as highlighting where the gaps are in current implementations. We also highlight network threats that target mechanisms used by Software Defined Networks.

Chapter 3 explores the design challenges and motivations of a means to integrate existing security applications into a Software-Defined network, highlighting the benefits of a hybrid approach where we integrate existing appliances. With this, we present a series of design requirements, leading to the 4MIDable framework. The framework design proposes an integration between existing security appliances, programmable data planes and off-the-shelf SDN controllers.

Chapter 4 presents a prototype implementation of the 4MIDable framework, with a controller built to provide the layers of functionality described in the Design. We realise the concept of the Monitoring, Control and Data Plane layers, with APIs to pass data between each one. We then present the motivations and implementation of two use-cases realised as applications for the framework: P4ID targeting Intrusion Detection, and offering a reduction in traffic being processed by an IDS, whilst maintaining similar levels of detection and P4Protect, offering protection to the control plane and other devices from network-based overload attacks.

Chapter 5 presents a detailed evaluation of the P4ID and P4Protect applications that are built using the framework. For each application, we establish baseline figures with the framework, where the application is not in operation. We use this to highlight traffic reductions, attack detection and attack prevention as appropriate. Through this, we show the flexibility that is afforded by the framework, and the layers at which the framework can be integrated with other platforms.

Finally, with Chapter 6, we re-visit the thesis outlines, presenting the main contributions and impact of our work. We also present avenues of future work, taking into account the growing capabilities of emerging platforms.

Chapter 2

Background and Related Work

2.1 Background

This chapter highlights the need for network security and the risks that are ever present, not only to network hosts, but to the network infrastructure itself. We present the most common approaches, discussing their benefits and limitations. We then discuss the impact of emerging network technologies on these approaches, most notably the softwarisation of networks, and how this more topologically diverse landscape not only offers new capabilities and use-cases, but equally, can negatively impact upon existing approaches to network security.

We then introduce a key aspect of this softwarisation, the emergence of Software Defined Networking (SDN). SDN is a paradigm separating the *control* of a network which decides how packets should be processed, from the high-speed *data plane* which forwards the packets. We study two implementations of this concept, OpenFlow, the first commercialised and widespread SDN platform offering a programmable control plane, and then, P4, which targets programmable data planes.

We investigate and compare the state-of-the-art approaches to security applications and monitoring, across traditional, OpenFlow and P4-based landscapes, highlighting the evolution of techniques as networks have become more flexible and capable, whilst also discussing limitations of current research.

2.1.1 The Network Security Landscape

2.1.1.1 The Evolution of Network Security

Networks were initially conceived as a number of independent and incompatible entities. Taking ARPANET [58] as an example, which aimed to connect a number of research institutions together and was established by the US Department of Defence in the 1960s. The goal of ARPANET was to provide a resilient network even in the case of a large scale network outage, not unlike the modern Internet. Though to achieve this, the focus was on openness and flexibility rather than security [85].

With this openness and flexibility, came risks, despite ARPANET being composed of known sites and users. As early as the 1970s, Creeper [138] was developed and released into the network. Creeper simply displayed a message 'I'm a creeper, catch me if you can', moving itself between vulnerable hosts. A subsequent version of Creeper replicated itself before spreading, leading to a counter-program, Reaper, which automatically removed instances of Creeper. Whilst Creeper was benign, it does highlight the concept of a 'worm'. A worm is self replicating code which spreads through vulnerable network hosts.

Throughout the 1980s, networks evolved from a series of individual networks towards a series of interconnected networks, now known as the Internet [130]. This was facilitated by the development and adoption of now common protocols including

Internet Protocol (IP) [123] and Transmission Control Protocol (TCP) [124]. With the greater accessibility to the Internet, the number of cyberattacks also increased dramatically, leading to even more threats online [105]

A number of high-profile breaches arose throughout the 1980s [76][32]. In 1986, the first high-profile and international security incident was discovered [141], in which military computers were being compromised via ARPANET and the data being sold to foreign intelligence agencies. However, this was not discovered by looking for a hacker, but instead it was found after the discovery of a small accounting error. In the end, the hacker attempted to access over 400 systems, using a compromised system in an American University.

In the late 1980s, the first disruptive worm emerged in the form of the Morris worm [107]. The Morris worm was a self replicating worm that spread between hosts, with no aim of directly causing damage or disruption to them. However, the worm spread exponentially across the Internet of the time and is estimated to have infected a tenth of all hosts, causing millions of dollars worth of disruption [32]. The Morris worm is often described as an early example of a Denial-of-Service attack, consuming the compute resources of each infected host.

To counter these new threats, a number of new approaches were fielded. One of the most common, which is still deployed widely, is the network firewall [105] [66]. The network firewall acts as a ‘barrier’ between networks, filtering traffic based on packet headers and a set of pre-defined rules.

Prior to firewalls, networks separation was enforced using gateways, with the work of Mogul an early example of implementation of packet filters [94] [19]. Cheswick [21] describes a means for using a packet filter gateway, to expose services from an internal server to an untrusted external gateway. The paper claims that their approach was even effective against the Morris worm.

As firewalls evolved, they moved from 'packet' to 'circuit filters', otherwise known as stateful firewalls. Stateful firewalls move beyond matching purely on individual rules, and also take the 'state' of a connection into account. That is, whether the connection has been established at the transport layer. In the case of Julkenen et al. this was used to enable outbound connections from within the trusted network to receive responses to a given flow [71]. Application-layer firewalls [13], another approach to exposing services on internal hosts to a wider network, rely on proxying application layer traffic. This, in some cases, allows the firewall to ensure that users are authenticated, protocols are being used correctly and logging of any error conditions [66]. However, it also relies on the firewall having support for each application layer.

In the late 1990s, a number of approaches to real-time detection of network intruders were proposed. Network Flight Recorder [127] offers a platform for collecting network data and analysing it with a bespoke scripting language. Whilst Network Flight Recorder has more flexibility, the work offered by Snort [128] focuses on how monitoring can be used to detect security threats. To achieve this, Snort again uses a bespoke language, for defining the characteristics or *signature* of a malicious flow. Zeek (formerly Bro) [115] on the other hand offers a combination of signature and *anomaly-based* detection, where unusual traffic patterns can also be flagged.

Parallel to the work towards firewalls and intrusion detection, Cisco's Netflow[31], was developed to enable individual switches in a network to generate flow statistics, a flow being defined by the authors as 'a unidirectional sequence of packets with common characteristics'. Flow monitoring offers insight throughout a network, rather than relying on monitoring solely at middleboxes. In order to work around the limitations of switch resources, flow records are sampled and aggregated on the switch before being forwarded onto a collector.

These highlight the importance and focus on securing networks. Contemporary approaches are not dissimilar, using combinations of middleboxes and flow monitoring to enforce network security. Though, despite this ongoing work, the network is under constant and growing threat.

2.1.1.2 Contemporary Network Security

Throughout recent years both the scale and intensity of these threats has rapidly increased. WannaCry [47], one of the most high-profile attacks in recent years had a multi-million pound impact on the United Kingdom's National Health Service, whilst similar attacks, such as NotPetya [52], are suggested to have caused damages in the realms of billions of dollars, impacting upon hospitals, shipping, logistics and even radiation monitoring systems. More recently, these style of attacks have continued, with a number of academic institutions [64] falling victim to ransomware in recent years. Whilst ransomware is only one category of attack, huge impacts can be felt from other attacks, such as Denial-of-Service attacks, the largest of which involved 340 million packets per second of traffic directed towards a Microsoft Azure customer [150].

The target of attacks can be summarised under three core principles [41]: *Confidentiality*, *Integrity* and *Availability*, all of which the network attempts to enforce through network policy and monitoring, whether in a traditional or a *software-defined* network.

Confidentiality is simply the principle that only authorised users have access to the data passing through a network, Integrity ensures that the messages are not being tampered with in flight across the network and Availability ensures that the network can be reached in a timely and uninterrupted fashion.

Network threats attack these principles in a number of ways, with a range of solutions to detect and mitigate against these attacks. Hosts on the network can be vulnerable to malicious traffic, compromising the confidentiality of the network. These hosts could be external attackers who have gained access, or compromised machines within a trusted network. Attacks targeting the confidentiality of a network can be detected through flow monitoring, by searching for suspicious, or unexpected flows, or by performing a process known as Deep Packet Inspection (DPI), whereby the body of each packet is searched against known signatures or anomalous traffic. Firewalls can offer some level of protection against the ingress of attackers, by blocking certain protocols or inbound access.

The integrity of network traffic can be mitigated by implementing checksumming and validation, which is implemented in a number of layers of the network stack.

The availability of a network, or hosts can also be damaged by network threats. Denial-of-Service attacks can be aimed against network infrastructure, attempting to overload a given device, link or controller, or can be targeting specific hosts on the network. It is possible to detect these through the use of firewalls or flow monitoring, and they can then be blocked at the relevant ingress point. Deep Packet Inspection can suffer the same challenges that network controllers or hosts do, at which point they too become overwhelmed by the denial-of-service attack.

2.1.1.3 Firewalls

Firewalls [49] are a core component of modern network infrastructure, often situated at a transition point between networks. For example, between a local-area network and the Internet. Firewalls perform a number of functions, but primarily whitelist or blacklist traffic based on a set of pre-defined rules or thresholds. Traffic is most often categorised using a 5-tuple of packet fields, source and destination addresses, ports and connection state. A rudimentary level of traffic categorisation can be performed with this data, focused on 'well-known' [27] ports, which are associated with a common service, for example, port 80 for HTTP. Modern firewalls can process large volumes of traffic, and in some cases, can attempt to scan or mitigate against network threats. Some even include intrusion detection or intrusion prevention capabilities.

2.1.1.4 Deep Packet Inspection

Deep Packet Inspection (DPI) builds on the inspection that a firewall performs, moving beyond packet headers, with the ability to also process packet payloads. Similarly to firewalls, DPI is typically implemented as a middlebox, situated between networks, monitoring traffic passing in and out. DPI can be implemented either passively, as Intrusion Detection [128] [159] or actively, as Intrusion Prevention. Intrusion Detection typically consumes a mirrored copy of traffic, therefore having minimal impact on traffic forwarding. Intrusion prevention on the other hand, can actively block malicious traffic but has to be within the packet path. This adds latency, and if the IPS starts to fall under heavy load, packet processing can be heavily impacted. [160]

In either case, these systems can be signature [159] or anomaly based [116]. In a signature-based appliance, a known list of traffic patterns is matched against defined signatures. Lists of signatures of recent threats and attacks are published either publicly or as a commercial product. These systems have limited capacity to detect unknown threats. Anomaly-based detection analyses the 'typical' behaviour of a network, flagging up traffic that falls outside these baselines. The challenge with this approach is that there is a learning phase, where the anomaly models are created.

2.1.1.5 Flow Monitoring

Whilst firewalling and deep packet inspection is typically implemented as part of middleboxes, thus being best suited to implementation at network edges, flow monitoring is designed such that it can be deployed throughout a network. To achieve this, flow monitoring, in various guises and iterations, is lightweight enough to be implemented within traditional switches. Flow monitoring can then be used to identify a range of flow characteristics, especially those associated with Denial-of-Service attacks. Flow records are typically exported, usually by switches themselves, and directed towards *collectors*, which aggregate and process the exported records. A number of implementations exist, with NetFlow [25] originating as early as 1996 targeting Cisco switches. Netflow version 5 defines a flow as *'as a unidirectional sequence of packets that all share seven values'*. Netflow has since been superseded by the open standard IPFIX [24], which defines a more flexible structure for flow records, with the caveat that larger flow records are more resource-intensive to collect. sFlow [119] takes a packet instead of flow-based approach to sampling. This allows sFlow to sample traffic that IPFIX and Netflow do not support, though with less control over which traffic generates records.

The approaches above, broadly categorised between flow monitoring and middleboxes, have been designed for traditional networks, and often rely on their placement between networks where the physical topology is known. Emerging networks are undergoing a process of *softwarisation*, where the network is transitioning from a series of interconnections to itself being a programmable entity. This has been achieved by de-coupling the control elements of a network, the *control plane* from the high-speed devices that actually forward packets, the *data plane*. This decoupled architecture has given way to the concept of Software-Defined Networking (SDN). SDN offers the capability of dramatically altering network landscapes, whereby physical and logical networks can be topologically diverse. This can limit the efficacy of traditional approaches, especially those offered by middleboxes, which are typically placed based on an understanding of the underlying physical network.

Whilst potentially challenging the implementation of traditional security techniques, SDN also offers a breadth of new opportunities for implementing both existing and new capabilities. With a greater level of flexibility, combined with a global view of the network, SDN, and more recently, programmable data planes offer significant opportunities for enhancing the security of computer networks.

2.1.2 Software Defined Networking

Whilst traditional networks can be considered as static, passing packets to their destination based on pre-configured rules, newer approaches allow a significantly increased level of flexibility for performing not only typical network behaviour, but also for adding monitoring and mitigation.

Programmable networks have been a focus of networking research for a number of years, with early work focusing on ActiveNetworks[125], whereby packets would include a means to modify the behaviour of network devices programming them. In the case of Smart packets [131], this was achieved by including executable code with each packet being sent, though Psounis et al. describe this as being "extremely dangerous".

Whilst this work never reached commercialisation, the research area continued to evolve, with 4D [51] motivating a clean-slate approach, whereby control is separated from the data plane, simplifying the data plane, whilst equally enabling a richer control plane. SANE [16] and Ethane [17] built on this concept, being built around the notion of a centralised control plane, implementing network-wide policy.

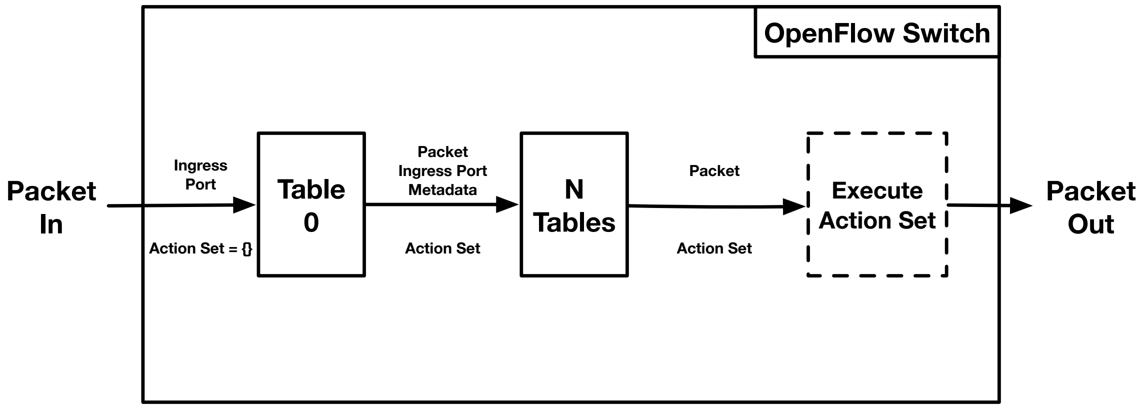


Figure 2.1: OpenFlow Pipeline

The work towards forging the distinction between *control* and *data* planes eventually coined the term *Software-Defined Networking* and subsequently, OpenFlow. OpenFlow [92] describes a protocol and design, initially aimed at commodity Ethernet switches, to provide a centrally programmable flow table with a control-plane protocol for managing table entries and gathering statistics. OpenFlow was initially focused on ease of implementation for existing vendor hardware, supporting a limited number of header fields and a single match-action table. With OpenFlow, vendors do not have to expose the inner workings of their hardware.

As OpenFlow has evolved [22], the pipeline has grown significantly with support for a wider range of fields, multiple tables, meters and counters for gathering statistics and controlling forwarding behaviour.

OpenFlow has seen adoption in academia, vendor hardware and industry-based deployments, ranging from university networks to Google's own WAN, B4 [67]. Numerous works building upon OpenFlow, encompassing network flexibility, visibility, enhanced security and ability to adapt to a range of topologies have been published and are discussed in this chapter.

OpenFlow centres around the concept of a software-based *controller*, typically managing a given network from a single point. This global view allows the controller to make decisions in conjunction with contextual data from across the network. Controllers are written in a variety of general-purpose programming languages [5] [73] [151] and often offer extensibility to add bespoke functionality, implementing discovery, routing, forwarding and monitoring. Controllers generate flow rules and flow information requests which the switches then respond to.

To enable more advanced functionality in OpenFlow, the controller is often invoked to process packets itself, with the packet either mirrored or redirected towards the controller [132]. In either case, a number of platforms have limited bandwidth for this link between controller and switch [28], which can especially impact upon packet forwarding where the controller is required to forward packets on to their destination. These latency and bandwidth limitations can limit the suitability of OpenFlow for large scale monitoring. To counter this, some works have proposed extensions to OpenFlow switches in order to better enable security and monitoring capability within the data plane [28] [120].

2.1.3 Towards Data Plane Programmability

Despite the success of OpenFlow, it has always targeted inflexible data planes. In the initial version [92], data plane complexity was intentionally restricted to a subset of common features identified across a number of vendors' hardware. Not only is OpenFlow 1.0 limited to 12 packet header fields, it also only supports a single table. Subsequent versions of OpenFlow have added support for additional packet header fields and multiple tables, but these changes require hardware upgrades to merchant silicon, or modification to software switches.

However, platforms have emerged that support the re-programmability of switching hardware in the field [10][109]. This field of research has led to the creation of P4, an open-source, extensible language and specification for programming and managing programmable network devices. P4 [11] offers "a high-level language for programming protocol-independent packet processors", or in other words, a cross-platform language and compiler that can define the behaviour of network devices.

The approach offered by P4 is to move from a bottom-up architecture, where the data plane's capability dictates the capability and flexibility of the network as a whole, to a top-down approach, where the data plane's behaviour is instead described by the programmer as a high-level language [91]. This enables applications to be platform-independent. These devices are then managed using an open-source interface known as the P4Runtime [110]. The P4Runtime allows for runtime control of P4 built-in objects, as well as target-specific P4 constructs, field re-configuration of devices and sending packets between devices and controller(s). However, it does not target platform-specific aspects, such as port and traffic management. When a P4 data plane application is compiled, the relevant P4Runtime artefacts are generated for the control plane. These specify message formats, data-types and identifiers. The P4Runtime contrasts with OpenFlow in that it supports a flexible range of fields and messages, based on the current data plane application, rather than relying on fixed message structures.

Given the flexible structure of P4's data plane, it is possible to implement an OpenFlow compatible data plane in P4 and this has been shown as part of the switch.p4 project [44] in combination with [45], though this implementation has fallen to an unmaintained state since 2016. This implementation reinforces the differences between OpenFlow and P4. OpenFlow was designed as a compromise based on what switches at the time were capable of, whereas P4 was built to enable programmability in flexible switching hardware.

Whilst P4 is a programmable and flexible language, it should be noted that there are some constraints, which are commensurate with the capabilities of programmable switching hardware [10]. These constraints are primarily in how resources are allocated, and the flow of application logic. P4 does not support dynamic memory allocation, the size and structure of variables is defined at compile-time. Loops are not supported either, however, bounded or finite loops can be described as a series of steps, facilitating compatibility with a pipelined device architecture. P4 also lacks support for floating point numbers, but they are generally not needed for packet processing [42].

P4 has undergone a series of revisions, initially published as P4$_{14}$. This first version was focused on a switch architecture, and had many primitives as part of the core language. P4$_{16}$ is the most recent version of the language and makes a number of semantic and syntactical changes to the language, which break backwards compatibility. These changes include eliminating language features, moving them into libraries. P4$_{16}$ provides a 'stable language definition', aiming to maintain future compatibility. We discuss the differences between the language revisions further in Section 2.1.3.2.

2.1.3.1 Portable Switch Architecture

In order to better understand the capabilities of P4$_{16}$, we introduce it in the context of the Portable Switch Architecture (PSA) [53]. PSA is self-described as analogous to the standard library in the C programming language. The PSA specification describes a number of types and language constructs representing the behaviour of a typical network switch and is an evolution based on previous models and language versions [11][103].

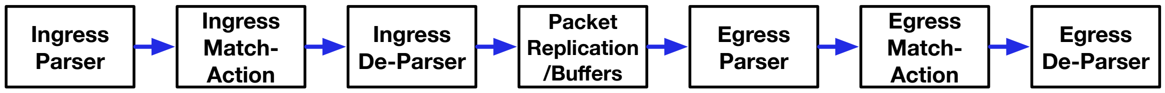


Figure 2.2: P4 Portable Switch Architecture Diagram

PSA is composed of programmable P4 'blocks' or stages, with a platform-specific stage for packet buffering and replication, which enables vendors to implement their own mechanisms in fixed function hardware. These stages are shown in Figure 2.2.

The first stage of the programmable blocks is the *parser*. The parser describes how a bitstream representing a packet should be divided up into headers and their fields. The programmable parser allows for new headers to be supported by an already deployed switch, as well as deploying bespoke protocols within hardware itself.

```
header ethernet_t {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> etherType;
}
```

Figure 2.3: Example of a P4 Header Definition

```
state start {
    transition parse_ethernet;
}
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        0x800: accept;
        default: reject;
    }
}
```

Figure 2.4: Example of a P4 Parser

Figure 2.3 demonstrates the definition of an Ethernet header [122]. This header is composed of three fields: Source and Destination addresses, each 48 bits in length, and the ‘type’ identifier, which is 16 bits in length and identifies the subsequent protocol.

Figure 2.4 shows a parser definition for the Ethernet header structure shown in Figure 2.3. The parser is a finite state machine, ending either in an accept or reject state, allowing a packet to be dropped if a protocol is not recognised or a parsing error occurs. For each stage in the parser, a packet header is extracted and the parser transitions to the next state. The transition is typically based on a type field from the most recently extracted header.

In the example shown, the parser first transitions to parsing the Ethernet header as described. At the end of this state, the parser checks the Ethernet type field that has just been parsed. If it is 0x800, representing an IPv4 header, that header is parsed, else the parser rejects and drops the incoming packet.

Parsers can parse multiple headers of the same type, with primitives in the parser supporting stacks of headers, and looping through states to populate the stack. Whilst P4 doesn't support unbounded loops, the parser is bounded by the length of the packet header [111]. Parsers can also parse data into *metadata* fields, which can then be used as part of table matches. Metadata fields can either be user-defined, and populated in the parser and match action stages, or they can be part of the *platform metadata*, which are processed automatically by the platform. Metadata fields can be used to parse packet data into the same fields from different headers, for example, TCP and UDP port numbers can share a metadata field, as they are mutually exclusive. This is useful for using a single table to be able to match on transport layer ports regardless of which protocol is in use, reducing the amount of table memory required.

Parsed packets then move to the ingress block. The ingress block is composed of one or more *match-action stages*. These match-action stages define one or more match-action *tables*. Each table defines one or more keys; header or metadata fields that are associated with a given table entry.

Tables support a number of match types, in PSA, they are as follows:

- Exact - The key must match exactly

- Ternary - A key and mask of equal length are supplied. The mask specifies which bits in the value are matched on. A priority for ternary entries is also supplied, for when multiple keys match.
- Longest Prefix Match (LPM) - Key is supplied with the number of prefix bits to match on. Priority is implicit from the longest prefix, hence the name.
- Range Match - Represents min..max intervals

These entries are designed such that they can be implemented in different types of memory depending on the hardware implementation [46]. Typically, exact match tables are implemented in hash-memory and tables containing range, LPM or ternary matches are implemented in Ternary Content Addressable Memory (TCAM). It should be noted that in either case, in hardware switches, physical memory can be a limiting factor and flexible table structures can alleviate this. When compiling these tables, the compiler may analyse table order dependencies, sharing stages between tables that can be parallelised where appropriate for the platform [10].

Each entry is associated with an *action*, a function called when a given entry is matched. Actions can modify packet headers or metadata, including adding or removing headers by setting their valid identifier. Actions can also call a number of P4 constructs:

- Counters - These are indexed arrays that can be accessed by the control plane. P4 actions can increment these, either as a one-to-one relationship with a table's entries, or an index calculated within an action itself.
- Meters - These are used to keep statistics about packets or traffic rates, giving three categories of current state, either green, yellow, or red.

- Registers - These are stateful memory that can be read and written to from P4. This allows for stateful packet processing entirely within the data plane.
- Digests - These are used to send notifications to the data plane. These are commonly used for implementing a learning switch. Digest messages are filtered such that duplicate messages are not sent to the controller.

These features are in addition to being able to perform basic calculations over packet fields, and calling a number of platform specific externs, including checksum calculation units.

Figure 2.5 gives an example of a P4 table, matching on the header shown in Figure 2.3, and a PSA platform metadata field. Each P4 platform has a number of fields that are available to the match-action pipeline and include the ingress port and ingress timestamp of a packet, as well as queueing metadata and whether a packet has been cloned or recirculated.

The table shown has 3 associated actions, *forward* which sets the egress port for a given packet, *NoAction* which does nothing and *drop* which sets a bit for the queueing engine to drop the packet.

Taking *forward* as the example action, Figure 2.6 shows how actions can accept parameters to set either packet headers or metadata fields. In this case, forward takes two parameters, updating a standard metadata field for the egress port and updating the source MAC address of the packet. These parameters can also be used in conjunction with externs, for example, to update a register field.

Tables can be made dependant, such that the result of one table being applied, and whether an entry was hit can be used to determine if subsequent tables are applied.

This, however, can have an impact on the number of match-action stages a given P4 program may require [10].

Once a packet has passed through the match-action stages, it is processed by the de-parser, which places valid packet headers in a programmer-defined order. In the case of the de-parser, a packet header is deemed valid if it has previously been parsed, or has been set as valid in the match-action pipeline.

Packets then pass through the buffering/replication engine. The P4 behavioural model [103] and PSA implements multicast, cloning and packet recirculation in this stage. This is implemented outside the P4-programmable stages.

Packets, including any cloned or multicasted copies, are then passed to the egress parser, match-action stages and de-parser. Their operation is almost identical to the ingress stage, barring one key limitation. To facilitate multi-pipeline design, packets in the egress stages cannot be redirected, their destination has been set by the buffering/replication stage, which allows packets to be moved between pipelines in the relevant switch designs. Packets finally are passed to the egress buffers, which are again platform specific.

2.1.3.2 P4$_{14}$ and P4$_{16}$

P4$_{14}$ was the first published version of the P4 language, offering the *abstract switch* architecture [11]. The abstract and PSA architecture are fundamentally similar, sharing the overall flow of parsing, match-action tables and de-parsing.

However, as the first iteration of the language, P4$_{14}$ did present some drawbacks. The language included a number of constructs specific to certain architectures, leaving undefined constructs when targeting others. The language also made certain elements

```
table l2DestinationMatch {
  keys = {
    psa_ingress_input_metadata.ingress_port: exact,
    hdr.ethernet.srcAddr: ternary,
    hdr.ethernet.dstAddr: lpm,
  }
  actions = {
    forward
    NoAction
    drop
  }
  size = 1024;
  default_action: drop;
}
```

Figure 2.5: Example of a P4 Table

```
action forward(bit<48> srcAddr, bit<9> port) {
  hdr.ethernet.srcAddr = srcAddr;
  psa_ingress_output_metadata.egress_port = port;
}
```

Figure 2.6: Example of a P4 Action

implicit, such as the de-parser being generated based on the parser. This resulted in additional parser complexity to generate the relevant de-parser.

P4$_{16}$ [14] introduces a number of changes to the language, whilst retaining a level of backwards compatibility with P4$_{14}$. These changes include moving a number of language constructs to target-specific libraries, known as *externs*. This reduced the number of keywords in the language from over 70, down to 40.

Externs allow target specific functions that cannot be controlled by P4, for example, registers, to be accessed through APIs. In software switches, it is possible to add bespoke externs to further enhance capability.

P4$_{16}$ also allows flexibility in the structure of pipelines, rather than itself defining their structure. This has led to a number of architectures, including the Very Simple Switch, which models a switch with a single table, the *v1model*, which offers the same architecture as that of the P4$_{14}$. To continue providing a reference architecture, the *PSA* model has been developed, with a number of frequently used types, externs for counters, meters and registers and 'packet paths' to manage the flow of packets. The aim of this is to allow applications to be written that are portable across any platform implementing the PSA model.

2.1.3.3 P4 Summary

P4 offers a number of benefits, either in conjunction with or as a more capable alternative to OpenFlow-based networks. Firstly, programmable targets offer a level of homogeneity across the network, that may not have been equal in OpenFlow, owing to differences in implementations between switches [129] [3]. This can benefit from both the perspective of expected behaviour, and deploying monitoring more widely within the network. This capability is further enhanced with the addition of other P4 capable devices, such as Network Interface Cards (NICs). Programmable NICs [63] can enable this monitoring functionality to be pushed to the hypervisor, or even host level.

Secondly, P4 can be used to implement behaviours that are both more flexible, and allow devices to be more autonomous within the data plane. This is further enhanced by the ability to store local metadata entirely within the data plane, which can be then read from or written to registers. The ability to hold state within the data plane itself allows switches significantly greater autonomy.

Thirdly, a recurring issue present in OpenFlow comes from the use of resources. Resources in this case referring to table memory: hash tables and TCAM. Whilst a number of optimisation approaches exist for OpenFlow, this memory is still allocated in a fixed fashion to OpenFlow tables. P4 on the other hand, allows the data plane to be reconfigured such that unused resources can be re-allocated as needed. This can offer much greater capability for additional monitoring or forwarding rules in the data plane.

2.2 Related Work

2.2.1 Security in SDN

Network flexibility and programmability have long been studied in the context of enhancing network security. SANE [16] and Ethane [17], precursors to OpenFlow, were driven by a need for managing access control in large-scale networks. To achieve this, they demonstrated the separation of the control and data planes, a core component of SDN.

Implementing monitoring and mitigation in Software Defined Networks is not without challenges. These challenges include managing resources available for monitoring and forwarding, as well as where these functions should be placed in the network.

In this section, we explore the solutions that have been developed using both OpenFlow and P4 based architectures. Through these, we can highlight the limitations in relying on either control or data plane based programmability, and that the best trade-off may lie in a combination of the two.

We start by examining flow monitoring, a use-case designed to be lightweight and pervasive, and contrast this to how the roles traditionally requiring middleboxes are deployed into Software Defined Networks. We then focus on how these capabilities and functionalities have been exposed through a range of security frameworks.

2.2.1.1 Flow Monitoring

Flow monitoring in SDN can benefit from the global view of the network. A centralised controller has the context available to it to place monitoring tasks more effectively along the network path, and there is flexibility in what defines a flow to be sampled.

However, with this flexibility comes some challenges. Firstly, the challenge of efficient placement of monitoring functions. Switch resources are limited and there is a balance to be struck between adding monitoring functionality and interrupting forwarding capability.

Secondly, there are costs associated with gathering statistics. Flow setup and expiration, where the controller is alerted after a period of time, has a processing overhead [164]. Polling for traffic statistics from switches can also be expensive [142], especially if they have many entries, again limiting fidelity and requiring additional load on the controller.

Flow sampling is one technique used to tackle the resource challenge. Flow sampling is a process where only a subset of network traffic or network traffic headers are inspected, to reduce the overhead of inspecting traffic.

Suarez-Varela et al. [143] [144] target a Netflow/IPFix compatible solution for OpenFlow, relying on several sampling methods to gather traffic, with a table entry added to the switch for each flow. Flows are reported to the controller when that entry expires. The work shows that a hash-based flow selection method is the most effective for identifying the greatest number of flows, and the author posits that other approaches can favour certain types of flow, such as heavy hitters [96], flows which consume a large proportion of available link or path bandwidth. Flexam [137] proposes a means for either random or deterministic packet sampling using OpenFlow primitives to send packets, or parts of packets to the controller, and argues that it is better suited to short-lived flows than other sampling approaches such as sFlow. However, Flexam does rely on an extension to the OpenFlow specification in order to implement their approach. OpenSample instead uses sFlow [119] sampling capability already present in some commodity hardware, combining it with OpenFlow and a bespoke controller implementation. OpenSample makes estimates about traffic statistics based on packet metrics and calculates averages based on the sample rate.

The important thing to note here is that no modification is required to switches to implement.

Another technique used to tackle the resource issue is to use intrinsic messages as a source of data. FlowSense [164] demonstrates the use of existing OpenFlow control traffic to identify flows and gather their statistics. However, this approach is vulnerable to rule wildcarding, where rules are combined to reduce the number of flow entries, and it can only provide data when flows expire. Therefore, it is not suited to real-time visibility of network behaviour.

In PayLess [23], the authors claim a significant improvement in statistical accuracy compared to FlowSense, by polling at a varying interval. The authors claim that there is up to a 50% reduction in messaging overhead compared to periodic polling, and argue that the zero-cost approach of FlowSense comes with significantly reduced accuracy.

Oh et al. [100] propose an 'active' flow monitoring mechanism, where they base monitoring on flows that are in use by active flow rules. Their aim in this is to reduce the monitoring overhead by only targeting ports that are in use. Compared to other solutions, the authors describe how they use the overall understanding of the network topology to minimise requesting duplicate statistics. As the traffic volume on a given port increases, the authors' mechanism reduces the monitoring interval. This approach would be most effective on partially loaded switches, but for switches operating at capacity, the benefits could be more limited.

Relying on flow setup can negatively impact performance, as in production networks, wildcarding of forwarding rules [28] is often used to both reduce table occupancy and minimise the amount of controller to switch communication.

The placement of flow monitoring should also be considered. DIFANE [167] uses an approach whereby packets are forwarded to a subset of *intermediate* switches, which are configured for monitoring. The aim of this is to reduce control plane load and complexity, whilst dividing the roles of monitoring and forwarding between different switches. FlowCover [142] takes a similar approach, by aggregating flow monitoring into a smaller number of switches, whilst also trying to optimise polling rates.

Living on the edge [89] also explores placement and opts for placing monitoring functions on ingress and egress switches, but their implementation demonstrates that these switches can easily become overloaded.

OpenNetMon [153] instead focuses on monitoring edge switches, with their methodology following the principle that core switches will be under significantly greater load with a larger number of flows, and that polling these switches too regularly could have a negative impact on performance. This approach for edge switches is combined with a regular sending of probe packets, which are used to estimate the path latency for each flow.

The rate at which switches are polled can also influence the statistics, and the load on switches.

In their implementation, the authors of SDN Mon [120] separate monitoring and forwarding, however, this is achieved using a programmable software switch [126], which both limits the scope of implementation on other devices, and highlights a benefit from using more flexible platforms for monitoring. In Devoflow [28], the case for more data plane autonomy is also made. The paper demonstrates an approach whereby the data plane is allowed to operate semi-autonomously, by the extensive wildcarding of forwarding and monitoring rules. The authors argue that for certain use-cases, flows do not need to be detected until they reach a certain size.

With the introduction of P4, and the rise in popularity of programmable data planes, novel techniques also become possible. One of the key drivers behind this is P4's support for stateful packet processing entirely within the data plane. This is used in a range of ways, from collecting traffic metrics, to detection of certain classes of attack. In-Band Network Telemetry (INT) [154] [99] is a technique for increasing network visibility through the use of programmable data planes. It allows forwarding devices in a network to insert additional headers into each packet, containing metadata available from the packet pipeline. Figure 2.7 shows an example of an INT header being inserted between the IP and transport layers of a packet, which is the approach taken by the Open Network Operating System (ONOS) [5].

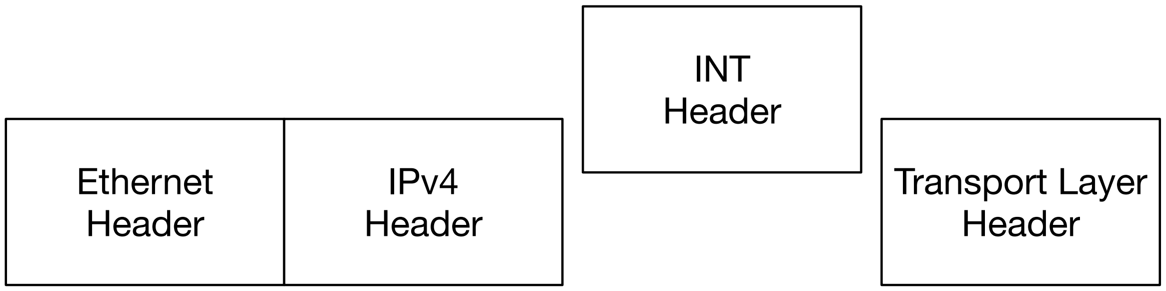


Figure 2.7: P4 In-band Network Telemetry Example

In the ONOS implementation, data available includes ingress/egress ports, queue occupancy, latency, port utilisation as well as other metrics that can be retrieved from the device, or from state registers managed by the device. As traffic progresses through the network, these headers can be removed and sent, as new packets to collectors in the network. This permits a push-based approach, negating the impact or delays of polling switches for data. However, the collection of such a high level of data can also pose challenges. For example, with ONOS and IntMon [154], the performance implications of receiving this telemetry data at the controller can reduce throughput to 0.1Kpps (kilo-packets per second). The author of the paper describes the costs in two areas, namely CPU utilisation and control plane bandwidth. The authors of INTCollector [155] claim that they can improve upon this approach by filtering the events that need to be stored. To achieve this, in collectors, they use high-speed kernel-based filters to identify events which should be escalated to the 'slow-path' for further processing and storage. To further increase performance, the focus of INTCollector is not strictly on raw data, but instead, identifying changes in data points. Vestin [156] also relies on the filtering of events before they are processed, using per-packet or per flow thresholds pre-filtering the traffic within the data plane itself.

Whilst INT in the above examples, provides data from the internal packet processing pipelines of a switch, it only gives instantaneous data. P4 platforms can go beyond this, collecting data in stateful memory, which is then later retrieved by the control plane.

FlowStalker [18] again demonstrates P4's capability for greater autonomy. In FlowStalker, flows are counted in the data plane, and when a configured threshold is reached, a second stage of monitoring is triggered. To collect this data, the authors propose the use of In-Band Network Telemetry and a *crawler packet*. The *crawler packet* is sent to the switch, and headers are added to the packet with the contents of metadata field.

The *crawler packet* is sent from the controller when the controller receives a 'threshold warning packet', which is triggered from the data plane. FlowStalker aims to offer a clustered approach for monitoring, aiming to reduce the impact on the data plane. The author finds that throughput drops by approximately 12% on the P4 behavioural model, but there is minimal overhead in the control plane.

ShadowFS [113] builds on FlowStalker to further improve monitoring performance. To do so, the author introduces a cache mechanism for the most frequently accessed entries. This targets software rather than hardware platforms, where table access speeds can vary based on a number of factors, including the number of table entries. ShadowFS aims to abstract this implementation complexity away from the programmer.

With programmable parsing, P4 platforms can also inspect a greater range of packet headers than is possible for OpenFlow. This greater level of capability starts to overlap with features that would traditionally require a dedicated middlebox or appliance. As an example, Narayanan et al. [97] implement the parsing of ARP and DHCP requests, with a view towards mitigating against DHCP spoofing and starvation. This is achieved through the use of port-based access control, combined with a whitelist of expected IP addresses for a given port. This work is also demonstrated on the NetFPGA [63] hardware platform.

DroPPPP [74] combines the use of spoof detection and three-colour meters [60] to mitigate against Denial-of-Service attacks in P4. The authors do not specify on which platform their implementation was evaluated. The anti-spoofing measures are implemented using a stored hash of MAC and IP address, if these do not match with subsequent packets, this is flagged up as an attack. When an attack has been flagged, the new address can only be bound after 5 seconds since the last flagged attack. The authors use the three-colour meters to detect packet rates of hosts connected to the switches, as abnormally high rates can be associated with DoS attacks. Their approach reduces the impact of DoS attacks on packet loss and reduces the CPU load on the controller at the same time.

Sketches are a data structure that can approximate or summarise data, for example, to estimate frequency, or determine whether a value is a member of a particular set. This is achieved with bounded resources, so can be suitable for deployment into hardware switches. Sketches can be applied to estimating traffic rates, or determining whether a particular digest or notification has already been sent.

Count-Min sketches, which are used in a number of works presented in the upcoming section, aim to estimate the frequency of a given event, with a guarantee that it will never undercount. Count-Min sketches may overestimate frequency due to hash collisions.

Bloom filters are another type of sketch, which are used to determine whether an element is a member of a set, and guarantee that they will never produce a 'false negative', claiming a value is a member of a set when it is not. However, they can produce 'false positives'.

P4 offers enough flexibility to implement sketches inside the data plane, and there are a number of works now exploring the capabilities offered.

OpenSketch [166] is an earlier example of offering an architecture for implementing sketch-based monitoring application using FPGAs, rather than commodity switches, and they motivate the usage of Static Random Access Memory (SRAM) over TCAM due to the higher cost of TCAM.

Univmon [84] offers a framework built in P4 for the deployment of more generic sketch-based monitoring applications, and despite their more generic approach, shows a significant memory reduction to achieve similar accuracy to that of OpenSketch. Their implementation is only evaluated in the software-based behavioural model, but the authors suggest that their data structures would also fit on memory-constrained hardware platforms

SketchVisor [61] is an implementation of sketch-based measurement with two levels, a 'normal' and 'fast' path, with the 'fast' path trading accuracy for throughput in high load situations. The authors implement SketchVisor as an extension to OpenVSwitch [118].

Sketch-based algorithms themselves could however be the target of malicious actions. In their work, Pereira et al. [117] focus on introducing a cryptographic stage to a count-min sketch. This inhibits an attacker from being able to predict the behaviour of the sketch, and thus, they are unable to target particular flows or users. Their approach is only demonstrated in the P4 software switch, and as presented, would not work in a commodity hardware switch, such as the Intel Tofino.

In summary, a trend has emerged whereby the control plane have a finite capacity to interface with and monitor network devices. By delegating work towards switches, especially programmable ones, such as those based around P4, it is possible to achieve more effective monitoring with a reduced load on the controller. Data plane based implementations can also be more pervasive, inspecting traffic throughout the network.

2.2.1.2 Middleboxes

Deep Packet Inspection is not a capability intrinsically present within any current version of the OpenFlow specification, as it is focused on packet headers rather than payloads. P4 has similar limitations, insofar as it has limited support for variable-length headers [111], such as those found at the application layer.

Despite this, both OpenFlow and P4-based devices can be used to facilitate routing to IDS, or, in some cases, the data plane can be extended to encompass IDS features.

SnortFlow [163] proposes a means to integrate Snort [128] based intrusion detection into a cloud environment by running it as part of the hypervisor. Their approach uses the output from the IDS to generate rules to be installed into switches, though, the paper focuses on the performance implications of SnortFlow, rather than the detection and mitigation capability.

Application-aware Traffic Management for OpenFlow Networks [69] proposes an architecture whereby Deep Packet Inspection is paired on a general purpose host with a software OpenFlow switch. On receipt, packets are mirrored to both the DPI appliance and OpenFlow switch. When the DPI flags a flow, in this case, for QoS (Quality-of-Service), the controller can then apply the relevant meters to the packet. The authors do not discuss the performance implications of their solution, and suggest that further evaluation is needed before their system could be deployed in more complicated topologies.

A number of frameworks use OpenFlow to determine whether traffic should be forwarded to a dedicated IDS [39] [78] [20], these are discussed alongside other frameworks in the next section.

P4, on the other hand, does offer more scope for behaviours that would typically require one or more middleboxes.

P4 can offer a varied range of capabilities on this front. For example, Ndonda et al. [98] implement an Intrusion Detection System for Industrial Control Systems. Their implementation uses the P4 data plane to provide whitelisting, with any unrecognised traffic redirected towards the control plane. In this use case, the bulk of traffic will only pass through the data plane, with only unknown flows experiencing the performance penalty of being forwarded onto an IDS. The authors draw attention to latencies as high as 60ms even in a relatively low-traffic network. In theory, some of this delay may be reduced by delegating more decision-making into the control plane.

In P4-ONIDS [146], the authors look to extend the data plane with the installation of IDS rules, and a state-based mechanism for forwarding the first packets of a flow to a given IDS. This is used in conjunction with an off-the-shelf IDS, Snort [128]. In their work, the authors highlight the value of combining P4-based architectures to enhance existing systems.

They first take IDS rules and combine them with a compiler to generate aggregated rules describing 6-tuple matches. These are: source and destination addresses and ports, layer 4 protocol and TCP flags if any. They then prioritise these rules for installation into the switch based on a 'severity' factor. Their use of range matches can offer a level of de-duplication but may require additional TCAM in switches [42]. The authors acknowledge the limitations of only filtering on a rule-based approach, and also include a mechanism to allow the first N packets of each flow to be inspected by the IDS. To do this, they rely on a count-min sketch, a hash-based probabilistic data structure to track event frequency. To ensure that they are clearing this data structure for new flows, they trigger a data plane-based series of operations, which is bound to the size of the sketch filter.

Whilst the authors show that their approach can be effective in reducing the number of alerts reaching an IDS, whilst maintaining detection of up to 90% of alerts, they do not categorise their approach by attack type, nor do they explore how this could be further improved. Their approach also requires multiple tables to implement. Using compiled rule-sets also means that a reconfiguration of the IDS would require loading new rules into the data plane.

P4 also enables firewall-like behaviours to be implemented into switches, in this section, we focus on the broader scope of functionality being introduced into the data plane.

P4Guard [30] is an example of firewall functionality being implemented in P4. In this case, the author defines a high level policy language in conjunction with a controller and P4 data plane to implement layer 3 and 4 firewalling functionality. Their results suggest that the P4-based solution uses resources more efficiently than their previous OpenFlow-based implementation VNGuard [33] at moderate traffic loads.

Li et al. [83] propose a stateful SDN firewall, using P4 and targeting cloud data centres. Their work keeps track of connections but appears to rely on the control plane to determine state information and store it in tables. Their connection tracking relies on the TCP three-way handshake, blocking connections after a FIN packet has been sent to close the connection. The authors motivate their work as having minimal overheads by reducing the need for communication to the controller at any given time.

Security Middleware Programming using P4 [157], on the other hand, uses the data plane to offer rate based limiting, again capable of inspecting the network and transport layers of traffic. Bans are implemented using stateful memory in the data plane, allowing a quicker response than invoking the control plane. Though in this case, it is important to note that the paper does not present any evaluation or results of the solution.

P4DDPI [1] aims to push the capabilities of P4 devices further, and offers a DNS filtering mechanism implemented in the data plane. This makes use of recirculation to implement the parsing of DNS packets in the Tofino data plane, however, recirculation incurs performance costs. The authors also present the resource utilisation of each stage of the switch their implementation was tested on, which shows that the switch still has capacity for performing additional functions alongside P4DDPI.

P4Knocking [169] shows how port knocking, a feature not frequently available on hardware switching platforms, can be implemented within a P4 data plane. Port knocking [4] is a security technique whereby a TCP SYN packet is sent to a sequence of ports, at which point, the firewall will allow a connection request for a given IP to a specific port or range of ports. This has been implemented in a P4 data plane through the use of registers to store state between packets of which ports have been previously accessed by a given address. By contrast, this would not be possible in OpenFlow without invoking the controller, or extending the OpenFlow data plane to support stateful processing [6].

P4-MACSEC [59] focuses on adding MACSEC, a means for link-layer encryption, between P4 switches within a network. To achieve their approach, the authors demonstrate the implementation of a two-layer control plane, with a controller local to each switch, paired with a controller that manages all the switches. They then use P4 in conjunction with a modified version of the behavioural model to implement secure link layer discovery and configuration of encrypted links between hosts. It is important to note that they could not replicate this in hardware, despite attempting to use the NetFPGA SUME platform. The workarounds to the limitations the authors discovered would involve significant FPGA programming, and the author states that they view that this conflicts with the platform independent view that P4 offers.

In the case of Gray et al.'s [50], their P4 data plane is used to extract flow information at line-rate which is then fed to a collector. They highlight that the Tofino platform is able to inspect each packet, even at high traffic volumes. They aim to resolve the IDS capacity constraint by only forwarding relevant and aggregated features. Dao et al. [29] implement a neural network model into the P4 data plane, using a BMV2-based switch. The authors use this model to explore the impact of different model parameters on applying classifications on incoming traffic. This again highlights how P4 can be used

Gray et al.'s [50] work highlights that P4 can, in some cases, be only one component of an overall system. Gallium [170] follows this notion, and proposes a solution to use P4 to reduce the processing overhead associated with C++ middlebox applications. This process of analysing the C++ source code is used to provide a counterpart P4 application for pre- and post-processing of packets. The authors evaluate their solution with Click [90] middleboxes and show an average reduction in processing cycles between 20 and 79%. Latency is reduced by an average of 30% when paired with a hardware P4 switch. Though, this approach relies on being able to recompile the middlebox applications.

Feng et al. [40] propose a framework for enabling partial deployments, namely, determining which legacy switches should be replaced with programmable ones, how they can be managed together and how can security functionality be implemented on these switches. Their approach consists of three stages, identifying which devices should be upgraded, a unified controller that uses ARP broadcasting to route traffic through the programmable switch, and a data plane implementing rule-based firewall and intrusion detection. The author proposes using the control-plane for further traffic processing and enabling DPI, but this would have significant performance overheads at scale, given the bandwidth limitations inherent to the control plane [28].

Ding et al. [34] also motivate the need to support partial deployment of new functionality targeting programmable devices, highlighting the cost and complexity involved in replacing an entire network at once. With this, they offer a means for heavy hitter detection across the network, using as many switches as possible. In the case of heavy hitter detection, the author highlights the impact of positioning of monitoring capability, and states that when replacing a single switch and replacing it with one that can detect heavy hitters, the switch that should be replaced is *the one crossed by the highest number of distinct flows*. Their algorithm to detect the heavy hitters focuses on ISP networks, which the author describes as static, and allows a traffic matrix to be generated using historical data. Their algorithm is also sketch-based, implementing a count-min sketch using P4 registers.

In this section, we have seen how middlebox behaviours have been brought into the data plane using various means. In OpenFlow, the tendency has been towards extending the underlying software platform, or combining OpenFlow with a dedicated platform, and using OpenFlow to select traffic for further processing.

With P4, the opportunity to push more diverse behaviours into the data plane has been realised, and we see how this is used to implement additional capabilities. Equally, we start to see where the needs of networking functions compared to 'traditional middlebox' functions start to conflict. For example, being able to parse variable length application payloads. In some cases, recirculation offers a workaround, but in others, it may be more effective to continue to offload to a platform that is better suited to the inspection needed. Given P4's flexibility, this still offers a range of opportunities in terms of using P4 as part of a wider system architecture.

2.2.1.3 Security Frameworks

In this section, we study relevant security frameworks, which offer interfaces, languages and tools for building security applications using SDN.

Security frameworks for SDN vary in their scope, from targeting a particular class of threat, to aiming to provide a set of reusable tools for detection, mitigation and protection.

FRESCO [135], for example, aims to reduce the number of lines required to write security functions as OpenFlow functions by up to 90%, achieved through the use of a 'security kernel' which allows multiple security applications to coexist.

SDN4S [75] proposes a framework whereby detection is out-of-scope of the framework, with a focus purely on automated remediation of network threats. To do this, it uses a pre-programmed set of remediations that are triggered by external network events or detections. It is important to note, however, that despite the authors' focus on reduction in remediation time, there are no figures for the improvements given in the paper.

TENNISON [39] combines detection and remediation, offering adaptive levels of monitoring, from flow monitoring to intrusion detection. This enables dynamically adjusting to network capacity and resource demands. However, being OpenFlow based, TENNISON is still heavily dependent on the controller for much of the processing.

A number of these frameworks, including TENNISON are built around the Open Network Operating System (ONOS), which offers an extensible SDN controller, and a number of associated applications. Athena [79] offers a framework for network-based anomaly detection in OpenFlow networks, in conjunction with an API and a number of detection algorithms. The author motivates that there is no additional hardware required, aside from OpenFlow capable devices for implementing their techniques into the network. This is then combined with machine learning techniques to detect anomalies.

Alsadi et al. [2] propose a security monitoring architecture based on data plane programmability using P4. Their architecture is also based around the ONOS controller, and describes the three stages as 'live packet inspection', 'network capacity measurement' and 'real-time traffic monitoring. Their approach also includes support for rudimentary 'Deep Packet Inspection', whereby it is possible to add a constant byte-string to search for at a given location in a packet. With their architecture, they also offer an implementation of In-Band Network Telemetry to measure the delay of each link as traffic passes through the network, and asymmetric flow detection, where a large imbalance of incoming vs outgoing traffic may be indicative of an attack.

Yu at al. [165] focus on high-speed reconfiguration and responsiveness in the network, providing a language for dynamic packet process combined with a programmatically generated control-loop. The authors demonstrate how this can be applied to network failure detection and DoS detection, amongst other use cases.

Targeting the implementation of *Service Function Chaining*, Bonfirm et al. [9] propose a framework, *FrameRTP4*, focusing on real-time attack detection in 5G network slices. Their work focuses on how bloom filters can be used in conjunction with wildcarded rules. The author describes the benefits of wildcarded rules to reduce the number of flow entries by over 70% in some cases, to apply a sketch-based monitoring approach.

The author heavily emphasises the limitations of available resources in hardware switches, relying on figures cited by Bosshart et al. [10]. The proposed solution, *FrameRTP4* only analyses the 5-tuple in the ACL table, with separate tables needed to describe each transport protocol. This could be significantly improved in the implementation, given P4's support for metadata fields, allowing one ACL table to support multiple protocols. The author's evaluation only focuses on the software targets for P4, and does not discuss any performance implications of their approach.

Programmable In-Network Security for Context-aware BYOD policies [72] defines a framework for implementing security policies in P4 networks. These policies are written in *POISE*, a high-level abstraction language, which is processed into optimised P4 applications. The authors evaluate their solution on physical hardware and show that line rate can be achieved for security functions with a latency overhead of nanoseconds.

CloudWatcher [136] and PSI [168] take a very different approach, using SDN as the means to direct traffic through a series of security functions.

CloudWatcher [136] is a framework designed to re-route traffic through one or more security appliances, and achieves this with the combination of a policy language and the generation of routing rules to ensure that traffic is passed through a given appliance. Though the authors state that CloudWatcher suffers from both scalability challenges with large numbers of new flows, and there are edge cases where a route may not be generated.

PSI focuses on routing the traffic towards Virtual Network Functions (VNFs). These run in a virtualised cluster, which traffic is steered to, as opposed to trying to perform the inspection and detection throughout the network.

In P4, DPPX shares similarities with PSI, in that it targets VNFs, but, it implements P4-based network functions, with the motivation that the performance of a virtual P4 switch could be significantly greater than a virtualised version of a hardware appliance. To demonstrate their approach, the author shows how this can be used to direct packets to or bypass a virtual IDS system, with encrypted packets which could not be effectively inspected taking the 'fast path' away from the appliance.

2.2.2 Attacks on SDN infrastructure

With the aforementioned flexibility and capability that is introduced into Software Defined Networks, and further enabled by programmable data planes, the infrastructure itself can also be left more exposed to attack. Generally, this vulnerability can either be grouped into exploitation of bugs or vulnerabilities present within the data plane or control-plane code. A range of CVEs (Common Vulnerabilities and Exposures) exist for popular SDN controllers [140]. These vulnerabilities can resemble typical software vulnerabilities, such as Cross-Site Scripting, credential exploitation, buffer overflows etc. where an attacker is attempting to gain control of the infrastructure.

Another class of attack, which we explore further, is the resource exhaustion or Denial-of-Service attack. These attacks aim to overwhelm one or more stages of the infrastructure causing it to be unable to continue processing traffic as it otherwise would. Starting with the control-plane, and OpenFlow, a number of works have explored the vulnerability of common controllers to Denial-of-Service attacks. These typically centre around an excessive number of Packet-In messages being sent to the control-plane, the end result of this being that the controller is left in a state where it can no longer effectively process traffic [158] [87]. These attacks rely on SDN controllers taking a reactive approach to traffic processing. When configured for reactive forwarding, packets are forwarded to the controller, unless the switch has a rule installed to direct how the packet should be forwarded. An attacker can exploit this behaviour by sending spoofed packets to a switch, that it doesn't have rules installed for. These spoofed packets will be steered towards the controller for an appropriate flow rule to be installed. If the attacker sends these packets at a sufficient rate, this can then cause the controller to become overloaded. The controller will stop servicing PacketIn messages in a timely manner, and this may impact other areas of the system too, such as the management interfaces. The host running the controller may also experience excessive CPU and memory usage during an attack.

Mitigating against these attacks can be performed either in the control-plane, by the controller, or within the data plane itself. Detecting attacks in the control-plane maintains compatibility with both OpenFlow and P4-based platforms, but this is reliant on the control-plane being able to detect and remediate against the attack before the controller itself is overloaded by the attack [88].

With the move towards data plane programmability, this introduces further scope for the detection of Denial-of-Service attacks targeting the controller. The data plane can make the detection at this level before the controller becomes overloaded, avoiding the challenge of the control-plane becoming overloaded before it can respond to an attack. Detection techniques include entropy-based, where algorithms determine whether traffic is associated with legitimate traffic. StateSec [8] is an example of this, implemented using a series of finite state machines to avoid control-plane overhead. StateSec generates data that is read by the controller in order to detect attacks, which still requires the controller to be involved in protecting the data plane.

With the advent of P4, capability has been pushed further into the data plane, moving further away from the need for the control-plane to be involved in the detection of Denial-of-Service attacks. Entropy-based algorithms have been designed for P4 to use this mechanism to detect attacks [35] [54]. Euclid [65] exemplifies anomaly detection taking place in the data plane, combined with count sketches to track the number of anomalies being generated by traffic on the network. Other approaches take advantage of the programmability offered in different ways. Paolucci et al. [112] use P4 to track a characteristic of certain types of DoS attacks where the TCP port number increments with each connection attempt in a short period of time. However, their approach tracks by individual IP addresses, which may not be scalable with a growing range of IP addresses with finite space available in switch memory for tracking these flows. BUNGEE [48] furthers data plane autonomy by disseminating messages between programmable switches, with a view to reducing the number of switches involved in processing attack traffic until the attack is confined to as small a network area as possible. This aims to minimise the impact of the attack. Ripple [162] is another example of a data plane based coordination mechanism used to detect attacks, here the authors focus on link saturation.

Whilst more advanced control-planes can become susceptible to attack from overload, we can see that there are a range of approaches emerging to offload the detection and mitigation of these attacks into the data plane itself. With some work, the protection afforded to the control-plane is explicitly described, with other work, the techniques in use may be able to be applied specifically to the protection of the control-plane itself.

2.3 Summary

This chapter has detailed background and related work around network security and monitoring, categorising typical approaches based on their placement in the network. We then explored how functionality and placement have been both enhanced and challenged with the advent of SDN. We showed how current technology can offer a divergent approach between control-plane based and data plane based monitoring functionality. Control-plane based security offers broader capability, with the context that the control-plane can offer, but adds latency and performance overheads for traffic processing and the control-plane. Data plane based functionality shifts this balance in the opposite direction, pushing ‘intelligence’ into switches. This brings benefits in terms of pervasiveness, reductions in latency and offers a distributed platform. In other words, monitoring can be deployed more widely within the network.

Through the related work, we have identified a number of common trends in OpenFlow research, where there is a divergence between relying on the control-plane for packet processing, versus trying to offload computation into the data plane where it is feasible to do so. With this comes a balance of resources, trying to enable new functionality, whilst not impacting upon forwarding functionality. Approaches to this vary from optimisation to implementing bespoke pipelines, to divide resources between use-cases.

With P4, there has been renewed interest in what can be achieved using the data plane. Work has heavily focused on how functionality can be offloaded into the data plane, at times in conjunction with a bespoke control-plane to facilitate deployment. Work shows how this can be effective at providing a high-speed and low latency alternative to relying on control-plane processing, we also see how the data plane can be empowered to be more autonomous in processing and decision-making.

Through this process, we have also identified a number of benefits and shortcomings in current work, particularly in bridging the divide between programmable control-plane based architectures and those with programmable data planes. These are:

- SDN, in the form of OpenFlow and P4 can be used to effectively offer monitoring and security functionality within networks.
- Applications taking advantage of control and data plane programmability in combination can offer improved flexibility and capability than those targeting one area of programmability exclusively.
- There are a limited number of frameworks for deploying and managing combinations of programmable pipeline-based applications for the P4 ecosystem.

Especially when considering being able to integrate other existing applications, such as those originally designed for an OpenFlow-based platform.

In summary, whilst there are a range of tools and techniques now available, there is not a straightforward means for a developer to combine these. In the case that they do, it is then not necessarily straightforward to also have their new platform continue to provide 'typical' controller functionality.

We use the findings from this chapter to propose, design, implement and evaluate a framework for deploying security applications alongside P4 programmable data planes. This design considers existing tools, standards and observations from related research.

Chapter 3

Design

This chapter explores the motivations for a so-called ‘hybrid’ approach to deploy security applications into the data plane. In this ‘hybrid’ approach, we combine the capabilities offered by traditional tools and techniques, with the flexibility and decentralisation that modern SDNs can offer.

This hybrid approach aims to both enhance ‘tried and tested’ methodologies and platforms, by integrating them into SDNs, as well as enabling new techniques and tools to be implemented alongside them.

The overarching aim of this approach is to defragment approaches to enhancing network security with SDN, and to avoid, as much as possible, developers having to ‘reinvent the wheel’ for common behaviours. To this end, we promote modularity, code reusability and cross compatibility with other existing architectures.

3.1 The Case for a Hybrid Deployment

For a number of years, tools, techniques and architectures have emerged for implementing security functions in Software Defined Networks. These range from holistic approaches [39] [28] [79], considering the network as a widespread entity, to adding stateful stages into the processing of every individual packet [98] [83] [154].

However, combining these approaches can pose challenges. Whilst tools exist to implement aspects of these functions, and in some cases, to implement them alongside an SDN controller, combining them brings about challenges of scalability, flexibility, programmability and support for novel platforms, such a P4.

This results in significant 'start-up costs' for a developer or network administrator to bring these benefits into a network. There is a need for tooling to make this process easier.

Whilst the new platforms do bring benefits, the 'traditional' approaches are not necessarily redundant either. Tools for deep packet inspection [128] [101], for example, can be applied to SDN, but, they do not necessarily have a mechanism for feeding back to the controller. In other words, their feedback may be underutilised. These approaches can also be challenged by SDN's decoupling of logical networks from their underlying physical infrastructure.

These traditional approaches can also themselves stand to benefit from SDN-based platforms. Whilst a P4 data plane, for example, cannot entirely replace an IDS [111], prior work has shown that it is possible to effectively reduce the processing overhead of a given application or middlebox, by offloading common functionality into other platforms [170] [146].

We must also consider the benefits of centralised control planes [17] and control plane based processing. OpenFlow architectures have shown how effective the control plane can be, taking a global view of a network and applying policy with a holistic view [135] [39]. OpenFlow-based platforms can offer flow monitoring, traffic management, reactive network control, anomaly detection and more.

Equally, it is also important to consider how this intersects with the constraints that control plane based SDN can introduce. Controllers have a finite capacity, and switches generally have a finite capacity for ongoing communication between switch and controller [28] [164].

These trade-offs can be balanced, and the approaches may not need to be seen as mutually exclusive. A number of works in OpenFlow have highlighted the benefits of the programmable control-plane in combination with a more capable data plane offering a level of programmability [163] [69]. In these cases, it has often required a modified or customised software switch, or the use of an FPGA.

We propose that there is a need for an enabling framework, whereby the discussed benefits of centralised control of SDNs can be rationalised against the broader capabilities unlocked by programmable data planes. This will involve facilitating a combination of the two technologies, existing OpenFlow platforms do not necessarily offer the flexibility to integrate with programmable data planes as is, whereas the ecosystem around P4 is still comparatively immature and undergoing significant evolution.

We posit that this needs to be easier, and that this can be achieved through a level of abstraction introduced for existing applications and controllers. This abstraction can be seen as a compatibility layer, allowing existing applications to interact with an abstract representation of tailored device pipelines. This should have programmability as a counterpart, focusing on allocating programmable resources to applications that can take advantage of them.

In the next section, we discuss in depth the considerations that are involved in a design to combine the benefits of centralised control, whilst aiming to avoid detracting from the de-centralised benefits of programmable data planes.

3.2 Hybrid Design Considerations

In this section, we introduce a number of key considerations that must be taken into account for a hybrid approach to be successful. We focus on the immediate challenges in designing a framework to enable integrations with existing applications.

The Architecture of Existing platforms

Middleboxes [128] [101] and host-based applications have typically been designed for featuring as part of a traditional network architecture. This involves a network tap and placement at key points throughout the network. With this, they can consume a mirrored copy of traffic, or, intercept and process network traffic.

A framework needs to be able to accommodate these architectures effectively in order to coexist and cooperate with them. This may include accommodating the underlying architectures, as placement of network functions can greatly influence their performance [89].

Steering Traffic towards Devices

Given the broad scope of middleboxes available, there is a range of requirements in terms of the traffic they need to process. For example, Deep Packet Inspection is less effective on encrypted traffic [108], so an appliance may not benefit from seeing the bulk of traffic from encrypted protocols. Equally, the handshake phase of some protocols can impart useful information. SSH, as an example, uses a plaintext exchange to initialise a connection. An appliance may keep track of the rate at which these new connections are being made. Middleboxes may also provide feedback on traffic that they have processed.

Mechanisms to enable this steering may include specific pipeline stages, suited to a particular use case or application. These individual pipeline stages may encompass a series of stateless or stateful actions, where the pipeline can determine how traffic should be processed, and whether there is any associated output. This output may consist of mirroring or redirecting packets, sending packets to the control plane, updating internal counters, or sending a digest to the control plane to inform it that specified pre-conditions have been met or exceeded.

Monitoring Applications

When these messages are directed towards the control plane, they must be routed through to the relevant destination so that they can be processed. This routing may involve multiple stages depending on the level of monitoring taking place on the network.

Therefore, we need not only mechanisms to determine which traffic should be forwarded to the middle box, but also a mechanism to allow the middle boxes and appliances to feed back to the control and data planes. These platforms can produce logging output, but this is often in proprietary format, such as Snort's [128] binary logging format.

This mechanism needs to be flexible to facilitate integration with any number of potential platforms, providing a translation layer from the logging format into any network configuration requests that need to be generated as a result. The mechanism may need privileged access to individual system state, for example, logs, so it would be beneficial if the log collection mechanism could run independently of any controller. This is particularly applicable to host-based monitoring scenarios, where data such as logs often requires privileged access.

Re-allocating Network Resources

With the configurability offered through programmable data planes, there is also scope for the dynamic re-allocation of resources based on network demand. Resource contention, especially for that of This must consider both the impact of any re-configuration process, and determining an appropriate trade-off as to which resources should be allocated to which functionality.

This is driven by the contention for resources that exists between different switch features [89] [153] [120]. As a result, and especially in changing network conditions, it would be beneficial to be able to alter this balance depending on administrative needs.

Network Control

Despite these data plane specific considerations, we must not neglect the fundamental features of an SDN controller [5], which is to provide a means to manage a network from a central point. The controller must be able to service network requests, implementing relevant protocols, discovery mechanisms and fault tolerance. Again, a key factor in the design of this architecture is to be able to harness existing tools and techniques wherever possible. The architecture of the controller must also be such that it can service requests from a number of devices or nodes in parallel. Controllers themselves can be a point of failure [28] [164] [88], or a limiting factor in Software Defined Networks, therefore, it is imperative that the controller architecture is designed with scalability in mind.

3.2.1 High-Level Design Requirements

Based on the design considerations described in the previous section, we establish a series of design requirements for a framework facilitating monitoring with programmable data planes. These requirements address combining programmability and flexibility with an underlying goal of maintaining compatibility where possible.

- A management plane capable of connecting middleboxes and monitoring applications to the control plane. This can vary in complexity from processing pre-defined configurations to responding to events or notifications from the monitoring device on-demand. The management plane must support the deployment and lifecycle management of these applications including clean removal of the application from the network.

- System should work with an existing SDN controller for forwarding. This should be able to be integrated with minimal to no modification of the SDN controller. This reduces the implementation effort, whilst allowing compatibility with a wider range of network devices and facilitates integration with OpenFlow and traditional network hardware.

- Monitoring activity should not impact upon forwarding, monitoring rules must not conflict or interfere with forwarding.
- The data plane should be able to detect and mitigate against overload situations, to protect the control plane and overall network function. This helps the system resist malicious attack, as well as accidental failure.
- Provide a documented and efficient set of interfaces to integrate network appliances. Network appliances should be able to specify directives for traffic to block, alter or redirect, either dynamically or based on their static configuration. Applications may also include bespoke logic for the data plane.
- Monitoring should be able to be enabled in any programmable device, with traffic re-directed to the appropriate appliance where necessary.

In this section, we have presented the design motivations and considerations for a framework targeting an underlying programmable architecture. We have drawn on both the benefits afforded by the flexibility, in terms of moving capability into the data plane, and caveat this against the typical SDN approach of centralised processing and control. With this, we have proposed that a hybrid approach, which aims to combine the centralised processing offered by Software-Defined Networking can be combined with the distributed capability offered by programmable data planes.

3.3 Architecture and Design

3.3.1 Introduction

Building on the requirements ascertained in the previous section, we present here the design for a framework to enable and facilitate the deployment of 'hybrid' applications, where logic is spread across both the control and data planes. With this, we present the design of a new framework, designed to bridge between existing applications, Software-Defined control and programmable data planes.

The 4MIDable framework is a framework focused on providing flexible monitoring and remediation capability, targeting programmable data planes. The framework, shown in Figure 3.1, is focused on integrating existing tools and platforms with SDN and programmable switch pipelines. To enable this, we divide the framework into a series of layers, each with distinct responsibility. These layers communicate through a series of interfaces.

The *Management Layer* orchestrates between security applications and the framework, through monitoring agents. Security applications include any middle box, host-based application or appliance of which the configuration and output can be integrated with 4MIDable using *Monitoring Agents*. For brevity and clarity, we refer to all security applications and applications as middle boxes and appliances in this section. The framework is designed around this concept of *agents*, which provide the interoperability between the framework and the off-the-shelf middle boxes. This interoperability allows the middle box to influence how traffic in the network is processed.

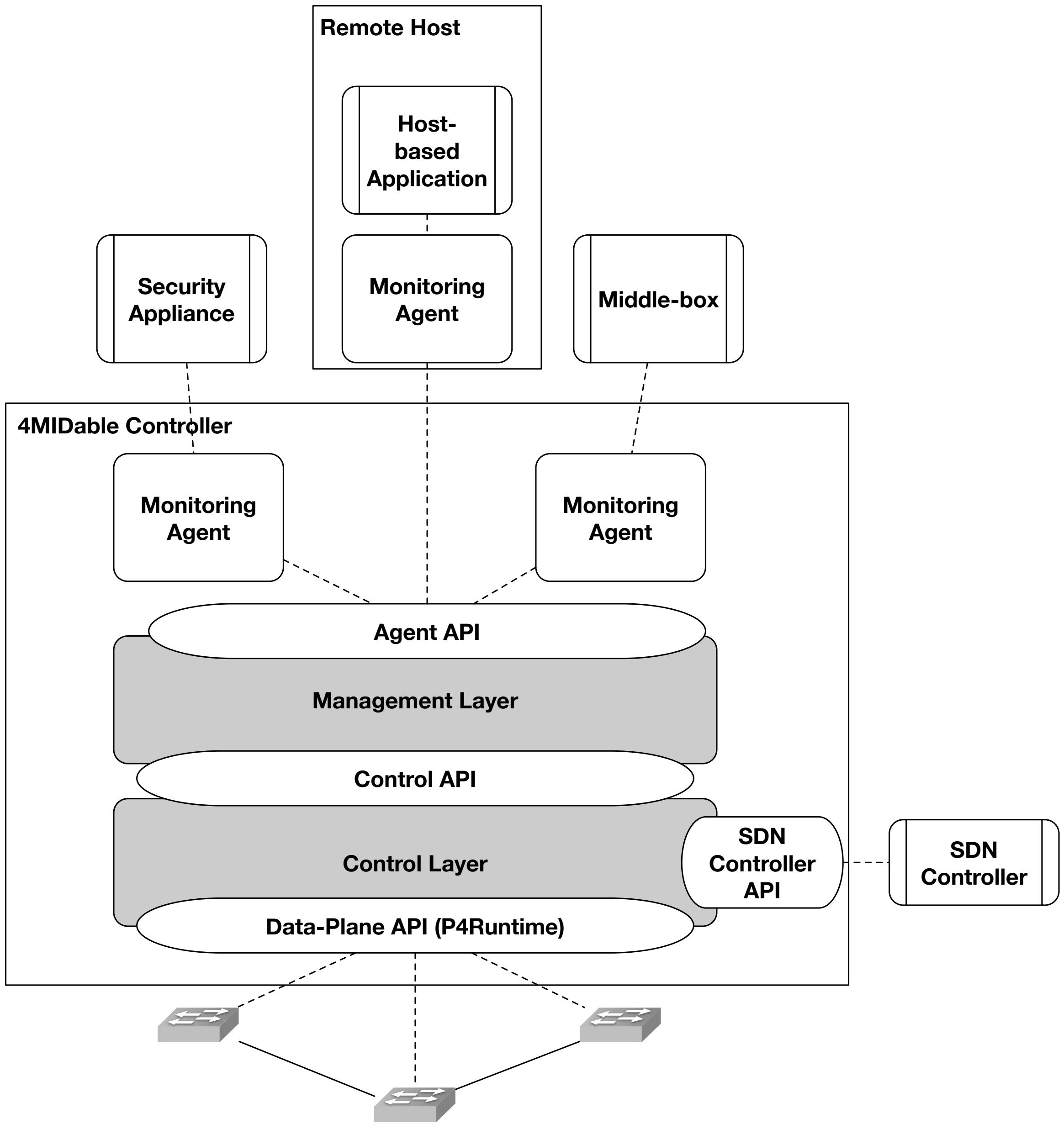


Figure 3.1: 4MIDable Architecture

The *Control Layer* provides external APIs to the Management Layer and 'SDN Controller API'. The Control Layer processes requests from the Management Layer, which can either request how to influence how traffic is processed, or request statistics from the Control Layer. The SDN Controller API offers a similar interface, where an off-the-shelf SDN Controller is able to integrate with our platform and specify flow rules that should be installed, and request statistics from the data plane. The Control Layer handles connections with the data plane, installation and removal of rules, statistic requests and pipeline configuration requests. The Control Layer is designed to provide the necessary abstractions to allow the SDN controller and other monitoring applications to co-exist.

The *Data Layer* represents a collection of programmable switches. These are sent pipeline configurations, table entries and statistics requests by the control layer. Depending on the installed pipeline, the data plane may also be capable of a level of autonomous processing. The data plane can send traffic, statistics or digest messages back to the Control Layer.

An existing SDN controller can be integrated with the controller, to provide forwarding and routing behaviour commensurate with a typical network controller [5] [151]. This is enabled by offering an interface to the SDN controller, using the P4Runtime server interface to proxy requests destined towards switches. This proxy-based approach enables the control-layer to provide an abstract pipeline to the SDN controller.

3.3.2 Management Layer

As introduced in the previous section, the Management Layer provides the link between existing middle boxes and SDN control. To achieve this, the management layer provides a framework for *Monitoring Agents* and manages their lifecycle. Monitoring agents should be able to be instantiated, updated and terminated independently of the rest of the system.

A *Monitoring Agent* is a self-contained application, which interfaces with the Management Layer. The purpose of an agent is to provide the link between an existing middle box and the rest of the system. Monitoring agents are created for each type or implementation of middle box, and are able to consume static configurations as well as dynamic outputs, such as logging and telemetry, that are generated by the appliance. For example, a signature-based Intrusion Detection System will have a static configuration, which details the networks that it is monitoring, as well as the signatures that it is trying to detect. This static configuration may be used by the agent to restrict traffic directed to the IDS to only that which is specified in the configuration. The IDS will also produce logging output in normal operation, detailing the alerts and remedial actions to take. The agent can consume these logs and produce a request to the data plane to take an action, such as blocking a specific flow or host.

When a monitoring agent needs to take action in the data plane, it generates a request message that is passed down to the data plane layer via the Control Layer. This request comprises a list of header fields, and their values. The Control Plane will process this request and generate the appropriate flow rules, if any are required.

Figure 3.2 shows the flow of data through a Monitoring Agent. Each class of middle box can produce logging output in a bespoke format, so this is handled by each agent. Agents are written in a general-purpose programming language so can take advantage of any relevant libraries and extensions to facilitate this. By making the log parser part of each self-contained agent, this can facilitate compatibility with specific platforms, such as the Snort Intrusion Detection System's [128] proprietary binary logging format.

Processing logic is again part of the self-contained agent. This is because the appropriate action may include specific behaviours associated with a particular table or pipeline stages. Once the processing logic has completed processing, it then moves to the request builder. The request builder uses an Application Programmable Interface (API), which can take a flow request or statistics request generated as part of the processing logic. This will then be serviced by the lower layers as appropriate. When a response is ready for the agent, this will again be passed back up as appropriate.

The Management Layer provides a local and remote interface for the instantiation of monitoring applications. In the case of local applications, the Monitoring Layer will launch an instance of the application for each instance of a middle box that is being monitored. For remote instances, the Management Layer provides a remote API with the same functionality as is offered to a local application instance. A remote instance is an instance of a monitoring agent running on a host separate to the controller. This approach reduces the security implications of giving a controller access to the logging output of all appliances. Instead, privileged access to any output or data is limited to the remote host.

For each agent, the Management Layer provides an abstracted request interface, that is used to send table entries and counter requests to the control layer. Monitoring agents, as discussed in the next section, are able to specify their pipeline requirements. Requests are able to either specify an exact pipeline stage to which they are associated, or, they can be installed in the table that most closely matches the request fields. This closest match approach is used to facilitate pipeline-independent applications, where the Control Layer will determine which table(s) a rule should be installed into.

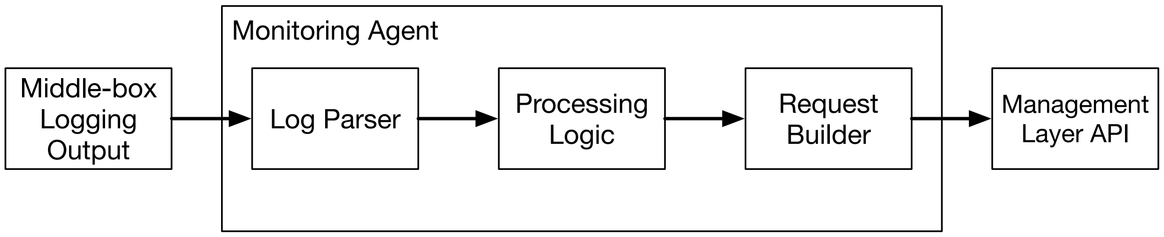


Figure 3.2: Monitoring Agent Flow

3.3.2.1 Monitoring API

The Monitoring API is provided by the Management Layer to manage the lifecycle of agents as well as servicing their requests. The Monitoring API describes the structure of messages associated with this lifecycle management, as well as the generalised design of request message.

Table 3.1 shows the API used for lifecycle management of applications. This interface is intentionally minimalistic. Applications are able to specify their own internal configurations and application-specific arguments can be used to supply these arguments. This allows applications with minimal configuration to avoid having to implement configuration parsing where it is not necessary, whilst equally offering the option of supplying detailed configurations to other applications that require it.

When an application is created, it is assigned a unique identifier, associated with that instance of the application.

Each application can implement one or more interfaces for processing traffic. Each interface is based on P4Runtime [110] messages. Applications can handle *PacketIn* messages, Digest messages and Idle Timeout Notifications. *PacketIn* messages are entire packets sent from the data plane to the control-plane and are the primary mechanism OpenFlow uses to pass data to the controller. *PacketIn* messages are wrapped in a CPU header, which is parsed by the Control Layer. When a *PacketIn* is passed to a monitoring agent, it includes a series of metadata: The ingress port and switch ID, identifying where the packet originated from, a reason identifier, which is used to identify the ID of the agent the packet should be sent to and clone, which a boolean flag identifying whether it was a clone of the packet sent to the control plane. Cloned packets typically do not have to be forwarded back to the data plane.

Method	Input	Output	Function
CreateAgent	Instance of an Application Object	Application Unique Identifier	Loads an instance of an application into memory
StartAgent	Agent Identifier, Application Specific Arguments	Status Code	Starts the application with the given identifier, returns status on success or failure
StopAgent	Agent Identifier	Status Code	Stop the agent with the supplied identifier, returns status code

Table 3.1: Monitoring Layer: Agent API

When the agent has finished processing a *PacketIn*, it is passed back to the Monitoring Layer, with a number of forwarding metadata fields. The first field, Stop

Processing, dictates whether the packet can be processed by any other agents running as part of the system. In some cases, multiple agents may have registered an interest in the same packet or type of packet. The Send field specifies whether the packet should be sent back to the data plane, to either a specified egress port, if the packet has a single destination, or to a multicast group that has been configured on the switch.

Digest messages typically provide one-way communication and in the 4MIDable-framework have a one-to-one relationship between the digest and the destination agent. Digests are typically much smaller than a full *PacketIn*, containing only a subset of headers. We have designed this part of the framework around the notion of digests being part of a pipeline-dependent application. This means that we can delegate parsing of digests to the application that created them. The first bytes of the digest must again contain the identifier of the agent. This allows the digest to be sent to the correct agents. Unlike PacketIns, there is no continued processing of the digest by any other agents.

Idle Timeout Notifications are used to alert the control-plane that a specific table entry has been removed. Agents can use these when implementing rules that timeout after a period of activity. If the rule is still required, then the agent can generate the appropriate request to re-install it. Note that these notifications may be duplicated or generated by the *Control Stage* if multiple rules share the same table entry. Generating the notification allows each agent to receive an alert that the rule has timed out, and each agent can re-install a rule if needed. This can also be used to 'greedily' reclaim unused table entries when network constraints may require it.

Field Name	Description
Device ID	Switch ID to install the flow rule into
Application ID	The ID of the application requesting the rule
Match Fields	The packet header fields to match against
Action Name	The name of the action to call in the pipeline
Action Parameters	Parameters for the named action
Priority	Priority for the rule being installed
Table Name	The table to install the rule into

Table 3.2: Control Plane: Rule Request Interface

When an application generates a flow rule to be installed into a device, it produces a Rule Request, the format of which is described in Table 3.2. Each flow rule is destined to be installed to a single device. The Match Fields and Action Parameters are both lists of fields, which can be specified in any common format. The Control Layer handles translating these into the appropriate data types for the P4Runtime. This abstraction layer provides a more generic programming interface, by allowing the developer to focus on the most convenient data type. The translation is discussed further in the *Control Manager* section.

The Management API is designed such that it can be used by applications running locally, as part of the controller, or remotely. In the remote case, the monitoring agent initiates the connection with the controller when it is launched. Relying on the agent to connect limits the access that the controller needs to each remote host. When running remotely, agents should aim to minimise the number of *PacketIn* messages required, and prefer digests. This will serve to reduce the volume of traffic that must be sent to the remote host.

3.3.3 Control Layer

The Control Layer provides interfaces between other layers of 4MIDable, and is fundamentally responsible for consuming requests from the Management Layer and SDN controller, and either installing data plane entries, or returning traffic statistics from the data plane

A detailed overview of the Control Layer is shown in Figure 3.3. The Control Layer is centred around the notion of the *Control Manager*, which is shown in depth in Figure 3.4. The Control Manager (CM) underpins the flexibility of the framework, by providing the interoperability layer between all other layers. The CM provides reconciliation of requests, translation between pipeline configurations, and an abstract pipeline representation to the forwarding controller. This abstract pipeline facilitates integration with an off-the-shelf SDN controller whilst maintaining pipeline flexibility. Our design for the abstract pipeline is discussed further in the next section.

The Management Interface provides an API for the monitoring layer. This API is used to abstract away the underlying pipeline for pipeline-independent applications, whilst offering pipeline-specific applications the full programmability available. The State Database is used by the CM to store internal state. The state database is used to facilitate data plane management, and rule installation. The state database can also be replicated to allow multiple instances of the framework to run in a distributed configuration.

The SDN Controller Interface provides an instance of the P4Runtime Server to the SDN controller connected to the framework. This allows the SDN Controller to use existing interfaces to connect to the framework, whilst allowing the framework to forward requests through the CM.

The Data Plane Interface uses the P4Runtime [110] to connect to and manage network switches. Using this allows interoperability with a number of hardware and software P4 platforms, and facilitates integration with network controllers supporting the P4Runtime specification. This also facilitates proxying requests from the SDN controller, as it minimises the data manipulation that needs to occur.

3.3.3.1 Control Manager

The Control Manager provides interoperability between layers of the system. Figure 3.4 shows the stages that take place when the CM receives a request from either the Management API, or the SDN Controller API. The CM will process this request, and it will either result in a configuration update being installed into one or more switches, or, it will return statistics, counters and meters, that have been collected from the switch.

The format of a rule request is introduced in Section 3.3.2.1. When a request is received, it is first passed to the *request validation* stage of the CM. This stage starts by checking the ID of the device the rule should be installed to. If it does not exist, the request is rejected. If the request has specified a table name, then we check that the switch is running a pipeline with that specific table and action present. We then check that each match parameter and type is compatible. A match type is compatible if the table's key can implement that match type. For example, ternary matching can also support exact matches. But, an exact-match table key cannot support longest-prefix or ternary matching. We then ensure that the specified action parameters match the specified action.

If a table name is not specified, we must ensure that the rule can be installed as specified into the pipeline. This is the approach taken by *pipeline independent* applications. In this case, we search for table(s) with the requested action and action parameters. If there are multiple tables, then we find the smallest entry with all match fields. The smallest entry is the table with the least number of unused match fields. This approach reduces the amount of unused or 'wasted' table memory, that is not being used as part of the match entry. It is possible to have this unused memory if ternary matches with an empty mask are used. For fields with an empty ternary mask, any field will match.

Once a table to install the rule has been identified, the rule request is then stored by the *Request Logging* stage. The request logging stage stores the entire rule request in memory. The request logging stage is used to assist in reconfiguration if a switch pipeline is updated, and detects any duplicate requests before they are passed to the next stages. Each entry in the request log is given a unique identifier, that can be referred to by other entries. This linking is used when rules are consolidated together.

The stored request then moves to the rule consolidation stage. The rule consolidation stage processes requests into individual table entries to be installed into switches. For each request, the consolidation stage first checks whether the rule is already matched by any table entries that are already installed in the switch. If equivalent entries are already installed, the consolidation stage stores a reference to the request. An equivalent entry is defined as an entry where either a rule matches exactly, or an already installed rule would have the same action. For example, a rule matching an IP address exactly and forwarding to an arbitrary port would be treated as equivalent to a rule already installed matching the same IP prefix and sending to the same port. This approach reduces the volume of table memory used, but can detract from statistics gathering, if table-based statistics counting is in use. When an equivalent rule is already installed, the shared database is updated with a reference to the rule. This is to ensure that an appropriate rule is installed if the equivalent rule is removed.

Finally, before installing the table entry, a table occupancy check is run. This is to ensure that there is enough space in the switch's memory to install the rule. The size of tables is generally set in P4 platforms at compile-time. If there is not enough space to install the rule, the Control Layer will return an error to the Management Layer and the Monitoring Agent that attempted to install the rule.

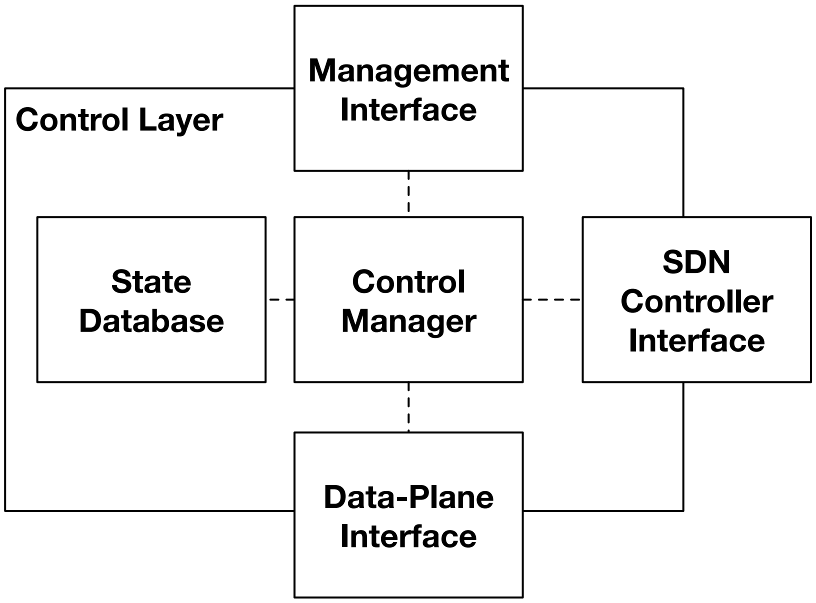


Figure 3.3: Control Layer

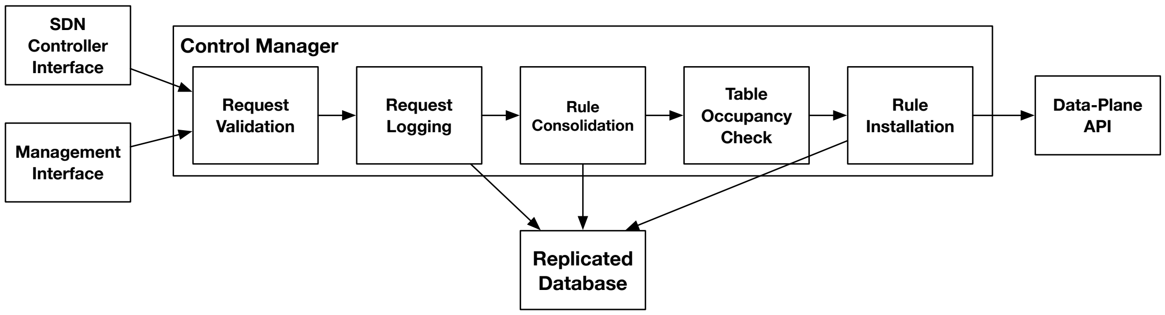


Figure 3.4: Control Manager

3.3.3.2 SDN Controller API

The SDN Controller API provides an interface for an existing SDN controller to manage forwarding on the network. We take this approach, as this allows us to retain flexibility for individual monitoring agents, without requiring modification on the controller side. This is whilst the integration offers us the network forwarding functionality and topology discovery expected from a contemporary network controller. We focus our integration on the ONOS [5] controller, as it has support for the P4Runtime, the platform-independent control plane API for P4 programmable switches. Using the P4Runtime throughout facilitates passing requests between the SDN controller and the data plane. The SDN Controller Interface relies on the 'agnostic pipeline', the requirements of which are discussed in Section 3.3.4. By using this application-independent pipeline, the SDN controller is guaranteed to have the functionality required for forwarding behaviour.

When requests are received from the SDN Controller, they are in the standard P4Runtime format. This is then passed to the control manager, following the same series of steps as a request received by an agent. This processing ensures that the

Method	Input	Output	Function
Set Pipeline	Pipeline Configuration	Response Code	Change the P4 application in use
Read Table	Table Identifier	Table Entry List	Retrieve switch table entries
Write Table Entry	Table Entry	Response Code	Install a data plane forwarding rule
Read Counter	Counter ID	Count Read Response	Read one or more counters from the data plane
Send PacketOut	Packet	Response Code	Send a PacketOut to a given switch

Table 3.3: Data Plane Layer: Data Plane API

rule can be installed as requested. Any errors in this case are returned to the SDN controller in the appropriate P4Runtime error format.

Typically, frameworks will interface with an SDN controller, using an application on the SDN controller to provide the interface between a security framework and controller. However, with the move towards programmable data planes, it becomes challenging to programmatically handle pipeline diversity where monitoring agents are expecting different pipelines.

3.3.4 Data Plane

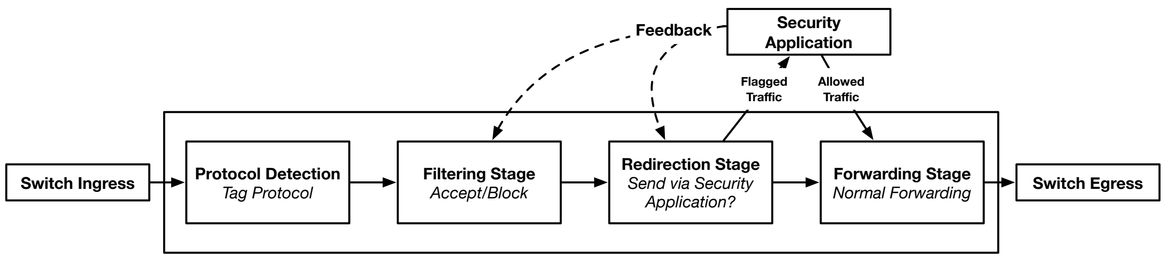


Figure 3.5: Data Plane Pipeline Architecture

The data plane consists of a number of programmable switches performing a combination of monitoring and forwarding functions. The system is designed such that it can be deployed in tandem with non-programmable devices, using standard protocols to encapsulate traffic that is to be monitored or otherwise processed differently.

3.3.4.1 Pipeline Stages

This section describes the abstract flow that is suggested through the stages of the data plane pipeline. These stages are shown in Figure 3.5. This pipeline is customisable to suit network administrator requirements, with the forwarding stage following the 'Agnostic Pipeline' principles as discussed.

Parsing

Packet parsing is the first stage of any P4 pipeline. Packet parsing separates the bitstream into individual header fields that can then be used by the switch to match against table entries, update counter values and so forth. Network traffic headers are typically standardised, for protocols including Ethernet, ARP, IP etc. We provide a standardised naming convention for these header fields and parser template code, to accelerate development by avoiding having to re-implement network parsers. This is based on the naming conventions described in the $P4_{16}$ specification [111]. We only implement a subset of network protocols, as administrators may choose to minimise support for protocols in the interest of switch resources. For some protocols, these standard parsers may not be sufficient. In this case, the network administrator can use the supplied convention and abstract pipeline as a starting reference point, adding in their functionality as needed.

Detection Stage

The detection stage is designed to categorise traffic according to rules from monitoring applications, for example, tagging traffic belonging to specific application-layer protocols. This application layer detection can be customised to suit, but due to the resource overheads, is not added by default. In P4, application layer detection requires parsing the application-layer headers, which can often be of variable length. The HTTP protocol is an example of this, where the HTTP headers cannot simply be described as a series of fixed length byte fields. However, taking HTTP as an example, the first bytes of the header will always identify the action verb, so can be used to identify the HTTP connection taking place.

Even some encrypted protocols, including SSH, have plaintext handshake stages, which again can be used to identify a particular type of connection taking place on the network. This again is left to the administrator's decision, as each protocol detection will require a portion of table memory that could be used for forwarding or other monitoring purposes.

The detection stage is also used to provide packet counters and metering of traffic. We use this stage as the statistics can then be used by both the forwarding and monitoring sections of the framework. These statistics include port counters, to measure the volume of traffic being processed on each port. Counters can also be attached to tables in each stage for more fine-grained statistics gathering.

Filtering and Re-Direction Stage

The filtering and re-direction stage is able to block, mirror or re-direct traffic. This can be based on the detection stage, allowing decisions to be made based on the protocol that has been detected. This can also include stateful thresholds, for example, if a certain number of packets have been seen meeting certain criteria. Stateful processing can be implemented entirely in the data plane, and managed using table entries that have been configured using the data plane. This stateful processing can also be used to block traffic, such as when rate limits have been exceeded by a flow, or when a rule has been installed by the control-plane to block one or more flows.

Traffic directed towards the data plane is encapsulated in a CPU header. The CPU header is introduced in Section 3.3.2.1, and contains the Device ID, the Ingress Port that the packet arrived on, a reason identifier, used to route the packet to the correct monitoring agent, and a flag to identify whether the packet was a cloned packet. Cloned packets do not need to be returned to the data plane to avoid interrupting traffic flows.

When traffic is re-directed, it can either be re-directed to a particular port, which is most effective when an appliance is connected to that port on the device. Alternatively, the traffic can be encapsulated and forwarded to another switch. This encapsulation approach means that we can redirect traffic to monitoring appliances that are not directly attached to the switch. Whilst this may not be feasible for all traffic passing through the switch to be re-directed, it enables a targeted monitoring approach.

Forwarding Stage

The forwarding stage is the realisation of the concept of the ‘agnostic pipeline’ introduced earlier. The goal of the agnostic pipeline is to provide a uniform interface for forwarding and simple monitoring behaviour, without having to add any specific pipeline code. This allows monitoring applications to run across pipelines where they are not introducing any pipeline-dependent behaviour. The agnostic pipeline also reduces development time, by providing a baseline implementation of the pipeline for network administrators, that can be used immediately. We use reference implementations as a guide to the functionality that is needed for a minimal implementation that is compatible with an SDN controller. By providing too many features in the reference pipeline, we could risk reducing the programmability that is exposed to the rest of the system.

Taking the ONOS 'basic' pipeline as a reference point for the minimum functionality needed for integration with the table, we see that there are a number of functionalities provided as part of the pipeline. These are detailed in Table 3.4.

As we can see, a number of the functionalities included within the ONOS Basic pipeline have already been discussed as part of the pipeline. We offer support for all packet headers described by the ONOS pipeline. Port counters are handled in the Detection Stage, whilst port meters are not used in our implementation. Packet I/O is the stage that handles sending packets to and from the controller, and is described as part of our framework. The forwarding table describes a baseline that would need to be supported to operate with ONOS. The implementation of this table is discussed further in the next chapter, taking memory and staging requirements into account. The Weighted Cost Multipath Routing (WCMP) section is not relevant to our use-case, and is simply an implementation of a multi-path routing algorithm. If this was required, a network administrator would be able to add this as part of their pipeline design.

Section	Functions
Parser	Ethernet, IPv4, TCP and UDP headers
Port Counters	Counters for Packets In and Out on each switch port
Port Meters	Meters to determine if traffic is exceeding a certain rate
Packet I/O	Handles packets being sent to and from the controller
Forwarding Table	Ternary matches on port, Ethernet Addresses, IPv4 addresses and protocol, and UDP or TCP port
WCMP	Weighted Cost Multi-path Routing

Table 3.4: ONOS Basic Pipeline: Required Attributes

Based on this, we can ascertain that we must have support for Ethernet, IPv4, TCP and UDP headers. This combination is sufficient to allow the controller to handle Layer 2 and Layer 3 network forwarding. A table matching on the Ethernet Type and invoking the controller can then handle other types of encapsulated traffic. This includes ARP (Address Resolution Protocol) traffic that maps IP addresses to Ethernet addresses and is used by end hosts to communicate. We also offer support for VLAN tagging, as this can be used for traffic encapsulation when redirecting traffic towards monitoring appliances. The pipeline design used by ONOS also highlights how ternary matches are favoured. Ternary matching allows exact matching, where a field must match exactly, as well as mask-based matching, where only part of a field has to match. This gives the best flexibility by allowing the table to offer exact matching, mask-based matching and by extension, an approximation of longest prefix matching. Longest Prefix Matching (LPM) matches based on the entry with the longest mask prefix. This can be approximated with ternary matches and match priorities.

3.4 Discussion

At the beginning of this chapter, we presented a series of design requirements for a 'hybrid' approach to deploying applications for SDNs with programmable data planes.

To address these requirements, we have presented an architecture for a modular framework, designed to bring together various elements and capabilities of software defined networks. Through this, we achieve both increased flexibility, through the support for multiple data plane applications simultaneously, and increased integration with off-the-shelf platforms. We also offer an interface for existing SDN controllers, to allow forwarding to be managed by a platform that is designed for

The architecture described builds on the typical SDN architecture of flexible control planes, by adding support throughout for flexibility at the data plane level. This provides greater flexibility whilst simultaneously retaining functionality and compatibility with existing platforms. Our abstraction layer facilitates this underlying programmability further, by enabling pipeline-agnostic applications to run across pipelines, without relying on any modifications to the applications themselves.

The 4MIDable architecture requires significant development, in terms of the controller framework to enable this, the programmable interfaces to manage and schedule monitoring agents, as well as the appropriate pipeline guidelines to facilitate building pipelines for the 4MIDable platform. With a focus on interoperability, as well as backwards compatibility, the framework will also support partial deployment, especially where programmable switches can be placed closer to middle boxes and monitoring devices. We also allow for horizontal scalability, by designing in a state database which can be shared between instances of the framework.

Chapter 4

4MIDable Implementation

4.1 Introduction

In this section, we describe the implementation of the 4MIDable framework, a realisation of the design presented in Chapter 3. 4MIDable focuses on integrating security appliances with Software-Defined Networks. This approach is underpinned by the flexibility offered by P4 [111] programmable switches. Our design targets both an open-source software switch, as well as commodity hardware platforms with P4 support.

By integrating existing appliances with an SDN architecture, we aim to provide greater flexibility in determining which traffic needs to be forwarded to an appliance, as well as enabling appliances to provide feedback back into the network. This feedback can consist of traffic requests, or remediation requests where traffic associated with certain flows is blocked.

As well as presenting the implementation of the core framework, we describe two use-cases for the framework. The first, P4ID, integrates an off-the-shelf Intrusion

Detection System, Suricata [101], with the framework. P4ID reduces the volume of traffic reaching an IDS, by focusing on the initial traffic at the start of each network flow. The second, P4Protect, shows how we can harness the data plane to provide protection against Denial-of-Service attacks to existing SDN infrastructure. We demonstrate this protection taking place not only with the 4MIDable framework, but also with the ONOS [5] SDN controller.

4.2 4MIDable Framework

As introduced in Figure 3.1, the framework design comprises a series of layers, with each layer handling separate responsibilities in the flow of data through the system. The *Control Layer* provides the core of the framework, handling the processing and passing of messages between the data plane, monitoring and SDN controller layers. The Control Layer also handles connection to data plane devices. The *Management Layer* provides the means with which to run individual monitoring agents that produce traffic requests to be installed into switches, and consume packets, digests and statistic updates from the Control and Data Layers. The *Data Layer* is composed of a series of programmable switches being managed by the framework, and these can be loaded with a pipeline configuration supplied by *Management Layer* applications.

The implementation of this framework realises the design requirements for a hybrid approach presented in Sections 3.2 and 3.2.1. To satisfy the requirements for flexibility and well-defined interfaces, we use the Go [36] programming language. Using Go provides support for using the P4Runtime [110], whilst facilitating multi-threading, and being designed for a microservice based architecture. Language features of Go we use include the flexibility offered by interfaces, combined with the message-passing primitives and concurrency offered by go-routines. Go also offers a number of open-source libraries for interacting with applications and other systems.

4.2.1 Control Layer

4.2.1.1 P4Runtime Helper

The P4Runtime provides 'a control plane specification for controlling the data plane elements of a device defined or described by a P4 program' [110].

To do this, the P4 compiler generates a 'P4Info' artefact describing the behaviour of the data plane in terms of the semantics that the controller must use to interact with the data plane.

The P4Runtime Helper is a translation layer which consumes this artefact and is fundamental to the operation of the Control Layer, and offers the flexibility needed to allow the Monitoring Layer to provide abstracted *Flow Requests*, abstracting the agent programmers from having to build individual table entries. This approach promotes code re-use and provides an in-built level of validation, by ensuring that data-types are correct when table entries are being sent to devices.

The P4Runtime Helper consumes the P4Info configuration file describes the objects (tables, counters, registers, etc.) that are contained within a P4 pipeline, and maps their human-readable names into individual and unique identifiers that are used by the P4Runtime. Each pipeline configuration requires its own instance of a P4Runtime Helper.

In operation, the P4Runtime Helper provides a series of getter and builder methods. These getter and builder methods are built around the P4Runtime data types, and are further abstracted by the Control Manager into other layers. A

GetByName method exists to get the identifier of any entry in the P4Info file by the human-readable name. The method also takes a type enum, which specifies the P4Info data type, such as Table, Counter etc. that the entry is associated with.

The builder methods use this functionality extensively, by providing a wrapper around other request types. The builders implemented in the P4Runtime Helper include those for Table Entries, Digests, Counters and multicast configuration. Taking the Table Entry builder as an example, applications follow a series of steps to build the appropriate request. First, a TableRequest is built, which specifies Table Name, Action Name, lists of Match and Action Parameters, Priority and whether the entry should be the default entry for a given table. The TableEntry is a data structure we have implemented as a wrapper for the P4 primitives. During the processing of the request, the helper performs a series of data validations, producing a byte-string for each match parameter passed. The validation step finds the appropriate match field in the P4Info file, converts the interface to a byte string, and then pads it appropriately. This padding is required by the P4Runtime, otherwise an error is generated. The conversion step supports parsing byte arrays, strings containing MAC or IP addresses, hex-encoded values starting with '0x', as well as unsigned and signed integers.

This processed request then passed to a method (BuildTableEntryFromRequest) to build the P4 Update message which can be installed to the switch. This method then returns a P4 Update message. P4 Update messages can be batched for installation into switches, so this approach can offer scope for performance optimisations within the system.

4.2.1.2 Control Manager

The Control Manager, as introduced in the design chapter is the core of the Control Layer's rule processing functionality. The Control Manager provides external interfaces for an SDN controller, or, a locally running controller application, an interface between the Monitoring and Control Layers, as well as management of the data plane. Management of the data plane includes processing requests that are destined to be installed into switches, or for requests for data to be returned from them.

When a request is passed to the Control Manager, it first passes through the *request validation* stage. This stage ensures that the rule or request can be installed or serviced appropriately. If the request is to read data back from the data plane, the request validation stage ensures that the entity to be read exists using the *P4Runtime Helper*. If an entry is to be installed, this stage uses the *P4Runtime Helper* to identify the table, action and stage of the pipeline to install the entry into.

As entries can be supplied either as abstract entries, to be installed into any table which can implement that request, or they can be supplied as an entry specifying an exact table. For entries supplying the table name, we first check that the table name exists in the P4Runtime Helper. If it does, we use that ID to get the actions associated with that table, to ensure that they match, then validate the fields. This order reduces the number of checks we are making for each request, if the request is an invalid one.

When the table name is not supplied, we first request a list of tables with actions matching the one supplied with the request. If there are multiple tables with matching action names, we then check which tables of this subset implement all the match fields that we require, whilst not specifying any that we do not have any entries for. If both tables implement all required match fields, we then move to the total size of match entries. Matches that would occupy the least table memory are prioritised. This is based on the number of keys, an entry can be inserted into a table with unused or null value keys, provided the match type is LPM (Longest Prefix Match), Ternary or Range matches. However, each entry in a table will occupy the same space in memory, regardless of how many match fields are used. Using the P4Info file, and the bit-width of each match parameter, we can identify the 'smaller' table to fit the entry into. Semantic guidelines are also given in the data plane section, to discourage the re-use of names where the behaviour of the action is different, or the behaviour occurs at a different stage in the pipeline.

Once a request has been successfully validated, it is passed to the request logging stage. This interfaces with a MongoDB [147] database. We use MongoDB as it relies on a NoSQL approach, which makes it more flexible for storing requests. Requests can be comprised of a varying number of fields, therefore an SQL database would not be suited to an architecture where the underlying pipeline and structure is liable to change, even in light of the abstract pipeline, where a number of stages are provided with table guidelines. When a request is received, it is stored in the database as part of a *collection*. *Collections* in MongoDB can be described as being conceptually similar to SQL tables. We define a collection for requests received for each table, indexed by the match fields of each entry. MongoDB also allows us to build indexes based on these requests, making look-up of values more efficient, by reducing the scope of the search that the database has to make with each run. Using an external database also facilitates the horizontal scaling of the application, storing state in a replicated database allows multiple instances of the controller to communicate and coordinate rule installations and management.

When past the validation stage, rules move into the consolidation stage. The consolidation stage exists to reduce table memory occupation as well as avoiding conflicting rules in the data plane. A key part of this stage is to make rules broader where possible. That is to say, that if two rules have the same associated action, and one can be described as a subset of the other rules matches, the two can be combined. This approach can consolidate multiple rules into a single table entry, but does reduce the number of statistics that are available when direct counters are being used. Direct counters are those where a counter is associated with each individual table entry. The first stage of the rule consolidation stage is to search for any entries with a matching action signature. A matching action signature will describe identical behaviour, and reduces the next search, which focuses on the match keys that form part of the match. We use MongoDB's search capability to search for any entries where all keys match. If all keys match, then the rule request is linked to the existing entry using a database field. This allows the table entry to skip removal in the data plane if another application is still requesting a particular rule. If not, then the entries are searched for any masked matches, where the masked value would also include the rule we are installing. This is achieved simply by using equality operators and bit-masks. This process is more resource intensive, adding a delay to the control plane before the rule is installed, so can optionally be skipped.

The final stage before rule installation is a table occupancy check. This takes place after the consolidation stage, because it is possible, through the consolidation stage to be at a stage where no additional table entries need to be installed. In this case, this stage is bypassed. Otherwise, for the table entry that has been requested, we ensure that installing the table entry will not conflict with another that has already been installed. This is to prevent the delay in waiting for a switch to determine whether an entry has already been installed.

Any errors throughout the Control Manager process are passed back to the Monitoring Layer, using a text-based error message as well as an *enum* type, which identifies the processing stage at which the request processing failed. This can be used by the agent to identify where there was a problem with the request. Typically, a request will fail either because it cannot be installed into the pipeline in use, or the table to install it in is already full.

Rule installation then proceeds after this, which uses the P4Runtime to send an *Update* message to the data plane. These messages can optionally be batched for better performance, invoking a remote procedure call once in a given time period rather than once per individual table entry. This may offer better performance when agents are generating many table entries in short periods of time.

4.2.1.3 Data Plane Interface

The data plane interface is a wrapper around the P4Runtime and handles communication with network switches. This includes the initial connection and handshake process, where the controller registers itself with a switch, as well as pushing the correct pipeline configuration. Once the pipeline configuration has been installed using the P4Runtime, the data plane interface brokers messages from the data plane, encapsulating them in the appropriate data type and passing them up to the control plane.

The data plane must maintain a gRPC stream channel for each switch, which is used to furnish asynchronous communication between the control plane and the switch. To enable this, we use a 'goroutine' and series of Go channels, which are message passing primitives, to listen for and handle any messages coming from the data plane.

4.3 Management Layer

4.3.1 Monitoring Agents

Monitoring agents provide the link between existing middle-boxes and host based applications, and the framework. An agent will do one or more of the following: parsing static configuration, parsing dynamic configuration and logging output, processing PacketIn messages or digests from the data plane.

Monitoring agents are implemented in Go when running as part of the controller, for compatibility with the rest of the controller. This also offers the best performance, by reducing the overheads of passing data between different stages of the application. Each agent must implement two mandatory interfaces, a Start and Stop interface. These are used to allow the agent to make any connections, load any files required and install any initial flow requests as part of the static configuration read as part of the middle-box. The Stop method is used to enable agents to clean up gracefully before shutting down, this may involve terminating connections, clearing up any temporary files, and removing requested flow rules where possible. Note that on termination, the Control Manager will be notified and will search the database for the identifier of the terminated application. Any rules associated with it will automatically be removed.

Static configurations will be complete once an agent has been marked as started, and the agent thread will sleep until called from an event arriving at the agent, such as a digest or PacketIn. For dynamic configurations, agents can use Golang libraries to monitor logging output, APIs and more. In this case, the main thread of the agent, along with any spawned go-routines will monitor the output and generate rules accordingly. When a request has been generated, it will be passed into an asynchronous channel to be consumed by the Management Layer.

Monitoring agents use the Monitoring API to communicate with the rest of the framework, which is discussed in the next section.

4.3.1.1 Monitoring API

As introduced in the design chapter, the Management API is used to orchestrate between individual agents and the rest of the framework. The primary purpose of an agent is to parse configuration and logging output of a middle-box, security application etc. and translate this into configuration or flow requests to be installed into the network. The internal logic of each agent is user-supplied, but we use a common interface between agents to simplify development and facilitate integration.

We intentionally make the interfaces as minimal as possible, with a focus on making agents lightweight so that they can co-exist more readily without competing for resources. Local agents implement an 'Agent' interface, which provides methods to gracefully start and stop an agent. When an agent is launched, it is supplied by the Monitoring Layer with a numeric ID. This ID is used to both identify requests that have originated from a particular agent, and directing data and control-plane traffic back to the agent.

The Management interface is implemented in one of two ways. Local agents use native Go interfaces, this is to improve performance and avoid the overheads that are incurred when using gRPC to send messages locally. Remote agents use a gRPC service definition, exposing identical functionality to the interface offered to local agents. We use gRPC, as it is the same protocol that underpins the P4Runtime, and for remote agents, offers compatibility with a number of common programming languages including Golang and Python.

In either case, agents can implement a series of interfaces to process *events. Events* are a wrapper for P4 messages being sent from the data plane. Events can be one of the following: *PacketIn*, *Digest*, *Table Idle Timeout Notifications. PacketIn* messages are sent to applications when the data plane has requested that the whole packet to be processed by the agent. *Digests* are used for the data plane to notify the control-plane of something in the data plane meeting defined criteria. *Table Entry Idle Timeout* notifications are used to alert the agent that a given rule has been removed from the data plane for inactivity.

Agents only present these interfaces if they are capable of processing the request. If an event is directed towards an agent that cannot process it, it is either logged and dropped, or can optionally be passed through to a default fallback application or destination. When messages are received by the data plane interface, they are encapsulated in a *struct* for each type of message. To facilitate packet processing, this struct contains the ID of the destination application, the port from which the message was received for *PacketIn* messages, and the ID of the switch that sent the message to the control plane. The ID is used by the Monitoring Layer to route the request from the Control Layer up to the appropriate layer. For *PacketIn* messages, we also provide a flag to identify whether a packet has been cloned to the control-plane. This can be used by agents to determine whether a PacketOut may need to be generated to avoid interrupting traffic flow.

Whilst the gRPC interface offers the same functionality as for ‘local’ agents, bandwidth and latency must be taken into account, as contacting the controller can introduce performance overheads [164]. Where bandwidth and latency are of the greatest concern, we recommend using techniques such as digests or packet truncation, to reduce network load.

Pipeline management is another feature offered by the Management API and the framework. Given the complexity of dynamically merging data plane applications, we take an approach where monitoring agents rely on the use of tags. These tags identify the data plane specific capability required by each application, and allows a network administrator to balance the capabilities of the pipeline between different applications and use-cases, whilst allowing for potential optimisations where functionality can be shared between different aspects of the system.

When an agent is launched, it provides one or more pipeline tags to the controller. If the currently loaded pipeline can service the requested agent, then the agent is launched successfully. Else, the controller will search for a list of potential pipeline configurations that would be able to service all currently loaded agents. If a suitable agent is found, then the network administrator can launch a network reconfiguration. During this reconfiguration, the new pipeline is loaded into each switch, and all table entries are re-installed by the Control Manager. The network reconfiguration step is only able to be launched manually, as there will be an interruption in forwarding during the reconfiguration stage.

If the agent is not compatible with the pipeline and there is no suitable alternative, the network administrator will be presented with an error message, stating that the requested agent is incompatible with the pipelines currently available to the platform. This is followed by a list of all pipeline tags that are currently in use, and by which monitoring agents. This allows the Network Administrator to either load an appropriate pipeline into the system that will service all requests, or, will allow them to deactivate a conflicting monitoring agent.

4.3.2 Data Plane Layer

The data plane layer underpins the flexibility that is offered by the 4MIDable framework. To achieve this flexibility, whilst maintaining the centralisation of control offered by SDN, we make extensive use of abstractions. These abstractions allow us to provide a functionally equivalent 'abstract' pipeline across distinct data plane applications. Our design is such that multiple data plane applications can be implemented within a single pipeline configuration. Whilst work exists [56] to programmatically merge pipelines, this can place constraints on the flexibility that is being offered by the applications, and may not give the optimum combination of resources for a given use case. Instead, we provide a series of 'tags', associated with a particular data plane layer capability. Each pipeline configuration can be customised to expose a series of these tags, which are then used by the controller to determine which applications are compatible with the data plane.

4.3.2.1 Reference Abstract Pipeline

The reference pipeline is a pre-defined P4 application that describes the pipeline ordering. We provide this to ensure compatibility with existing SDN controllers, and this is not mandatory if an existing SDN controller is not being used with the framework.

The reference pipeline contains a series of naming conventions, based on the naming conventions included as part of the P4$_{16}$ and PSA [53] specifications, along with those used in examples given as P4 tutorials and as part of the BMV2 [103] repository. These naming conventions not only refer to the use of CamelCase vs snake_case for different types, but also, the abbreviation guidelines for common words including ‘source’ to ‘src’ and ‘destination’ to ‘dst’. Table names should be unique, for semantic clarity and actions should have a 1-to-1 relationship with tables to better enable direct counter functionality. This list is not exhaustive and further documentation is supplied with the reference P4 application.

The abstract pipeline provides a core set of functionalities that are needed for operation with a commodity SDN controller, whilst providing a pipeline design that can add monitoring capability into the network. These core functionalities are exposed by providing the SDN controller the P4Info information associated with the abstract pipeline, as supplied with no modification. This allows the controller to be compatible with any pipelines that are a superset of the functionality offered in the abstract pipeline. To enable this, we use P4 annotations to ensure that identifiers for tables and actions remain consistent across compiler runs. Using these identifiers enables the programmer to insert the required functionality at the right stage in their application.

The first stage of the abstract pipeline is packet parsing. We provide a minimal parser, which supports parsing Ethernet, IPv4, TCP and UDP headers. This set of headers was chosen as it offers compatibility with the ONOS 'basic' pipeline, which is shown in Table 3.4 in Chapter 3. The parser can be extended to add support for additional protocols as required.

The *Detection Stage* is supplied in the abstract pipeline as an empty control. The detection stage is not needed by the SDN controller. For a network administrator or programmer, the detection stage can be used to implement counters, meters and registers used to keep track of packets meeting specific criteria. This stage is also used for table entries associated with application layer detection and can be used to flag metadata fields associated with a particular filtering, forwarding or blocking table. For semantic clarity, the *Detection Stage* should not change the egress port of a packet.

The *Filtering and Redirection Stage* is used to take action based on the results of the *Detection Stage*. These actions can include mirroring, dropping, encapsulating and multicasting packets to another destination. This stage may be split across ingress and egress pipelines, to allow cloned or multicasted packets to be encapsulated appropriately before being sent to their destination. Typically, this stage will consist of one or more tables, that will change the path of a packet, based on detections made in the previous stage. This stage is again not used by the SDN controller, it is managed by the 4MIDable framework.

The *Forwarding Stage* is the stage that is exposed to the SDN controller, and is heavily representative of the 'basic' ONOS pipeline in a default guise. Using this as a reference point offers us information on the minimum capability required for operation with the SDN controller. Programmers are able to extend this further if more forwarding functionality is required, at the expense of switch resources. To simplify the pipeline design and ensure compatibility, the forwarding stage implements all functionality that the SDN controller requires in one stage. This includes port counters to gather port statistics, packet I/O where traffic is being sent to and from the controller, and the 'forwarding table' which offers ternary matches on switch port, Ethernet Source and Destination, IPv4 addresses and protocol and layer 4 protocol.

4.3.2.2 Adding Programmable Functionality to the Abstract Pipeline

When adding functionality to the abstract pipeline, programmers and network administrators are afforded the flexibility to determine how resources on the network should be allocated between different resources and stages. In the abstract pipeline, we have described the stages at which programmers would be likely to be adding this functionality, primarily in the detection and filtering stages. Administrators can also optionally inhibit other stages of the pipeline from executing for a given packet, if certain criteria have been met. This however would reduce the SDN controllers visibility of particular traffic flows, which may impact upon traffic management schemes that are in use on the network.

We exemplify how functionality can be added to the data plane in two realisations of applications built around the concepts introduced in the 4MIDable framework. P4ID brings pre-filtering in the data plane to Intrusion Detection with the goal of increasing the effective capacity of an IDS, without any modification to the IDS itself. P4Protect proposes a mechanism for protecting control-plane infrastructure from network-based attacks.

4.4 4MIDable Applications

In this section, we present the implementation of two applications for the 4MIDable framework. In P4ID, we focus on using a *Monitoring Agent* to provide integration with and feedback from an Intrusion Detection System. P4Protect focuses on integrating monitoring without requiring any modification to upstream applications.

4.4.1 P4ID

P4-Enhanced Intrusion Detection (P4ID) is an approach to use the data plane as a pre-filter for Intrusion Detection Systems. The pre-filter is designed to reduce the volume of traffic reaching a given IDS whilst enabling the IDS to make detections equivalent to receiving traffic directly. This serves to increase the effective capacity of the IDS, as well as making tunnelling a subset of traffic for inspection across the network a more viable proposition.

P4ID is an application written using the 4MIDable framework, targeting the programmability offered by monitoring agents, as well as that offered at the data plane level. This approach takes advantage of protocol handshakes, and that, malicious flows will typically be detected by a signature early in the flow's lifetime. The majority of network traffic is now encrypted, but handshakes can still be unencrypted, such as in the SSH protocol, or the Server Name Identifier in TLS traffic.

By reducing the volume of traffic that needs to reach an Intrusion Detection System, we can effectively increase the capacity of the IDS, or, provide a means for traffic to be tunnelled for inspection by the IDS without requiring bandwidth equivalent to the full throughput of the network.

4.4.1.1 P4ID Fundamentals

Intrusion Detection Systems have typically been designed and implemented for traditional networks where they can either process traffic in-line (Intrusion Prevention System), or, receiving a mirrored copy of traffic, generating alerts that can then be actioned by a monitoring system or network administrator. However, placing these systems effectively requires an understanding of the underlying network infrastructure. This underlying architecture can differ from the architecture presented through an abstracted software-defined network.

Having explored the capability of P4 platforms and implementations of middleboxes in Chapter 2, we can summarise that there are limitations to the programmability and capability offered within the P4 language. P4 is designed around the notion of 'match-action' pipelines, representative of those in hardware switches [11]. With this constraint, parsing variable-length headers is limited in current P4 platforms [103]. Further to this, IDS signatures typically describe patterns that should be matched, and regular expression matching, a common approach used by IDS running on commodity hardware. Regular expressions have limited support in P4 [62].

Similar approaches in SDN that have been taken have typically involved the controller making traffic forwarding decisions, either forwarding large volumes of traffic to the IDS, relying on initially configured flow rules, such as a VLAN identifier to forward traffic to the data plane [78], or only taking action to actively block or re-direct flows when an Intrusion Detection System has alerted on traffic. Tavares et al. [146] base their approach on a combination of compiled rules for 'high-priority' alerts, along with sketch-based counting, to send the first N packets of a flow to the IDS. They however, do not use the IDS to provide any feedback back to switches, and their sketch-based approach relies on *"a predefined set of operations corresponding to the size of the counting bloom filter that composes the count-min sketch of depth d times the width w"*.

Our approach with P4ID is instead to allow the data plane to act as a pre-filter for traffic reaching an IDS. When an alert is made by the IDS, the *P4ID Monitoring Agent* will parse the logging output associated with the alert, and generate a flow rule. This flow rule will either continue to forward traffic to the Intrusion Detection System, or block traffic entirely. This response can be configured by a network administrator based on their requirements and the rule sets that they are using with the system.

4.4.1.2 P4ID Data Plane

The P4ID data plane uses the abstract pipeline, as introduced previously, in combination with additional stateful stages. The data plane tracks new flows, and if a flow has not been seen before, then a configurable number of packets are sent to the IDS. When the threshold of packets to send has been exceeded, the switch allows traffic to be processed through the switch unimpeded. Any negative feedback, where traffic should continue to be forwarded to the IDS, or dropped by the switch is produced by the control plane. When flows have exceeded the threshold, there is also a timeout value stored in the switch. When this timeout value is exceeded, the flow is again forwarded via the IDS until the packet threshold is exceeded. This mitigates against flow collisions, by limiting the period in which traffic is likely to be allowed to bypass the IDS, whilst making up part of a new flow.

Flow collisions can be mitigated against further by parsing the TCP header, and always resetting the packet threshold if a packet has the SYN flag set, which signifies the start of a connection. However, this eviction strategy would only be effective for TCP traffic.

When a packet arrives at the switch, and has passed through the parser stages into the ingress pipeline, this is the stage at which P4Protect begins. The first step is to calculate a flow identifier, which is used to uniquely identify flows, and as a register index. To create this hash, we use a CRC32 hash of the input data, truncated to 18 bits in length. This gives 262144 possible entries in stateful memory at any given point on the BMV2 platform. Whilst increasing this further could reduce collisions, it would also double the stateful memory needed.

When the flow has been hashed, we then use that with two registers. The first register stores the timestamp at which the last packet associated with the flow hash was seen. Each time a new packet in that flow is seen, this value is updated. This timestamp value can be compared to the previous timestamp value to give the time elapsed. Once this value has been read from the register, the next stage is to either reset the packet counter to zero if the threshold has been exceeded, or, read the packet threshold, update it and write it back into the register. Values are stored into metadata fields to identify whether either threshold has been exceeded.

Threshold values can be programmed from the control-plane, and themselves read into metadata fields to be used for comparison. By using the control-plane, rather than hard-coding the threshold values into memory, the data plane application can be made more flexible, and can respond based on network demands for accurate detection vs traffic loads to the IDS.

The pipeline then determines whether traffic should be allowed to bypass the IDS, based on whether the packet count value has been exceeded. If it has, then traffic is processed as normal, otherwise, the traffic is forwarded or mirrored to the IDS by taking the appropriate action.

Taking this stateful approach overcomes the limitations that can present themselves when trying to describe Intrusion Detection signatures for P4 data planes. P4 data planes cannot typically parse the application layer of a packet, with limited support for variable length packet headers. Therefore, trying to implement IDS signatures as table entries could prove ineffective. This is further compounded by the most common protocols using assigned ports, a 5-tuple is not sufficient to classify traffic for Intrusion Detection compared to a signature searching for a specific sequence in a packet payload. Our approach also offers a simpler mechanism for clearing individual flows after a timeout, whereas P4-ONIDS [146] can only clear the entire filter.

When implementing the pipeline design, we made a number of design decisions to reduce resource usage and the overall impact of our approach. Firstly, the number of register entries we use is capped to 262144 entries, based on an 18-bit wide hash.

Whilst there is a large potential number of hash collisions, our evaluation in Chapter 5 shows that we can still maintain an acceptable level of detection with the system in this configuration. When we hash flows, we take 104 bits of data, or, 2^{104} possible combinations (Source and Destination addresses, ports and protocol identifier), which are then hashed to the 18-bit identifier used within the data plane. This gives 2^{86} possible combinations where a given hash input would equate to the same hash output.

This flow collision would also have to occur within the timeout period, otherwise the flow would be directed towards the IDS for monitoring anyway. Therefore, we put forth that the saving in switch memory is an acceptable trade-off compared to the potential for collisions.

In total, the registers used by P4ID would hypothetically occupy 262144 48-bit values, used for capturing timestamps, and packet counts can use either 8 or 16 bit registers, depending on requirements. This therefore requires 1.5 MB of memory used for timestamps, along with a further 0.26-0.52 MB being used in the implementation for the BMV2 software switch. Note that this can be reduced further by reducing the accuracy of timestamps, or the number of flows.

In our implementation for the Tofino platform for example, we reduce timestamp memory occupation by 50%, by truncating a 48-bit nanosecond timeout value to a 32-bit value instead. With this reduction, we can either offer a high-resolution timeout, but with a maximum value of only 4295 milliseconds (the maximum value a 32-bit register in nanoseconds can hold). The alternative is to reduce accuracy by discarding the lower 16 bits of the value, which equates to 0.6ms.

4.4.1.3 P4ID Monitoring Agent

The P4ID Monitoring Agent is responsible for instantiating the P4ID pipeline and monitoring the output from the IDS itself. The application is deployed as an agent running on the host running the Intrusion Detection System. The agent on start-up searches for the logging output file as specified in the *agent configuration* that is supplied on start-up. Our agent does not rely on any packet-in messages being forwarded from the control plane, so is suited to operate as a local or remote agent depending on where the IDS is located.

When the pipeline has been installed, the agent opens a connection to monitor the logging output from the IDS by following the 'Event' output file, which produces alerts in a JSON format. The logging format gives a message with the alert ID, the message associated with the signature that has been detected, and the 5-tuple of the flow that has been alerted on. This flow is parsed into the constituent source and destination ports and addresses, and a *Flow Request* is generated for that flow.

The monitoring agent highlights the simplicity and ease-of-development offered by the framework. It is able to take advantage of Golang's native support for JSON parsing into objects, in conjunction with an external library to monitor file updates. Flow requests based on these alerts are simply built using helper methods and passed into the control layer.

4.4.2 P4Protect

P4Protect is an example application designed to offer data plane-based protection to SDN controllers and other appliances running on the network, with no modification to them required. P4Protect is designed such that it can be combined seamlessly with other applications running with the 4MIDable framework.

We demonstrate this approach in use with an off-the-shelf SDN controller, inserting the P4Protect logic into the pipeline and highlight how it prevents an attack from being successful at overloading the controller. We focus here on the deployment of P4Protect as a self-contained module, requiring no modifications to be made to the appliance or controller being protected.

4.4.2.1 P4Protect Motivations

Software Defined Networks, especially in OpenFlow architectures are centred around the concept of a centralised flexible control plane orchestrating multiple devices. The controller processes traffic when a switch does not have the appropriate flow rule, or network configuration determines that the controller should see the incoming packet. The latter is often used as part of a 'learning' process, to allow the controller to discover the surrounding network.

However, the capability of the controller to process requests such as these is finite. Switches have a limited bandwidth to send data to the control plane, both in OpenFlow and P4-based architectures [28]. This bandwidth is typically a fraction of the overall capacity of the switch, being treated as a management interface, rather than designed for large volumes of traffic. Controllers also have finite resources, work [87] has shown that even a self-described 'carrier-grade' SDN controller can become overwhelmed with rates as low as hundreds of packets per second. Our own evaluation discussed in Section 5.3.1, performed against the same controller, shows that with a rate equivalent to only 84 kilobytes a second is enough to overwhelm the controller. When the controller is overwhelmed, we see that resource utilisation spikes, and it is no longer able to reliably service PacketIn messages arriving from the data plane.

We have designed the P4Protect module to be standalone, and implemented entirely in the data plane. This takes advantage of the flexibility offered by P4, and facilitates integration, by not requiring any modification to the control-plane or appliances being protected. This has another benefit, in that, when under attack, the control-plane is likely to be already struggling to serve all requests. If the data plane is required to communicate with the control-plane to initiate the protection, the messages from the data plane are liable to suffer the same packet loss that the attack is causing for all other traffic.

To counter these problems, we propose that the data plane should monitor the number of connections being made to the control plane, tracking it on an overall or per-port basis. Tracking all requests being directed towards the data plane would serve to identify attackers across multiple ports more readily, but would effectively limit all users connected to the switch under attack. Equally, tracking requests on a per-port basis reduces the likelihood of false positives, but could allow a motivated attacker across all ports to successfully bypass the protection that is in use.

P4Protect can also be deployed with control plane integration, where the Control Plane can control and manage P4Protect's behaviour. This could be used to implement protection when needed for other network appliances, or, based on controller conditions, could be used to dynamically adjust thresholds at which P4Protect's mitigation starts to drop traffic.

4.4.2.2 P4Protect Data Plane

The P4Protect data plane is distributed as an independent P4_{16} *control*, provided for both the BMV2 and Intel Tofino platforms. Distributing P4Protect as a self-contained control makes it more straightforward to deploy, and can be as simple as calling the control in the appropriate stage of a pipeline. By default, P4Protect only uses platform-defined fields and primitives. This means integration is not dependent on header support offered by an individual pipeline application or any naming conventions.

Fundamental to the operation of P4Protect, we focus on the number of requests going to a specified egress port, and track them by the ports that they are originating from. This helps to limit the memory required by P4Protect, by limiting the maximum number of entries to the number of ports available on the switch. This can be limited further, by instead tracking the overall number of requests being directed to a particular port. In this case, we only need a single entry in stateful memory to track the values.

We also support a control-plane involved mode, where either a single appliance can be rate-limited on a per-port basis, or, multiple appliances, up to the maximum number of ports available on the switch are monitored. This uses the same registers that would otherwise be used for per-port monitoring, and adds flexibility whilst maintaining a hard limit on the total resource utilisation of P4Protect.

When a request is directed towards a monitored port, the control calls the register-based tracking action. A monitored port is typically the CPU port that is used to send traffic to and from the control plane, but can also be associated with a particular network appliance. For a given timeout period t, the number of packets p directed towards the IDS is monitored. Each time a packet is steered towards the control-plane, we count the number of packets seen in the timeout period. This approach is not optimal, when compared to using meters, which are implemented in P4 as two-rate three colour markers [60]. However, using meters requires the control plane to be able to program them with the appropriate threshold. This conflicts with our key design requirement of avoiding modification of the control plane to facilitate integration.

However, in the mode where the control-plane has been modified to introduce P4Protect support, we can instead use meters to enable the functionality. Using meters gives us more flexibility in terms of how we manage them, and the capability that is offered. This capability comes in the form of indirect and direct meters, which allow us to not only rate limit based on source ports, but also, when attached to a table, allow us to rate-limit multiple destination ports at once, through the use of Indirect or Indexed meters. Indexed meters offer an array of meters that operate similarly to registers. A network administrator can vary the number of meter entries to suit their particular use case, for example, to introduce port based rate limiting in the data plane, using meters rather than the autonomous register-based method discussed above.

When the packet has been validated against the registers or meters, the packet is either allowed to pass unimpeded or marked to be dropped as the rate has been exceeded. In either case, traffic will be forwarded again once the rate falls below the threshold. In a port-based throttling approach, other ports should only experience a minor interruption in traffic forwarding whilst the protection engages. The configured threshold for packet rates influences this response. If the threshold is too low, the protection can engage erroneously, limiting traffic reaching the controller, but minimising any potential on other ports. If the threshold is too high, the controller will be sent into a temporary overload condition. In this case, there will be an interruption to other ports whilst the controller recovers.

4.5 Summary

In this chapter, we have presented a realisation of the design described in Chapter 3, the 4MIDable framework. With this framework, we enable the building and deployment of individual *Monitoring Agents*, each tailored to a specific task. These monitoring agents provide feedback from a variety of sources, and can be customised to suit a given middle-box, host-based application etc. With our implementation, we consider the overheads of interfaces, and the flexibility that remote interfaces offer. We provide a series of design guidelines for data plane applications, as well as a reference pipeline. Our reference pipeline is designed to provide compatibility with existing SDN frameworks, and we provide an interface for this integration. We propose that this integration capability promotes adoption of the framework in conjunction with existing networks and architectures.

Our implementation is intentionally agnostic where possible, allowing the integration of a series of pipelines, relying on the network administrator to divide resources between features, dependent on network requirements. This re-allocation allows monitoring applications to co-exist.

With this framework, we have presented two applications, P4ID, targeting the offloading of packet processing from Intrusion Detection systems, and P4Protect which provides a mechanism for the data plane to autonomously protect the control-plane from overload conditions. The design of both of these applications is based on the capability and flexibility offered by the framework.

Chapter 5

Evaluation

In this chapter, we evaluate the design and implementation of the 4MIDable framework, as introduced in Chapters 3 and 4, through two key use-cases and applications of the framework. The framework is designed to facilitate the implementation and deployment of monitoring applications within P4 networks. We evaluate this through two realisations of these monitoring applications.

We first evaluate P4ID: P4 Enhanced Intrusion Detection, which focuses on reducing the volume of traffic reaching an Intrusion Detection System whilst maintaining full detection of network threats.

We then evaluate P4Protect, which brings autonomous decision-making into switches. This decision-making is used to be able to detect and mitigate against denial-of-service attacks, specifically those targeting underlying SDN infrastructure. We show how this effectively remediates against attacks in real-time, allowing the infrastructure to recover and resume normal operation.

5.1 Framework Comparison

System Name	Platform	Extensible	Appliance Feedback	Open Source	Control Plane Independence
4MIDable	P4 (BMV2/ Tofino)	Yes	Yes	Yes	Yes
P4-ONIDS [146]	P4 (BMV2)	No	No	No	Yes
P4DDPI [1]	P4 (Tofino)	No	No	No	Yes
CIPA [20]	OpenFlow	No	No	No	No
SnortFlow [163]	OpenFlow	No	Yes	No	No
TENNISON [39]	OpenFlow	Yes	Yes	Yes	No
Gray et al. [50]	P4	No	No	No	No
Ndonda et al. [98]	P4	No	Yes	No	No

Table 5.1: Framework Comparison

Table 5.1 compares a number of the frameworks presented in Chapter 2, and categorises them based on platform, whether they are designed to be extensible, whether they can accept feedback from existing security appliances, whether they are open source, and whether they can operate independently of the control plane.

To summarise, there are key differences in terms of platform support, three of the listed platforms [20] [163] [39] only support OpenFlow, and two others only support P4 on BMV2 or the Tofino. 4MIDable supports both BMV2 and the Tofino platforms, and is designed to be extensible to support a broader range of platforms when as they become available.

In terms of extensibility, and appliance feedback, only TENNISON and 4MIDable have support for both extensibility and appliance feedback, however, TENNISON's extensibility is limited to running as part of the controller, and TENNISON relies extensively on *PacketIn* events, which could lead to performance concerns at a broader scale. TENNISON also lacks support for control plane independence, as it is using OpenFlow data planes.

Ndonda et al.'s 'Two Layer IDS' makes extensive use of the control plane, and does not perform any of the attack detection in the data plane. Instead, the P4 data plane simply implements the whitelist managed by the data plane. Gray et al., by comparison, focuses on using P4 only for metadata extraction. This metadata is then fed to a machine-learning based detection engine. By comparison, our P4ID application for the 4MIDable framework implements remediation in the data plane, and the P4Protect application adds data plane based detection of certain types of attack.

5.2 P4ID

P4ID provides a data plane based pre-filter for traffic, determining which traffic should be forwarded to an Intrusion Detection System (IDS). The IDS then provides feedback to the data plane, via the 4MIDable framework. This feedback is used to ensure that malicious flows continue to be monitored by the IDS.

We evaluate P4ID based on the reduction in traffic reaching the IDS, and the impact, if any, that this has on detection. With P4ID, we aim to reduce the volume of traffic reaching a given IDS, whilst maintaining a level of detection equivalent to forwarding all traffic to the IDS.

5.2.1 P4ID Motivation and Aims

With P4ID, we use the data plane's ability to store local state, to determine which traffic should be forwarded to an IDS. This determination is made based on how many packets of a flow have already been inspected by the IDS, the time elapsed since the traffic was last inspected by the IDS and whether the IDS has generated an alert for that flow. P4ID relies on the principle that an IDS will generate an alert based a packet or sequence of packets received near the start of a flow. When an alert is generated by the IDS, a *monitoring agent*, running as part of the 4MIDable framework detects this. The agent then generates a table entry to be installed into the switch. This table entry describes the 5-tuple flow supplied by the alert.

We therefore can reduce the volume of traffic being processed by the IDS, whilst detecting signatures and alerts equivalent to when all network traffic is being forwarded to the Intrusion Detection System. We show this as being effective with a range of attack types, including Denial-of-Service, Web Application Attacks, Brute-force attacks and botnet traffic.

5.2.2 P4ID Evaluation Architecture

Our evaluation architecture is focused on a P4-programmable switch operating in conjunction with an Intrusion Detection System. Traffic is sent to the switch, representing either benign behaviour, or benign behaviour mixed with traffic associated with a network attack. The switch processes these packets, and forwards them either to the IDS, or to a bypass collector. The IDS provides feedback of detections in the form of alerts. When an alert is generated, the 4MIDable framework is notified, and an action is taken. In our evaluations, the action taken is to generate and install a rule to ensure that traffic continues to be forwarded to the IDS if it is associated with the alerted flow. The bypass collector captures all traffic that is not being forwarded to the IDS. We use this to validate that the switch is correctly processing all traffic.

We present results based on both hardware and software platforms offering support for the P4 language. In software, we use the open-source P4 Behavioural Model [103]. BMV2 is a software switch that serves as a reference implementation of a P4 switch, with performance constraints. We compare this with the Intel Tofino [149] platform, which is a commercially available hardware platform supporting P4. In each case, the switch is managed using a bespoke P4ID application created using the 4MIDable framework, and the framework is connected to the switches using the P4Runtime.

The application monitors the logging output of the intrusion detection system. When an alert is generated by the IDS, the JSON output is parsed by the P4ID application and used to generate a flow request. This flow request specifies a flow 5-tuple associated with the alert. This is then installed as a table entry into the P4 pipeline, to ensure that all traffic from that flow is then forwarded to the IDS.

Figure 5.1 shows the evaluation environment in use, wherein we have traffic arriving to the switch, which is then flagged, or allowed to proceed as 'bypass' traffic. For the BMV2-based environment, the switch is connected to a series of virtual Ethernet (veth) pairs. Traffic is replayed to the switch's input interface using the tcpreplay [152] tool. Traffic is captured from the IDS and Bypass points using the tcpdump tool. We capture all traffic to ensure that the switch has processed all packets successfully.

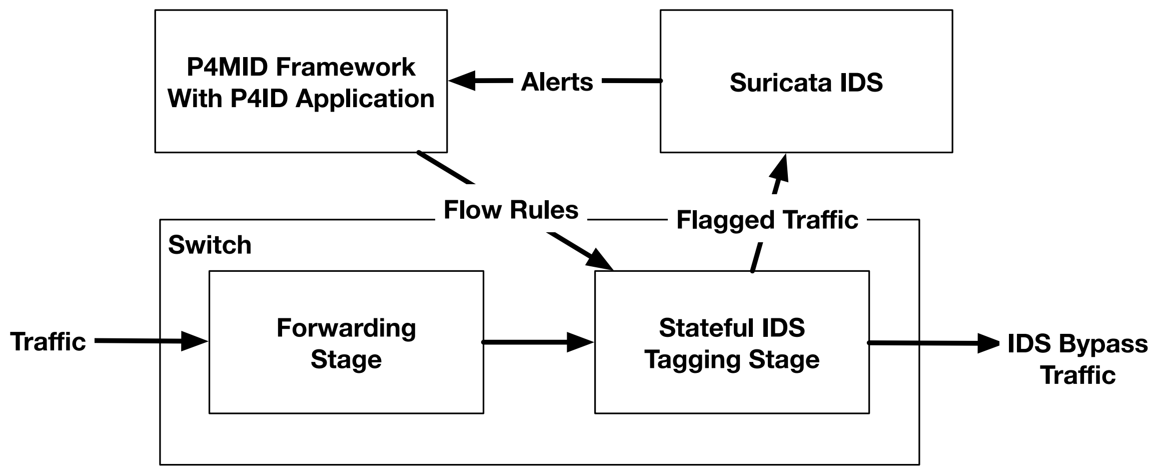


Figure 5.1: P4ID Evaluation Architecture

For the Tofino evaluation, the switch is connected to a series of physical network interfaces connected to the evaluation host. The evaluation host handles packet replay using tcpreplay, traffic capture using tcpdump and runs the Intrusion Detection System and 4MIDable framework.

5.2.3 Metrics

Signature-based IDS represent detections as a combination of *signatures* and *alerts*. Signatures describe characteristics of a packet or series of packets, for example, specific port values associated with a protocol, specific header contents or payload data.

To describe a series of packets, signatures can also define thresholds, which specify the number of times a particular detection has to be made before an alert is generated. These thresholds can help in reducing the number of false positives that are generated by the IDS [148]. An *alert* is the individual event of a signature detection. A single signature can generate many alerts, if the same pattern is being matched multiple times. As a result, a single attack can generate multiple alerts. This is especially common where an attacker sends a series of repeated packets.

We compare our approach to the state-of-the-art with signature-based IDS when applied to the range of attack types presented in our evaluation. Our aim is to provide 100% detection from the Intrusion Detection System whilst still significantly reducing the traffic that it has to process. Our work does not aim to enable signature-based IDS to detect previously undetectable attacks at this stage.

To validate this, we focus on the number of signatures and alerts being generated by the IDS, both with and without the P4ID application in operation, in conjunction with the number of packets, and volume of traffic that is being processed by the IDS compared to that which is allowed to bypass the IDS.

We also compare the impact on detection and traffic quantities when using varying P4ID configurations. The two variables we focus on are the packet count p and flow timeout t. The packet count p describes the number of initial packets of each flow which should be sent to the IDS, a higher value increases the volume of traffic automatically directed to the IDS. The flow timeout t is used as a data plane based eviction strategy for flow entries, and will reset the packet counter when packets from a given flow haven't been seen for t microseconds. A higher value of t will reduce the quantity of traffic that is directed via the IDS, as the counters will reset less periodically. A lower value increases the frequency at which individual flows are sent to the IDS.

5.2.4 Results

We present P4ID evaluated against a range of common attacks, using both synthetic, and publicly available datasets. We begin this section by introducing a stage of functional testing. For this, we replicate attacks in real-time, to ensure that attack traffic is being correctly processed by the IDS.

We then evaluate our platform against datasets representative of real-world attacks, one is generated using the Intrusion Detection Dataset Toolkit [26], and the other is a published dataset for the evaluation of Intrusion Detection Systems [133]. This combination allows us to compare the behaviour of the system with different attack types and identical background traffic, as well as in scenarios representative of more diverse background traffic. We present results for both the P4 'BMV2' software switch, and the Intel Tofino hardware platform.

5.2.4.1 Functional Testing

We validate the correctness of our approach by replicating attacks contained within the datasets in a controlled environment. This allows us to ensure that the pre-filtering is effective without background traffic influencing detection, which is possible, due to the finite size of the flow-tracking registers. In this case, we expect to see that the majority of network traffic will be steered towards the Intrusion Detection System.

For this, we use a Mininet [77] environment, running with an attacking host and a web server. The hosts are connected via an OVS switch, which is providing layer 2 switching between the hosts. The OVS switch is then configured to mirror all traffic to a third port. This configuration is representative of a typical network tap, where a core switch or router can mirror all traffic towards a security appliance.

This third port receives all traffic crossing the switch, which is then sent using a *veth* pair to a P4 BMV2 switch running the P4ID pipeline. The IDS and a collector are attached here, to validate that all traffic has been pushed through the switch, and to measure the volume of traffic forwarded to the IDS, compared to that allowed to bypass the IDS.

We replicate both stateless and stateful attacks in this environment. Stateless attacks require a signature to be matched once before an alert is generated. Stateful attacks require thresholds to be met within a timeframe, making these more susceptible to loss of fidelity. This is as our approach relies on reducing the number of packets from 'benign' flows that are being forwarded to the IDS.

For both attacks, our vulnerable server is running the Damn Vulnerable Web App (DVWA) [161], which consists of an XAMPP stack, running the Apache web-server and supporting database etc. This provides us with a platform to reproduce real attacks against.

We first focus on an SQL Injection attack, where we perform one instance of the attack, and only a single packet is required to be detected by the Intrusion Detection System to trigger an alert.

For the stateless attack, we perform an SQL Injection attack, where the malicious attacker sends an SQL command to the server as part of the HTTP URI. The EmergingThreats rule set includes a number of rules which search for these keywords. In the case presented here, the command injected contains the keywords *UNION SELECT*, as shown in Figure 5.2, which is then detected by the IDS and the appropriate alert generated.

```
alert http $EXTERNAL_NET any -> $HTTP_SERVERS any
(msg:"ET WEB_SERVER Possible SQL Injection Attempt UNION SELECT";
flow:established,to_server; http.uri; content:"UNION";
nocase; content:"SELECT"; nocase; distance:0;
reference:url,doc.emergingthreats.net/2006446;
classtype:web-application-attack; sid:2006446; rev:14;
metadata:affected_product Web_Server_Applications, attack_target Web_Server,
created_at 2010_07_30, deployment Datacenter, signature_severity Major,
tag SQL_Injection, updated_at 2020_09_01;)
```

Figure 5.2: SQL Injection Detection Rule

For the stateful attack, we use the SlowHTTPTest [134] tool, which overloads a web server by opening and maintaining numerous connections to a web server. Web servers have a finite pool of connections that can be used at any one time, and the tool aims to exhaust this pool. The tool keeps these connections open by periodically sending invalid header options, which the server is unable to process.

The rule to detect this attack focuses on the referrer header that is sent by the attacking tool, *slowhttptest*, and the signature defines fifteen occurrences within a thirty-second window before an alert is generated.

In both cases, the appropriate alert is generated by the IDS when the behaviour associated with the attack has been detected.

For both types of attack, we show the traffic distribution, which contrasts significantly with the results shown for other data sets. This variation is caused by a lack of benign background traffic, and is representative of P4ID generating and installing the appropriate rules into the switch, to ensure that flows associated with an attack continue to be directed via the IDS.

5.2.4.2 ID2T

ID2T (Intrusion Detection Dataset Toolkit) is a toolkit for injecting synthetic attack traffic into packet captures. This allows us to combine a known baseline packet capture, which contains only benign traffic, with attack traffic representing a number of attack categories. This approach enables us to compare how a switch running the P4ID pipeline will process traffic with each attack category.

To facilitate data collection, our base capture runs over approximately a five-minute period, containing 37,000 packets. Our base capture was captured on an office network running a variety of hosts. The base capture is used to provide consistent benign background traffic between each class of attack.

Table 5.2 shows the attacks included within each of the generated ID2T datasets, and the total number of packets. The benign set is a packet trace with no added traffic, which we use to determine the impact of varying the P4ID configuration. The **Botnet** category contains traffic representative of a peer to peer botnet. The **DDoS** attack replicates traffic associated with the slowloris [139] attack, which relies on holding open connections to a web-server for as long as possible. This exploits a web server's limited number of open connections available at any given time. **EternalBlue** is an exploit most commonly associated with a common ransomware tool that used it [95]. The **Port Scanning** includes traffic representative of the NMAP [86] network scanning tool. NMAP can create many new connections in a short period of time, as it attempts to connect to services on successive port numbers. **Sality** contains traffic associated with a family of malware, which can also act as a peer-to-peer botnet [38]. The final dataset contains **SQL Injection** [55] attacks, which aim to damage, gain access to or infiltrate databases.

Attack Type	**Packets**
Benign	37004
Botnet	37104
DDoS	46118
EternalBlue Exploit	37550
Port Scanning	38006
Sality	37592
SQL Injection	43427

Table 5.2: Packet Count for ID2T Datasets

One of the limitations of this approach is that attacks are synthetically generated for real traffic, such that they are approximations of attack traffic, with matching characteristics, rather than being real attacks. This is most prevalent for the botnet attacks, which cannot be detected in our evaluations. The Distributed Denial of Service attacks are also detected, but identified incorrectly by the IDS as a port-scanning attack. It is important to note that the limitation identified here is in the combination of IDS and rule set rather than being a limitation introduced by our approach.

Figures 5.3 and 5.4 show the percentage of detected signatures and alerts respectively when traffic is being pre-filtered by P4ID. The baseline for detection is 100%, which is when the traffic is processed directly by the IDS. In every case, attack signatures are successfully detected with a reduction in traffic being processed by the IDS in excess of 50% on the BMV2 platform, and a traffic reduction of over 70% on the Barefoot Tofino platform.

With alerts, we anticipate that the number of alerts being generated will be fewer, as the volume of traffic reaching the IDS is reduced. This reduction is particularly prevalent in a number of attacks, such as port scanning and denial-of-service attacks, where low volumes of traffic are automatically whitelisted if a given flow does not generate an alert. Despite the reduction, the attack signatures are still detected successfully.

The Distributed Denial-of-Service attack contained within this dataset is an abstract representation of a number of hosts attempting to connect to a particular host at a high rate. As the behaviour is not associated with a particular tool, the Emerging Threats rule set detects this as representative of a port-scan, based on a number of new connections in a short period of time. This limitation is indicative of a lack of an appropriate IDS signature to search for in the traffic. However, when the traffic is being pre-filtered, the attacker can still be correctly identified by the traffic reaching the IDS.

The EternalBlue attack [12] is representative of a network-based exploit to take control of vulnerable hosts and subsequently install a backdoor. Whilst the number of signatures and alerts generated is lower than the baseline, the attack is still detected in multiple stages as expected. These stages in chronological order are as follows: The vulnerable protocol in use is identified, and associated host. The installation of the DoublePulsar backdoor is then detected, along with the source and target of that attack. The external address of the attacker is also provided in the logs. The EternalBlue attack and response are then both identified by the IDS, along with further traffic associated with the exploitation. The missing signature is a false positive describing a port-scan on the network.

In all other cases, we achieve the same level of signature detection as the baseline, and in the majority of cases, we achieve full detection of alerts. Though, we suggest that when using the P4ID to filter traffic, network administrators should consider whether existing thresholds could be lowered to suit changes in traffic profiles.

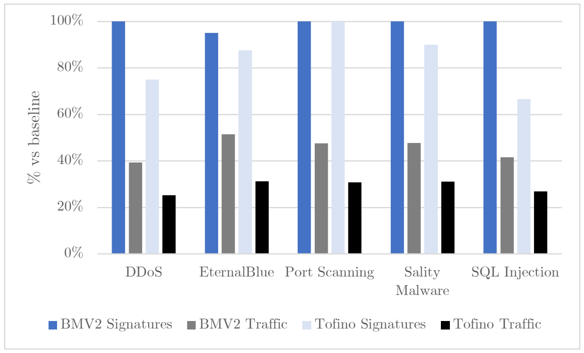


Figure 5.3: ID2T Highest Achievable Signature Detections

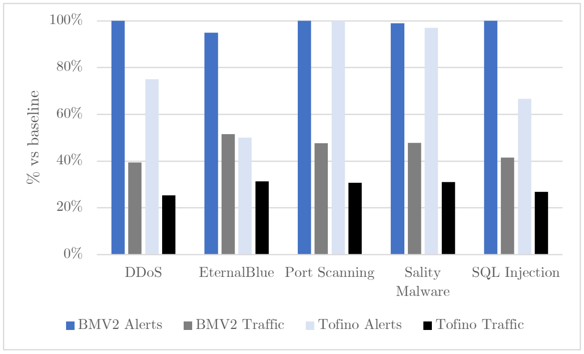


Figure 5.4: ID2T Highest Achievable Alert Detections

5.2.4.3 CICIDS2017

The CICIDS2017 dataset [133] is a contemporary and publicly available dataset, comprised of a number of benign packet traces combined with the attack traffic of a range of attacks. Each packet trace is named by the day of the week, and contains one or more attacks at specified times in the capture file. The attacks in this dataset include botnet, brute force, denial-of-service and web-based attacks. Whereas the ID2T datasets are over a short period of time, each IDS2017 trace is spaced over the course of eight hours, or equivalent to a working day. When replaying attacks, we have divided the dataset into a series of individual captures representing all traffic on the network at the time of the attack, and for a fixed period before

Attack Type	**Packets**	**Duration (s)**
DoS-GoldenEye	361535	899.93
DoS-SlowHTTPTest	374014	1379.76
DoS-SlowLoris	424886	1499.60
FTP Brute-force	6315532	3719.27
HeartBleed	427652	1319.23
Port Scanning	1485066	5699.84
Web Brute-force	1345926	2459.99
Web SQL Injection	51162	239.80
Web XSS (Cross-site Scripting)	233276	1399.19

Table 5.3: Packet Count for IDS2017 Datasets

and after the attack. To do this, we created a pre-processing tool, which consumes a configuration file describing the timestamps of each attack. The tool then divides the packet captures according to this configuration using the editcap [106] tool, producing individual packet captures for each category of attack.

For each dataset, we then prepare appropriate intrusion detection rules for each category of attacks. We use a bespoke rule set for each attack category, using the rules provided from the EmergingThreats [37] rule set. We use publicly available rule sets to demonstrate that our approach is effective, even with off-the-shelf configurations. Minimising the rule set for each attack type ensures that the IDS remains performant, as large rule sets can have a negative impact upon performance [70].

Table 5.3 lists the range of the attacks included within the data set, which are the attacks we use to evaluate our pre-filtering approach. These attacks were selected from the data set as they could be detected by Suricata [101] when it is processing all traffic in conjunction with published intrusion detection rule sets. We use the EmergingThreats [37] rule set as a comprehensive rule set, and we enable the rules included for each class of attack.

The attacks of note that are not included, as they are not detected by the signature-based IDS in our evaluations are: SSH Brute-force, Ares Botnet traffic, Hulk Denial-of-Service tool, Low and High Orbit ION cannon tools. SSH brute-force poses a particular challenge for signature-based IDS, as traffic is encrypted. The IDS only has visibility into the initiation of an SSH session. Whilst the IDS can detect these connection attempts, there is no way in a signature to determine whether the connection failed. Whilst it is possible to set a threshold for a certain number of occurrences in a given time frame, this could lead to false positives and is beyond the scope of this stage of work. The Denial-of-Service tools mentioned randomise their headers, specifically to evade detection from static signatures. This again limits the capability in detecting certain types of denial-of-service attacks.

The CIC-IDS2018 dataset, whilst more recent, is generated using the same tools and methodology as the IDS2017 dataset, and offers the same attacks as we focus on from the IDS2017 dataset. However, the network topology is significantly more diverse, and presents flows captured at each host, rather than from a central network tap. Our approach is designed around the integration with a network tap. Therefore, to align the dataset with our proposed architecture, we would have to perform data processing to collate and merge the datasets which introduces scope for error in terms of duplicated traffic.

Figures 5.5 and 5.6 highlight IDS detections when using the IDS2017 dataset in conjunction with the P4ID platform.

In general, we see that the Barefoot Tofino platform offers the greatest reduction in traffic, with an average reduction of 67% across all tested datasets. With this reduction, we continue to detect 97% of attack signatures. BMV2 on the other hand, offers a decrease in traffic of 70%, yet signature detection falls to only 91%. Again, the baseline is 100% of alerts and signatures when all traffic is being directed towards the IDS.

The Web Brute-force attacks are one case where not all signatures are detected. In this case, the signature that isn't detected is a false positive associated with an application that isn't running on the network. The HeartBleed attack is another case where not all signatures are detected, when running on the BMV2 platform. The alert that fails to be detected pertains to a vulnerable client receiving malicious data from a server. The Tofino platform successfully detects this attack, but, the Tofino implementation re-directs 50% of traffic to the IDS for this dataset, compared to BMV2 redirecting only 31% of traffic.

It is important to note, that, for some kinds of attack, signature-based IDS, and therefore P4ID is not the most effective solution. This is best exemplified by a number of attacks in the IDS2017 dataset, including SSH Brute-force, certain types of DoS attack, and scanning attacks. In the case of the SSH attacks, SSH is an encrypted protocol, therefore the IDS only has insight into whether a connection has been made on a specific port, and the handshake that occurs at the start of the session. This could be mitigated against by setting a rate limit for connections to a given host, but this could equally introduce false positives into the system.

A number of DoS attacks in the dataset, including the Hulk [68] attack are not suited to detection via signature-based Intrusion Detection. In the case of Hulk, the packet headers are randomised with each transmission to evade signature-based detection. The Low-Orbit and High-Orbit ION cannon traces included with the IDS2017 dataset suffer from similar challenges.

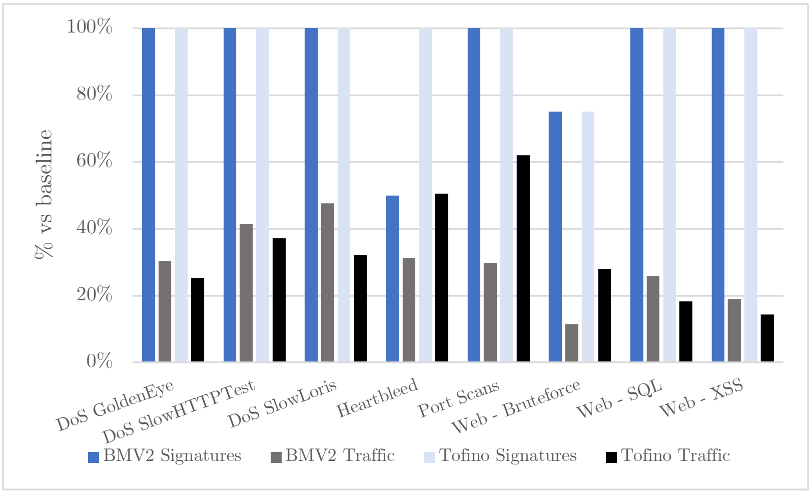


Figure 5.5: IDS2017: Signature Detection

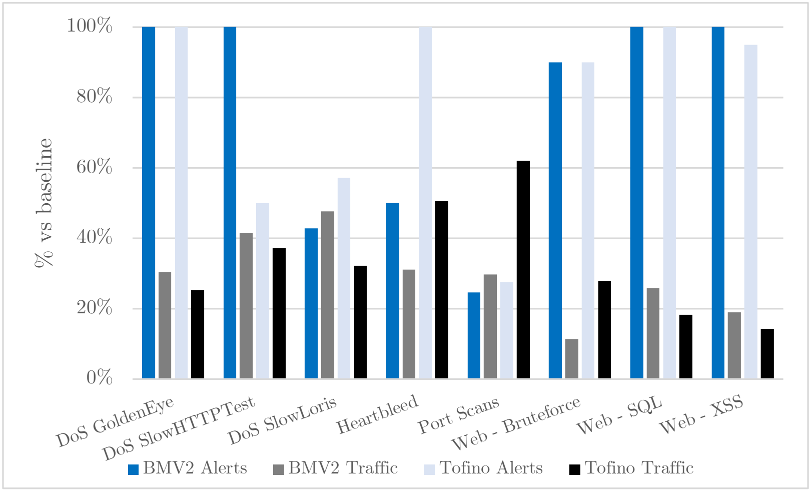


Figure 5.6: IDS2017: Alert Detection

5.2.4.4 CPU Utilisation under attack

To evaluate the performance benefits of P4ID, in terms of load reduction, we also measured the CPU utilisation of the IDS, whilst under attack. We first measured the average utilisation when all traffic is being sent via the IDS, to establish a baseline figure. This baseline figure can then be compared to when P4ID is deployed against a BMV2 switch and the Suricata [101] IDS. With this, we would expect to see a reduction in average CPU utilisation, as the volume of traffic being processed by the IDS is reduced significantly.

The high-load scenario uses the benign dataset from the IDS2017 dataset, replayed at a high rate (450 Mbps), alongside the malicious attack dataset being replayed in real time. The malicious dataset is replayed in real-time, to allow IDS rules which rely on timed thresholds to function correctly. The malicious datasets have a peak rate of 16 Mbps. The increase in traffic volume is essential in order to match a more realistic traffic found in production networks.

We execute both our baseline and P4ID IDS configurations and record in each run the average CPU utilisation for the duration of the experiments when running with BMV2. Table 5.4 shows the different in average CPU utilisation for each attack dataset between the two experiments. From the results, we highlight that the CPU utilisation of the P4ID configuration reduces the IDS load by an average of 31.9%, and in all cases, the IDS continues to detect all attack signatures correctly. It is worth highlighting that any load reduction depends significantly on traffic characteristics and traffic rates. For example, a network carrying long-running high-volume flows, can achieve better results. The employed dataset consists of mixed workloads, reflecting a mix of applications typically found in a production enterprise environment.

Dataset	CPU Average Delta	Dataset	CPU Average Delta
DoS GoldenEye	-26.4%	Web - Bruteforce	-25.8%
DoS-SlowHTTPTest	-33.4%	Web - SQL	-28.4%
DoS-Slowloris	-25.3%	Web - XSS	-25.3%
HeartBleed	-28.6%	Port Scans	-34.5%

Table 5.4: CPU Reduction for Suricata with offloading

5.2.5 P4ID Comparison

CIPA [20], is one of the first attempts to develop a distributed IDS using SDN technologies. The platform uses OpenFlow rules to apply forwarding decisions regarding malicious traffic in the data plane, and intrusion detection is underpinned by a distributed neural network implemented in the control plane. By comparison, our approach focuses on using existing tools, in this case, Suricata.

SnortFlow [163] proposes a means to integrate Snort-based intrusion detection into a cloud environment by running it as part of the hypervisor, using OpenFlow to provide network re-configurability; though, when their system is under high load, the detection agent (Snort) will start dropping packets. Whilst our approach also uses an IDS, the choice of IDS is flexible, and does not need to run as part of the hypervisor, and can run on bare-metal switches.

TENNISON [39] implements a more pragmatic approach to IDS, using multi-stage monitoring that filters only traffic determined to be of interest to the network administrator to a software IDS in order to apply DPI processing. However, unlike our approach, TENNISON relies heavily on the control plane to determine which traffic should be forwarded to the IDS, due to OpenFlow constraints, whereas P4ID relies on the data plane as much as possible.

The HEX Switch [114] introduces an OpenFlow switch architecture, allowing users to develop protocol extensions – including IDS features – as part of the forwarding logic. However, this requires specific hardware platforms and support the additional OpenFlow security actions.

Emerging technologies, such as P4, have overcome packet inspection limitations with the introduction of further programmability with significantly richer capabilities [149] [63] [57]. Achieving DPI with P4 remains challenging, as the focus of the language is on fixed-length bit-fields. While the language [111] does support variable-length header fields, the support of hardware devices, especially ASICs to match based on these fields, is limited. Ndonda et al. [98] describe intrusion detection for Industrial Control Systems with white- or blacklisting functionality for a specific protocol using P4. However, the work relies heavily on the control plane for flow processing. By contrast, in our approach, the only control-plane based processing is to install rules into switches once the IDS has made a detection. This will then continue to forward malicious traffic to the IDS or block it.

Gray et al. [50] use P4 switches to extract features for ML processing, achieving traffic support up to 100Gbps and improving detection rates in comparison to a Suricata instance. In contrast to our work, where we focus on the existing IDS to perform the detection role, rather than using P4 to extract metadata features. They then compare their results to the performance of Suricata, whereas we focus on the integration of existing appliances.

Dao et al. [29] implement a neural network model in the P4 data plane, using a BMV2-based switch. The authors use this model to explore the impact of different model parameters on applying classifications on incoming traffic.

P4DDPI [1] proposes an Intrusion Detection System for P4 to detect malicious domain names in DNS requests, and the authors show that it is capable of achieving line-rate on the Tofino platform. To achieve the full parsing of DNS headers, the authors rely on a process of recirculation, which can incur performance penalties.

P4-ONIDS [146] explores a similar approach with respect to IDS offloading as 4MIDable. The system allows installation of IDS rules into the data plane, by developing a compiler for IDS rule sets into 5-tuple matches, as well as, filtering the first N packets for each flow. The system uses a count-min sketch algorithm, for efficient lookup memory use, though which incurs slow reconfiguration times.

P4-ONIDS shares the closest similarity with our work, however, they rely partly on a process of compiling existing Snort rules into 5-tuple rules to be installed into switches. However, the majority of traffic uses well-known ports, so the rule-based approach can be overly selective. The authors acknowledge this, and propose their count-min sketch based approach. Their approach uses less memory than P4ID, but, is not demonstrated in hardware switches. Ours is evaluated both on the software-based BMV2 and hardware Tofino platforms. In our approach, we also use the detection output of the IDS to steer ongoing forwarding decisions.

5.3 P4Protect

In Chapters 4 and 5, we introduced and described the architecture of P4Protect. P4Protect offers an autonomous data plane-based protection mechanism primarily focused upon protecting SDN infrastructure from Denial-of-Service attacks. These attacks can not only overload a switch's control plane channel, but our evaluations show that the controller can stop responding to network traffic and requests entirely for the duration of an attack.

P4Protect can be implemented for any control plane or other network appliance, optionally with no modification to the controller at all. This modification-free approach allows it to be integrated with off-the-shelf controllers and applied to a broader range of pipelines.

We first introduce a simple class of attack, Address Resolution Protocol (ARP) flooding, which uses spoofed responses to trigger a learning process in the controller. This spoofing can be used to overload a controller with as few as thousands of packets per second. We then present our evaluation architecture, running a contemporary 'Carrier-Grade' SDN controller, ONOS [5] in conjunction with both OpenVSwitch [118], a high-performance, OpenFlow-compatible switch, and the P4 BMV2 platform used in the previous section. We demonstrate that the attack can be successful against either platform, with the same outcome of causing the controller to stop servicing requests in a timely fashion.

We then show how, using P4Protect, with no modification or additional applications loaded onto the controller, will successfully mitigate against this class of attack, throttling the rate at which malicious messages can be sent to the controller, protecting the controller from sustained overload. This is the mechanism that uses data plane *registers*, rather than *meters*. A meter-based approach is also supported by P4Protect, which can be used to supply feedback to the control plane, but, this approach cannot be implemented without modification being made to the control plane.

5.3.1 ARP Flooding attacks on an SDN Controller

SDN Controllers communicate with switches through a link common referred to as the *control plane interface*. The control plane interface is used to pass events between the switch and control plane. This link is used for the controller to be able to process traffic and manage the data plane. In some cases, this link may also be used to offer network monitoring capability by the controller. The bandwidth available for this interface is typically a fraction of overall forwarding capacity [28] in OpenFlow switches, and is reliant on Remote Procedure Call encapsulation in P4 [110].

One way in which this link is used is to facilitate the implementation of reactive forwarding applications, alerting the control plane when packets are received and enabling it to make forwarding decisions and initiate flow setup. Flow setup may involve emitting the packet and installing a flow rule to allow a given flow to subsequently bypass the controller.

Reactive forwarding is used to install flow entries into switches, after they have started to be sent to the network. Reactive forwarding enables the controller to respond dynamically, installing new rules when a host begins sending traffic. Part of the behaviour involved in this process is the sending of false ARP responses. A host can request the hardware address of a given IP address by sending a broadcast message to the network. This broadcast message will be responded to, either by the SDN controller, or other host, with a mapping of the MAC and IP addresses of that host. When running in Reactive Forwarding mode, the controller will also store this mapping, adding the hosts to an internal mapping of 'known' hosts.

This behaviour can be exploited by fabricating ARP replies. To fabricate an ARP reply, an attacker simply needs to randomly generate source and destination MAC and IP address combinations and send an ARP reply to the network. This will automatically get steered towards the controller for processing when the reactive forwarding application is enabled. The impact of generating a series of messages to the controller that then need to be processed can be effective in overloading a controller. Macias et al. [87] find that 1218 packets a second are sufficient to disrupt the normal behaviour of a controller. We find that by sending 3000 packets, equivalent to only

84 kilobytes per second of traffic, this is sufficient to allow a single host to disrupt normal processing by the ONOS controller.

5.3.2 P4Protect Evaluation Architecture

We use the ONOS SDN controller, as representative of a contemporary and full-featured controller, supporting a number of protocols and platforms. Using an off-the-shelf controller allows us to highlight the susceptibility of available platforms to this type of attack. The ONOS Controller is self-described [102] as being designed for 'high-performance' and suited to the scale of large network operators, with 'scalability' and 'resiliency'.

When deploying the ONOS controller, we use ONOS version 2.2.3 Sparrow, which at the time of writing is a Long-Term Supported version of the controller. LTS versions are described by the project as the 'stable' versions with 24-months of bug-fixes and support.

We deploy the ONOS controller in a Docker [93] container. Docker is a containerisation technology which offers us benefits in terms of reproducible environments, constrainable resources and resource monitoring. Docker allows us to constrain the application to a given number of CPU cores and amount of memory with CPU pinning, which ensures that the application is assigned two physical, rather than logical cores.

We deploy the ONOS application running a minimal configuration, the only applications enabled are GUI support, LLDP (Link Layer Discovery), the Host Provider, drivers for OpenFlow and P4, the P4 'basic' pipeline configuration, and the Reactive Forwarding application. This subset of applications supports enables topology discovery and forwarding to take place on the network, whilst not potentially wasting resources on unneeded applications. To add our P4Protect functionality, we have modified the P4 source application and resultant binary that is loaded by the controller when the pipeline is installed, but, we have made no material changes to the controller itself.

To enable us to measure the impact on the controller's capacity for packet processing, we have created a rudimentary ONOS application, which listens for PacketIns carrying a unique identifier. If that identifier is detected, the packet is sent back to the port it originated from. The application consists of an activate method, which registers the application. We then register a *Packet Request*, where PacketIns that match a specific Ethernet Type and Destination combination are processed by the application. The *PacketProcessor* method is designed to not make any requests from any other services in the system, to eliminate any overheads in waiting for them to respond. Instead, we use packet metadata to return it to the ingress port.

We have a client counterpart for this application, which is written in Python. The Python application uses the Scapy [7] library to assemble and send a packet to the listening application. The client sends an Ethernet frame, with the EtherType and destination MAC set to identify it to ONOS, and adds a sequence number in the frame payload. This sequence number is used to identify the response that has been returned for accurate timing measurements. The client records the time at which each packet was sent, and uses this to calculate the round-trip latency. The client then logs the result to disk.

Each experiment is monitored for Docker resource utilisation and the round-trip latency. Docker resource utilisation is reported by the daemon every 0.5 seconds, giving CPU usage as a percentage of available cores. As the only process running within the Docker container, this offers us the overall CPU utilisation of the controller over time. The round trip time is monitored by sending a packet to the latency monitoring application every second. This establishes a baseline rate when the controller is under load, and shows the impact on packet processing when the controller is under attack, by highlighting when packets are actually being lost due to the attack.

For both OpenFlow and P4, we run the switches using a Mininet [77] environment. Mininet is a network emulation tool with support for SDN switches, and allows us to build an experimental topology. For OpenFlow, we use OpenVSwitch [118] which is a production-grade software switch, that operates within the kernel. This represents contemporary, high-performance software switching. For P4, there is not yet a high-performance software architecture. The open-source software switch for P4, known as BMV2 [103] describes itself as being representative of behaviour, but with a series of performance constraints. We have configured our instances of BMV2 with the compilation flags suggested for best performance [43]. For BMV2, we use the Open Networking Lab Docker image [104]. This gives us a preconfigured environment with BMV2 configured for best performance, supporting the P4Runtime and use with ONOS.

5.3.3 Controller Performance Under Attack

We begin by establishing a baseline for how the controller performs when under attack. The attack consists of a single malicious host, generating false ARP replies at a rate of 3000 packets per second. This is equivalent to a traffic rate of 672 kilobits per second. With this, we are looking to identify the rise in CPU utilisation and round-trip latency of PacketIn processing.

We demonstrate this Denial of Service attack in 3 scenarios, unprotected OpenFlow with OpenVSwitch switches, unprotected P4 on BMV2 switches and BMV2 switches with P4Protect enabled.

Figure 5.9 shows the comparative CPU utilisation of the ONOS controller during each scenario. In each case, the Denial of Service attack starts after 30 seconds have elapsed.

We see that as soon as the attack starts, CPU utilisation peaks, consuming all resource available to the host running the controller. For each unprotected scenario, this then settles to approximately 150% of two cores, which our experimentation shows to be consistent with the controller being overwhelmed with requests. When running with a single core, CPU utilisation remains at near full utilisation of that core, suggesting that there is a thread-bound process that is being overwhelmed during the attack.

As introduced previously, we can measure the controller in terms of how quickly it can service a PacketIn message and respond. In our evaluations, we send these messages once per second, so as not to negatively influence the performance of the controller. We see that as the attack is underway, latency climbs sharply from an average of 40ms, peaking at over 960ms. Figure 5.7 does not plot points for packets that are lost. This highlights that for the majority of the attack duration, the controller stops servicing PacketIn requests from the latency monitor entirely.

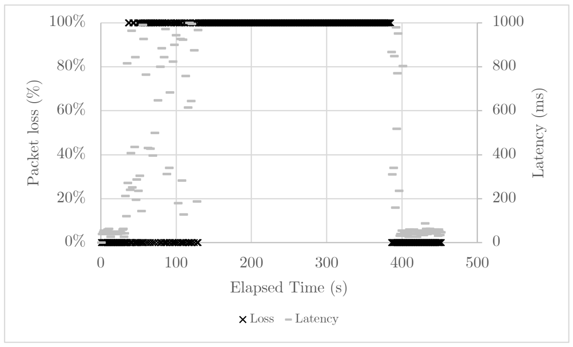
Latency and packet loss again increase significantly for the duration of the attack, with again, an excess of over 85% of PacketIn messages from the latency monitor being lost or dropped for the duration of the attack. The controller is left unable to service the requests, instead reverting to dropping packets.

In each case, the attack runs for 300 seconds, or five minutes, and when the attack ends, we see a difference in behaviour between the two architectures. The OpenVSwitch architecture recovers quickly, with CPU usage falling to normal idle levels as the attack stops. BMV2 on the other hand takes 60 seconds to recover. It is important to note that in each case, the underlying driver managing communication between ONOS and the switch is different, with OpenVSwitch using OpenFlow, and BMV2 using the P4Runtime. Therefore, this additional recovery period could be caused by either the switch or the underlying driver in the ONOS controller.

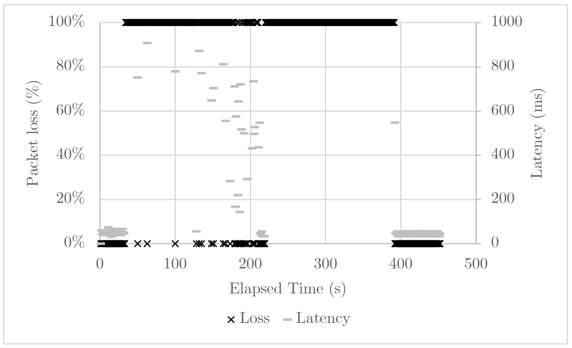
With both platforms we see that the controller can be forced into an unresponsive state for the duration of an attack, and in the case of BMV2, this persists for a time after the attack. This attack influences not only how the controller processes new flows, but will also interfere with normal operation of the network as existing flows time out, or switches periodically send their statistics to the controller. Any processing of traffic that invokes the control plane will be interrupted by traffic volumes of less than one megabit per second.

5.3.4 P4Protect Results

In this section, we present the same experimental scenario, but this time with the P4Protect autonomous protection enabled within the data plane. In this scenario, the attacker will again start generating malicious packets 30 seconds into the experiment. However, unlike in the previous scenarios, the data plane will be counting the number of packets being sent to the controller in a given time window. In our evaluation configuration, we are rate limiting based on the CPU port of the switch, and categorise it by each ingress port. We use the CPU port as this is representative of the link



(a) OpenFlow



(b) P4

Figure 5.7: Unprotected Loss & Latency when under DoS attack

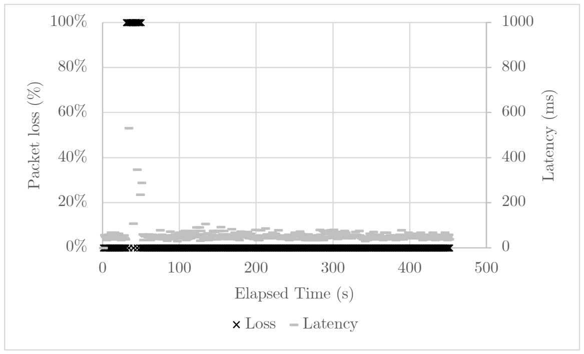


Figure 5.8: Loss and Latency when under DoS attack with P4Protect

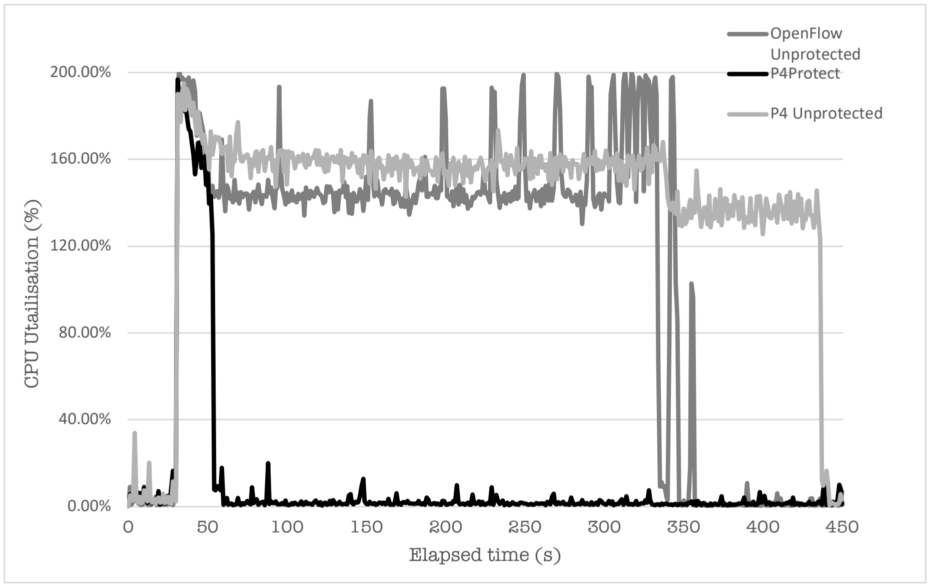


Figure 5.9: Comparison of protected and unprotected CPU utilisation under DoS attack

between control and data plane. If this protection was being applied to a different appliance, this value can be changed. We track traffic to the control plane on a per-port basis. Tracking on a per-port basis will detect attacks coming from an individual port, whilst minimising the impact on other ports on the same switch. Only the attacker's port will be blocked. This therefore requires a stateful register for each port on the switch, with 64 bits of data stored for each port. On BMV2, the maximum theoretical value is 512 ports, therefore, our solution will at most require 32768 bits of stateful memory to implement. This is only equivalent in memory utilisation of 305 5-tuple entries. The 5-tuple is a common flow identifier used for layer 3 forwarding.

We configure P4Protect with a threshold of 10,000 packets per port over a rolling ten-second window that can be sent to the controller before the automated protection engages. We continue to use an attacker sending 3000 packets per second to demonstrate the attack. The value of 3000 packets per second has been shown to be effective in overloading the controller in our evaluation environment.

Figures 5.8 and 5.9 and show the efficacy of P4Protect in mitigating against the attack. The attack begins when 30 seconds of recording has elapsed. Until this point, we see that the CPU utilisation is near idle levels, and response latency remains low, with no packet loss. The attack then commences from a malicious host at 30 seconds. We see that initially, behaviour is the same as that of the unmitigated environment. CPU usage rapidly peaks to 200% before falling to around 150%. Packet processing latency increases significantly, along with packet loss. Note that latency is not plotted for missing packets, gaps in the chart are used to highlight packet loss. However, with P4Protect enabled, the data plane rate limiting engages. Once this has engaged, CPU usage begins to fall, along with packet loss, and processing latency dropping quickly back to normal levels. CPU usage drops back to normal levels within 23 seconds.

With this, we demonstrate that P4Protect can effectively defend against this type of network attack, where a single attacker would otherwise be able to disrupt the controller by sending only 3000 packets per second. Whilst this approach protects the controller from sustained overload, our results show that the controller is prevented from being in normal operation until the threshold is reached. However, this threshold has been designed to be adjustable. If the threshold is too low, then legitimate traffic will be dropped. In this case, normal traffic processing would be disrupted, effectively bottlenecking the capability of the controller. Equally, if this threshold is too high, then the controller will be vulnerable to attack because the threshold to engage protection will not be met.

In the scenario presented, our filtering is based on individual switch ingress ports. If an attacker is attempting to overload the controller from multiple ingress ports, they could have a more disruptive impact upon the controller. This is because, with port-based protection, the threshold has to be met on each port before the protection engages, which is designed to avoid disruption to hosts connected to other switch ports.

To counter this, it is possible to add a second filtering stage, where packets are first filtered by their ingress port, such that no single port can exceed a particular rate of traffic being sent to the controller. This second threshold then ensures that no combination of ports is sending sufficient traffic to overload the controller through a particular switch. However, this approach begins to add further complexity to the P4Protect pipeline, as it would require additional registers to implement. This can impact compatibility with pipelined devices by moving towards using requiring an excessive number of pipeline stages. This is especially pronounced when trying to integrate the functionality with an existing pipeline. Existing pipelines may already be trying to use the majority of the available pipeline resources. In this scenario, meters may be a more appropriate solution. Meters are implemented as platform-specific constructs, and from a programmer perspective require fewer stages. However, as addressed before, the configuration of meters requires the control plane to be able to program their values. With this, we then lose the benefit of being able to apply this protection to platforms without any modification.

The P4Protect pipeline could be also be extended to filter based on types of traffic reaching the switch, allowing, for example, different classes of traffic different rates at which they can reach the controller. This approach would allow the switch to continue to forward ‘priority’ traffic to the controller, whilst offering defence against an attack. However, this would again require additional processing at the data plane level, which could fall out of data plane constraints. With P4Protect, the aim of the application is to avoid invoking the control plane, as, under attack, the control plane may be unable to mitigate against any attack whilst the attack is ongoing.

5.4 Summary

In this chapter, we have evaluated the 4MIDable framework, through the realisation of two applications. With P4ID, we highlighted the benefits of the hybrid approach, showing how the control plane can provide an interface between existing middle-boxes and a programmable data plane. Our evaluations show that we can successfully reduce traffic being processed by an IDS by over 70%, whilst maintaining detection of attacks and network threats.

We have explored the compromises and flexibility of P4ID, offering a configuration that can be adjusted by an administrator to suit network demands. Overall, this shows how we can use the framework to provide an effective off-load to traditional middle-boxes, whilst maintaining their functionality.

Offloading traffic processing in to the network allows functions to be distributed further into the network, without the overhead of passing traffic back to a centralised point.

With P4Protect, we evaluate how a modular application can be inserted as part of the data plane to offer protection without invoking the control plane. In this case, relying on the control plane to mitigate against the attack can prove ineffective if the controller is already in a state of overload.

We began by demonstrating the impact that a denial-of-service attack can have on a commodity SDN controller, causing it to be unable to control the network normally for the duration of the attack. We have shown how, with no modification required to the controller, or the controller's internal logic, a P4-based solution can be used to mitigate against this category of attack. Our solution enables the controller to quickly recover from the attack and resume normal processing.

In Chapter 3, we introduced a series of design requirements for the 4MIDable framework. With P4ID, we demonstrated how we could integrate with the architecture of existing platforms, which was demonstrated through the use of 4MIDable with the Suricata IDS. This also met the requirement of being able to steer traffic towards devices, and, implement monitoring applications using the framework.

P4Protect demonstrates again how we can implement monitoring in the data plane through the use of the 4MIDable framework, offering protection to network devices in the process, and it also highlights how we can integrate with an existing SDN controller, in this case, ONOS.

Chapter 6

Conclusions

With the evolution and drive in networking towards a more capable, diverse network architecture, combined with benefits of centralisation, new challenges emerge for managing this programmability and ensuring the security of computer networks. With ever-growing flexibility, the role of the network also changes, core network infrastructure can be empowered to monitor and protect both itself and other infrastructure. The network becomes more than merely a series of links between hosts.

On the other hand, this empowerment of the network, also brings potential attack and disruption vectors. The underlying infrastructure supporting these can fall victim to misconfiguration or malicious actors. In this thesis, we have highlighted some potential risks associated with the controller to switch interface, and overwhelming this link.

This thesis has examined how emerging architectures and technologies influence the design and implementation of network security approaches. This includes an overall move towards programmability and flexibility, whilst highlighting the divergence that is being forged between different platforms and their respective approaches. We explore the evolution of these technologies, beginning with the programmable control-plane offered by OpenFlow, designed around a fixed-function data plane. We contrast this with P4, where, the data plane itself is a programmable entity. This programmability risks conflicting with the control offered by OpenFlow, the interface to each data plane device is no longer uniform.

We highlight a need for a way to strike a balance between approaches. Centralisation offers an array of benefits, especially in terms of network visibility, and giving controllers context to manage networks more effectively. This centralised context also offers more capability to detect and mitigate against network attacks.

Equally, pushing intelligence and autonomy into network devices distributes processing load, reducing monitoring latency, load on dedicated monitoring devices and allows monitoring functionality to be deployed more pervasively within a network. This autonomy can even be used to afford protection to SDN infrastructure itself, giving the data plane the capability to detect and mitigate against attacks targeting SDN infrastructure.

This drive towards flexibility is underpinned by a fundamental and constantly growing need for network security, networks are the vector for an ever-growing range of attacks. Attacks are growing not only in scale but also in diversity. Emerging types of attack attempt to thwart existing defences.

Given these requirements, we propose the design of a framework and architecture to integrate programmability in the control and data planes. This takes the form of the design of the 4MIDable framework, which, offers an architecture for deploying existing applications and taking advantage of programmability where it is possible to do so.

Our design centres on the concept of *Monitoring Agents*, self-contained modules which can parse the configuration and output of existing middle-boxes, generating flow rules to direct traffic to these appliances, as well as implementing policy, such as blocking malicious traffic entirely.

We have evaluated this design through a prototype implementation of the framework, focusing on the realisation of two use-cases to highlight the capabilities of the framework. Through our evaluation, we have demonstrated how our framework can be used to integrate an existing Intrusion Detection System with the framework to reduce the volume of traffic it is processing, without negatively impacting upon detection of network attacks. We have also shown how our framework can be used to effectively defend against attacks targeting the control plane infrastructure of a Software-Defined Network. This uses an autonomous data plane based approach, to limit traffic reaching the control-plane when an attack is underway.

To summarise, we present the design and implementation of the 4MIDable framework, which is designed around the notion of a hybrid deployment, where we are facilitating centralised control to be combined with de-centralised pipeline-based flexibility. With this, we present and evaluate two realisations of applications for the framework, P4ID, focusing on Intrusion Detection, and P4Protect, showing how the data plane can be used to effectively defend control infrastructure from attack.

6.1 Thesis Contributions

This thesis introduces a specific set of motivations and aims for a 'hybrid' deployment of emerging network technologies. We argue that whilst new technologies can introduce new capability and flexibility into the network, this is counter-pointed by the benefits and capabilities of existing approaches. With this need for a balance, we propose, taking into account existing designs and architectures, a means to bring these technologies together.

From these motivations, we propose the design of the 4MIDable framework, a multi-layered framework which separates the logic of security and monitoring applications from forwarding. This separation allows us to present a uniform interface to applications regardless of the underlying application, whilst still allowing for flexibility with the notion of 'pipeline-dependent' applications. With our design, we introduce the concept of *Monitoring Agents*. These agents are self-contained translation layers, capable of processing traffic, configurations and logging output from middle-boxes. Agents provide an interface between the SDN control-plane and existing middle-boxes, appliances and host-based applications. By combining these agents with the framework, we can receive feedback from appliances and use this to influence network forwarding policy.

Based on this comprehensive design, we implement a prototype SDN controller framework, built to enable the abstraction between monitoring, control and data plane layers. With this framework, we explore the capability that becomes possible with this approach, and the implementation considerations that must be taken into account when deploying the platform in hardware.

The framework serves as the basis for two use-cases which we introduce in the thesis. The first, P4ID, explores how Intrusion Detection Systems can be integrated with the processing capabilities available in contemporary P4 platforms. The approach here is to reduce the volume of traffic needing to be processed by an Intrusion Detection System whilst maintaining a level of detection equivalent to a scenario where traffic was not being filtered or pre-processed in any way. P4ID takes advantage of the processing flexibility offered by the *Data plane* layer of the framework, supplying a bespoke pipeline design. We combine this with a *Monitoring Agent* designed to work with an off-the-shelf Intrusion Detection System, Suricata. The *Monitoring Agent* is used to process the logging output of the IDS, directing the network to steer traffic towards it accordingly.

Whilst P4ID focuses on how we can integrate existing middle-boxes with programmable networks, P4Protect is designed to protect core network infrastructure. Software Defined Networks typically work to centralise the control and orchestration of networks, a feature that attacks can work to exploit. An attacker can be in a position to steer excessive volumes of traffic towards a controller, appliance or other infrastructure. With this, the target of the attack can become overloaded and either offer degraded performance, or be left in a state where all packet processing stops for the duration of the attack. P4Protect uses data plane based stateful tracking of connections being made to the control infrastructure. This is then used to identify and block attackers, to allow the infrastructure in question to recover. Our design for P4Protect offers adjustable parameters to vary the point at which protection engages to suit network demands.

With our evaluation, we analyse the performance and efficacy of both P4ID and P4Protect. With P4ID, we use traffic representative of contemporary real-world network attacks and establish a baseline for detection with an off-the-shelf IDS using a published rule set. This baseline is established from all traffic being steered towards the Intrusion Detection System. From this, we then establish P4ID as a pre-filter to the IDS, to establish how we can reduce traffic, whilst maintaining detection equivalent to sending all traffic to the IDS. We present implementations both for software and commodity hardware platforms. Our evaluation explores the impact on different configurations of the system, both in terms of the traffic processed by the IDS, and the proportion of detections that are successfully made. Configurations can also influence the detection of different types of attack, which we discuss in our evaluation.

For P4Protect, we establish a baseline using an off-the-shelf and self-described 'carrier-grade' SDN controller as the target for attack. We first establish an environment using OpenFlow switches being managed by the controller, and show how, with a fraction of the overall forwarding throughput of the virtual switches, cause the controller to enter into a state of non-responsiveness. We measure this not only through the CPU utilisation of the controller, but through packet processing performance too. In our evaluation, the attack is successful from a single attacker sending at a rate of 3000 packets per second. We highlight this problem using OpenVSwitch, as it is a production quality switch.

We replicate the same experiment again using ONOS, but using the P4 behavioural model, the reference software implementation of a P4 switch. With this, we again see a very similar impact on the controller, with the end result being that the controller is again forced into a state where it is unable to effectively process traffic.

Finally, we replicate this scenario but with P4Protect enabled in the data plane. We show that P4Protect is effective in detecting and defending against this type of attack. We plot the attack as a time series, and show where the attack remediation allows the controller to quickly recover from the attack and resume normal forwarding behaviour.

Further to the above, this thesis also provides a reference pipeline design for building pipelines combining a number of distinct pipeline stages from different applications. This enables a network administrator to build a pipeline to suit their requirements, using artefacts from the *Monitoring Agents* they wish to embed as part of their pipeline. We also provide for an abstract pipeline, this abstract pipeline is used to facilitate the integration of an existing SDN controller with the platform, by specifying a series of headers and tables that must be supported by the platform.

6.1.1 Commercial and Research Impacts

The research in this thesis highlights the benefits that can stem from integrating existing middle-boxes and techniques with Software-Defined Networks. Through our research, we show that the combination of intelligence in the control and data planes can provide a means to push more functionality into the network itself.

Our designs focus on using existing platforms, both in terms of new networking hardware, and existing middle-boxes. This approach serves to lower the barriers to adoption, by allowing for partial deployments, where programmable switches are only installed into the network where they are needed, allowing existing controllers to be integrated and deployed elsewhere in the network.

With our research, we not only provide the platform, but also example use-cases that have been evaluated in this thesis. These examples can be used as a basis and foundation for deploying a broader range of applications built using the framework. We also provide a reference monitoring pipeline, designed to provide a basis for adding functionality into.

6.1.2 Summary

In summary, this thesis has made the following contributions:

- Proposed a set of guidelines and features for a security framework for emerging SDN architectures, facilitating a hybrid deployment.
- Realised the 4MIDable framework, implementing many of the features described in this design, taking into account design considerations of current programmable platforms. This also includes the design of a reference pipeline for adding functionality into a programmable pipeline.
- Proposed the design and implementation of framework applications for Intrusion Detection and Control Plane Protection
- Evaluated the efficacy of offloading Intrusion Detection into P4 data planes, to show the traffic reduction that can be achieved whilst maintaining effective detection.
- Investigated the behaviour of an SDN controller under attack, both with OpenFlow and P4 switches. Contrasted this with a scenario where P4Protect data plane based protection is enabled.

6.2 Future Work

With a continuing drive towards network programmability and Software-Defined Networks, there is a continued interest, development and commercialisation of programmable platforms. As these ecosystems expand, new toolchains and hardware with ever greater processing capability, flexibility and capacity are launched.

With this, the scope of capability offered by a hybrid deployment will grow. However, in the nearer future, we propose avenues of future research.

6.2.1 Data Plane Based Attack Detection for P4ID

One key drawback we identified with P4ID is that it was only as effective as signature-based Intrusion Detection, improving upon the detection made by signature-based IDS was beyond the scope of this stage of the project.

Future work will include further integrating Intrusion Detection into our P4 pipeline. Our previously published work has shown that implementing 5-tuple based rules built from IDS signatures can have limited effectiveness, and highlighted the need for the stateful approach proposed in this thesis. However, even with this stateful approach, we propose that this can be improved further by handling protocol detection within the data plane. Protocol detection in the data plane would enable us to detect and defend against brute-force and denial-of-service attacks, as well as identifying protocols in use on 'non-standard' ports. This capability may be further enhanced by emerging hardware platforms with support for additional packet processing.

6.2.2 Control-Plane Integration for P4Protect

One of the limitations of P4Protect as evaluated is the entirely autonomous mode in which it operates. Whilst this autonomy provides an effective way of ensuring that no additional traffic is being sent to the control-plane when it is under attack, this approach does not provide the control-plane with any data to be able to defend against or mitigate against the attack through the rest of the network.

Future work in P4Protect will integrate a feedback loop with existing SDN control-planes, such that the controller can be alerted by the data plane when the protection has successfully engaged. One of the challenges with this will be determining the best point to notify the control-plane. If the control-plane is notified too early, then it could be a false positive, interrupting normal network operation. Equally, if the notification is sent to the controller when it is already saturated under attack, then the controller will not be able to remediate against the attack.

6.2.3 Framework Scalability

A key challenge in SDN is scaling as the network grows. Centralised control, as evidenced by our previous evaluations is vulnerable to overload situations when a controller is receiving an excessive number of requests.

Centralised control also offers a single point of failure if only a single instance of the controller is in operation. Scaling of commodity SDN controllers typically relies on replicated instances managing a subset of the network, or in some cases, consensus mechanisms are used to ensure consistency between control-plane replicas.

Bringing scalability into the framework will centre around using the replicated store first introduced in Chapter 4. Using this replicated store, we can explore the most appropriate co-ordination methods between instances of the framework, and explore the trade-offs of different approaches.

6.2.4 Automatic Pipeline Assembly

One of the drawbacks of our current prototype implementation is that a pipeline configuration has to be manually built from fragments by a network administrator.

This approach, whilst allowing the network administrator to customise the pipeline configuration to suit requirements involved a manual step when choosing to deploy new monitoring agents and applications as part of the network. If a pipeline configuration does not support a given monitoring agent and no suitable configuration is available, the pipeline configuration simply cannot be used as part of the network.

Recent research [145] [121] explores how pipelines can programmatically be combined and assembled from a series of packet processing artefacts. Adding this functionality would allow the framework to become more flexible and give scope for adding and removing features as the network demands. Removing unused features from the pipeline would offer more resources to other functions that are in use.

6.3 Concluding Remarks

In this thesis, we explore the integration of control-plane based SDN, traditional middle-boxes and programmable data planes. Through our evaluations, we show that we can offload from traditional appliances successfully, as well as offering new protection to network infrastructure.

Despite being an emerging platform, P4 shows great promise, with production-grade hardware and vendor support continuing to expand, alongside continuing research interest. The shift towards network softwarisation will benefit from a cross-platform approach to programming forwarding behaviour.

Acronyms

ACL Access Control List.

API Application Programmable Interface.

ARP Address Resolution Protocol.

ARPANET U.S. Advanced Research Projects Agency Network.

ASIC Application-Specific Integrated Circuit.

BYOD Bring Your Own Device.

CPU Central Processing Unit.

DDoS Distributed Denial-of-Service.

DHCP Dynamic Host Configuration Protocol.

DNS Dynamic Name System.

DoS Denial-of-Service.

DPI Deep Packet Inspection.

DVWA Damn Vulnerable Web App.

ECMP Equal Cost Multipath Routing.

FPGA Field Programmable Gate Array.

FTP File Transfer Protocol.

GUI Graphical User Interface.

HOIC High-Orbit Ion Cannon.

HTTP HyperText Transport Protocol.

HTTPS HyperText Transport Protocol Secure.

ICMP Internet Control Message Protocol.

ICS Industrial Control System.

IDS Intrusion Detection System.

INT In-Band Network Telemetry.

IP Internet Protocol.

IPFIX IP Flow Information Export.

IPS Intrusion Prevention System.

ISP Internet Service Provider.

JSON JavaScript Object Notation.

LLDP Link Layer Discovery Protocol.

LOIC Low Orbit Ion Cannon.

LPM Longest Prefix Match.

LTS Long-term Support.

MAC Media Access Control.

MACSEC Media Access Control Security.

NAT Network Address Translation.

NMAP Network Mapper.

ONOS Open Network Operating System.

OVS OpenVSwitch.

P4 Programmable Protocol-Independent Packet Processors.

PSA Portable Switch Architecture.

RTT Round Trip Time.

SDN Software Defined Networking.

SNMP Simple Network Management Protocol.

SRAM Static Random Access Memory.

TCAM Ternary Content Addressable Memory.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

VLAN Virtual Local Area Network.

VNF Virtual Network Function.

WAN Wide Area Network.

WCMP Weighted Cost Multipath Routing.

XAMPP XAMPP Apache + MariaDB + PHP + Perl.

XSS Cross-Site Scripting.

References

1. AlSabeh, A., Kfoury, E., Crichigno, J. & Bou-Harb, E. *P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection* in *NDSS Symposium 2022* (2022).

2. Alsadi, A., Berardi, D., Callegati, F., Melis, A. & Prandini, M. *A Security Monitoring Architecture based on Data Plane Programmability* in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)* (2021), 389–394.

3. Appelman, M. & de Boer, M. Performance analysis of OpenFlow hardware. *University of Amsterdam, Tech. Rep,* 2011–2012 (2012).

4. Barham, P. *et al.* Techniques for lightweight concealment and authentication in IP networks. *Intel Research Berkeley. July* (2002).

5. Berde, P. *et al. ONOS: towards an open, distributed SDN OS* in *Proceedings of the third workshop on Hot topics in software defined networking* (2014), 1–6.

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

7. Biondi, P. Scapy: explore the net with new eyes. *Technical report, Technical report, EADS Corporate Research Center* (2005).

8. Boite, J., Nardin, P.-A., Rebecchi, F., Bouet, M. & Conan, V. *Statesec: Stateful monitoring for DDoS protection in software defined networks* in *2017 IEEE Conference on Network Softwarization (NetSoft)* (2017), 1–9. doi:10.1109/NETSOFT.2017.8004113.

9. Bonfim, M., Santos, M., Dias, K. & Fernandes, S. A real-time attack defense framework for 5G network slicing. *Software: Practice and Experience* (2020).

10. Bosshart, P. *et al.* Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* **43,** 99–110 (2013).

11. Bosshart, P. *et al.* P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* **44,** 87–95. ISSN: 0146-4833. doi:10.1145/2656877.2656890 (2014).

12. Boyanov, P. Educational exploiting the information resources and invading the security mechanisms of the operating system Windows 7 with the exploit Eternalblue and Backdoor Doublepulsar. *Association Scientific and Applied Research* **14,** 34 (2018).

13. Braden, R., Clark, D., Crocker, S. & Huitema, C. *Report of IAB Workshop on Security in the Internet Architecture - February 8-10, 1994* RFC 1636 (RFC Editor, June 1994).

14. Budiu, M. & Dodd, C. The p416 programming language. *ACM SIGOPS Operating Systems Review* **51,** 5–14 (2017).

15. Carpenter, B. & Brim, S. *Middleboxes: Taxonomy and Issues* RFC 3234 (RFC Editor, Feb. 2002). `http://www.rfc-editor.org/rfc/rfc3234.txt`.

16. Casado, M. *et al. SANE: A Protection Architecture for Enterprise Networks.* in *USENIX Security Symposium* **49** (2006), 50.

17. Casado, M. *et al.* Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review* **37,** 1–12 (2007).

18. Castanheira, L., Parizotto, R. & Schaeffer-Filho, A. E. *Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4* in *ICC 2019-2019 IEEE International Conference on Communications (ICC)* (2019), 1–6.

19. Chapman, D. B. *Network (In) Security Through IP Packet Filtering.* in *USENIX Summer* (1992).

20. Chen, X.-F. & Yu, S.-Z. CIPA: A collaborative intrusion prevention architecture for programmable network and SDN. *Computers & Security* **58,** 1–19 (2016).

21. Cheswick, B. *The design of a secure internet gateway* in *USENIX Summer Conference Proceedings* (1990).

22. Ching-Hao, C. & Lin, Y.-D. OpenFlow version roadmap. *Study, Dept. of Computer Science, National Chiao Tung University, Taiwan* (2015).

23. Chowdhury, S. R., Bari, M. F., Ahmed, R. & Boutaba, R. *Payless: A low cost network monitoring framework for software defined networks* in *2014 IEEE Network Operations and Management Symposium (NOMS)* (2014), 1–9.

24. Claise, B., Fullmer, M., Calato, P. & Penno, R. Ipfix protocol specification. *Interrnet-draft, work in progress* (2005).

25. Claise, B., Sadasivan, G., Valluri, V. & Djernaes, M. Cisco systems netflow services export version 9 (2004).

26. Cordero, C. G. *et al. ID2T: A DIY dataset creation toolkit for Intrusion Detection Systems* in *2015 IEEE Conference on Communications and Network Security (CNS)* (2015), 739–740.

27. Cotton, M., Eggert, L., Touch, J., Westerlund, M. & Cheshire, S. *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry* BCP 165 (RFC Editor, Aug. 2011).

28. Curtis, A. R. *et al. DevoFlow: Scaling flow management for high-performance networks* in *Proceedings of the ACM SIGCOMM 2011 conference* (2011), 254–265.

29. Dao, T.-N., Hoang, V.-P., Ta, C. H., *et al. Development of Lightweight and Accurate Intrusion Detection on Programmable Data Plane* in *2021 International Conference on Advanced Technologies for Communications (ATC)* (2021), 99–103.

30. Datta, R., Choi, S., Chowdhary, A. & Park, Y. *P4guard: Designing p4 based firewall* in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)* (2018), 1–6.

31. Delgadillo, K. & Marketing, C. I. P. Netflow services and applications. *Cisco Whitepaper* (1996).

32. Delio, M. *The Greatest Hacks of All Time* Feb. 2001. `https://www.wired.com/2001/02/the-greatest-hacks-of-all-time/`.

33. Deng, J. *et al. VNGuard: An NFV/SDN combination framework for provisioning and managing virtual firewalls* in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)* (2015), 107–114.

34. Ding, D., Savi, M., Antichi, G. & Siracusa, D. An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection. *IEEE Transactions on Network and Service Management* **17,** 75–88. doi:`10.1109/TNSM.2020.2968979` (2020).

35. Ding, D., Savi, M. & Siracusa, D. Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4. *arXiv preprint arXiv:2104.05117* (2021).

36. Donovan, A. A. & Kernighan, B. W. *The Go programming language* (Addison-Wesley Professional, 2015).

37. *Emerging Threats Documentation* 2021. `https://doc.emergingthreats.net/` (2021).

38. Falliere, N. Sality: Story of a peer-to-peer viral network. *Rapport technique, Symantec Corporation* **32** (2011).

39. Fawcett, L., Scott-Hayward, S., Broadbent, M., Wright, A. & Race, N. Tennison: a distributed SDN framework for scalable network security. *IEEE Journal on Selected Areas in Communications* **36,** 2805–2818 (2018).

40. Feng, W., Zhang, Z.-L., Liu, C. & Chen, J. *Clé: Enhancing Security with Programmable Dataplane Enabled Hybrid SDN* in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies* (2019), 76–77.

41. Fenrich, K. Securing your control system: the "CIA triad" is a widely used benchmark for evaluating information system security effectiveness. *Power Engineering* **112,** 44–49 (2008).

42. Fingerhut, A. *P4 Guide* `https://github.com/jafingerhut/p4-guide`. 2020.

43. Foundation, O. N. *Performance of BMV2* `https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md`. 2016.

44. Foundation, O. N. *Switch.p4* `https://github.com/p4lang/switch`. 2016.

45. Foundation, O. N. *P4 OpenFlow Agent* `https://github.com/p4lang/p4ofagent`. 2017.

46. Foundation, O. N. *p4spec: P4 Language Specification* `https://github.com/p4lang/p4-spec`. 2020.

47. Ghafur, S. *et al.* A retrospective impact analysis of the WannaCry cyberattack on the NHS. *npj Digital Medicine* **2,** 98. doi:`10.1038/s41746-019-0161-6` (2019).

48. González, L. A. Q., Castanheira, L., Marques, J. A., Schaeffer-Filho, A. & Gaspary, L. P. *BUNGEE: An Adaptive Pushback Mechanism for DDoS Detection and Mitigation in P4 Data Planes* in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)* (2021), 393–401.

49. Gouda, M. G. & Liu, X.-Y. *Firewall design: Consistency, completeness, and compactness* in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* (2004), 320–327.

50. Gray, N., Dietz, K., Seufert, M. & Hossfeld, T. *High Performance Network Metadata Extraction Using P4 for ML-based Intrusion Detection Systems* in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)* (2021), 1–7.

51. Greenberg, A. *et al.* A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communication Review* **35,** 41–54 (2005).

52. Greenberg, A. *The Untold Story of NotPetya, the Most Devastating Cyberattack in History* Oct. 2018. `https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/`.

53. Group, T. P. A. W. `https://p4.org/p4-spec/docs/PSA.html`.

54. Gutiérrez, S. A., Branch, J. W., Gaspary, L. P. & Botero, J. F. Watching Smartly from the Bottom: Intrusion Detection revamped through Programmable Networks and Artificial Intelligence. *arXiv preprint arXiv:2106.00239* (2021).

55. Halfond, W. G., Viegas, J., Orso, A., *et al. A classification of SQL-injection attacks and countermeasures* in *Proceedings of the IEEE international symposium on secure software engineering* **1** (2006), 13–15.

56. Hancock, D. & Van der Merwe, J. *Hyper4: Using p4 to virtualize the programmable data plane* in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), 35–49.

57. Harkous, H., Jarschel, M., He, M., Priest, R. & Kellerer, W. *Towards understanding the performance of P4 programmable hardware* in *2019 ACM/IEEE*

Symposium on Architectures for Networking and Communications Systems (ANCS) (2019), 1–6.

58. Hauben, M. History of ARPANET. *Site de l'Instituto Superior de Engenharia do Porto* **17** (2007).

59. Hauser, F., Schmidt, M., Häberle, M. & Menth, M. P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-SDN. *arXiv preprint arXiv:1904.07088* (2019).

60. Heinanen, J. & Guerin, R. *A Two Rate Three Color Marker* RFC 2698 (RFC Editor, Sept. 1999).

61. Huang, Q. *et al. Sketchvisor: Robust network measurement for software packet processing* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), 113–126.

62. Hypolite, J. *et al. DeepMatch: practical deep packet inspection in the data plane using network processors* in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), 336–350.

63. Ibanez, S., Brebner, G., McKeown, N. & Zilberman, N. *The P4-¿ NetFPGA workflow for line-rate packet processing* in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2019), 1–9.

64. Ilascu, I. *Over 25% of all UK universities were attacked by ransomware* Aug. 2020. `https://www.bleepingcomputer.com/news/security/over-25-percent-of-all-uk-universities-were-attacked-by-ransomware/`.

65. Ilha, A. d. S., Lapolli, A. C., Marques, J. A. & Gaspary, L. P. Euclid: A Fully In-Network, P4-based Approach for Real-Time DDoS Attack Detection and Mitigation. *IEEE Transactions on Network and Service Management* **PP,** 1–1. doi:`10.1109/tnsm.2020.3048265` (2020).

66. Ingham, K. & Forrest, S. A history and survey of network firewalls. *University of New Mexico, Tech. Rep* (2002).

67. Jain, S. *et al.* B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* **43,** 3–14 (2013).

68. Jazi, H. H., Gonzalez, H., Stakhanova, N. & Ghorbani, A. A. Detecting HTTP-based application layer DoS attacks on web servers in the presence of sampling. *Computer Networks* **121,** 25–36 (2017).

69. Jeong, S., Lee, D., Choi, J., Li, J. & Hong, J. W.-K. *Application-aware traffic management for OpenFlow networks* in *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)* (2016), 1–5.

70. Jiang, H., Xie, G. & Salamatian, K. *Load balancing by ruleset partition for parallel IDS on multi-core processors* in *International Conference on Computer Communications and Networks, ICCCN* (2013).

71. Julkunen, H. & Chow, C. E. *Enhance network security with dynamic packet filter* in *Proceedings 7th International Conference on Computer Communications and Networks (Cat. No. 98EX226)* (1998), 268–275.

72. Kang, Q. *et al. Programmable in-network security for context-aware BYOD policies* in *Proc. USENIX Security* (2020).

73. Kaur, S., Singh, J. & Ghumman, N. S. *Network programmability using POX controller* in *International Conference on Communication, Computing & Systems (ICCCN'2014)* (2014), 134–138.

74. Keles, A., Angin, P., Alemdar, H. & Onur, E. *DroPPPP: A P4 Approach to Mitigating DoS Attacks in SDN* in *Information Security Applications: 20th International Conference, WISA 2019, Jeju Island, South Korea, August 21–24, 2019, Revised Selected Papers* **11897** (2020), 55.

75. Koulouris, T., Mont, M. C. & Arnell, S. SDN4S: Software defined networking for security. *Hewlett Packard Labs, Palo Alto, CA, USA, Tech. Rep* (2017).

76. Landreth, B. & Rheingold, H. *Out of the inner circle: a hacker's guide to computer security* (Microsoft Press Bellevue, Washington, 1985).

77. Lantz, B., Heller, B. & McKeown, N. *A Network in a Laptop: Rapid Prototyping for Software-Defined Networks* in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Association for Computing Machinery, Monterey, California, 2010). ISBN: 9781450304092. doi:10.1145/1868447.1868466. https://doi.org/10.1145/1868447.1868466.

78. Lara, A. & Ramamurthy, B. OpenSec: Policy-based security using software-defined networking. *IEEE transactions on network and service management* **13,** 30–42 (2016).

79. Lee, S., Kim, J., Shin, S., Porras, P. & Yegneswaran, V. *Athena: A Framework for Scalable Anomaly Detection in Software-Defined Networks* in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017), 249–260. doi:10.1109/DSN.2017.42.

80. Lewis, B., Broadbent, M. & Race, N. *P4ID: P4 enhanced intrusion detection* in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2019), 1–4.

81. Lewis, B., Broadbent, M., Rotsos, C. & Race, N. 4MIDable: Flexible Network Offloading For Security VNFs. *Journal of Network and Systems Management* **31,** 52 (2023).

82. Lewis, B., Fawcett, L., Broadbent, M. & Race, N. *Using p4 to enable scalable intents in software defined networks* in *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), 442–443.

83. Li, J., Jiang, H., Jiang, W., Wu, J. & Du, W. *SDN-based Stateful Firewall for Cloud* in *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)* (2020), 157–161. doi:10.1109/BigDataSecurity-HPSC-IDS49724.2020.00037.

84. Liu, Z., Manousis, A., Vorsanger, G., Sekar, V. & Braverman, V. *One sketch to rule them all: Rethinking network flow monitoring with univmon* in *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), 101–114.

85. Longstaff, T. A. *et al.* Security of the Internet. *The Froehlich/Kent Encyclopedia of Telecommunications* **15,** 231–255 (1997).

86. Lyon, G. F. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning* (Insecure. Com LLC (US), 2008).

87. Macıas, S. G. & Botero, J. F. Performance Evaluation of the ONOS Controller Under an DDoS Attack (2019).

88. Manavi, M. T. Defense mechanisms against distributed denial of service attacks: A survey. *Computers & Electrical Engineering* **72,** 26–38 (2018).

89. Mann, V., Vishnoi, A. & Bidkar, S. *Living on the edge: Monitoring network flows at the edge in cloud data centers* in *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)* (2013), 1–9.

90. Martins, J. *et al. ClickOS and the art of network function virtualization* in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), 459–473.

91. McKeown, N. & Rexford, J. *Clarifying the differences between P4 and OpenFlow* 2016. `https://p4.org/p4/clarifying-the-differences-between-p4-and-openflow.html`.

92. McKeown, N. *et al.* OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* **38,** 69–74. ISSN: 0146-4833. doi:`10.1145/1355734.1355746` (2008).

93. Merkel, D. *et al.* Docker: lightweight linux containers for consistent development and deployment. *Linux journal* **2014,** 2 (2014).

94. Mogul, J. C. *Simple and flexible datagram access controls for UNIX-based gateways* in *In USENIX Conference Proceedings* (1989).

95. Mohurle, S. & Patil, M. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science* **8,** 1938–1940 (2017).

96. Mori, T. *et al.* Identifying heavy-hitter flows from sampled flow statistics. *IEICE Transactions on Communications* **90,** 3061–3072 (2007).

97. Narayanan, N., Sankaran, G. C. & Sivalingam, K. M. *Mitigation of security attacks in the SDN data plane using P4-enabled switches* in *2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)* (2019), 1–6. doi:`10.1109/ANTS47819.2019.9118071`.

98. Ndonda, G. K. & Sadre, R. *A two-level intrusion detection system for industrial control system networks using p4* in *5th International Symposium for ICS & SCADA Cyber Security Research 2018 5* (2018), 31–40.

99. Niu, B., Kong, J., Tang, S., Li, Y. & Zhu, Z. Visualize Your IP-Over-Optical Network in Realtime: A P4-Based Flexible Multilayer In-Band Network Telemetry (ML-INT) System. *IEEE Access* **7,** 82413–82423. doi:`10.1109/ACCESS.2019.2924332` (2019).

100. Oh, B.-H., Vural, S. & Wang, N. A Lightweight Scheme of Active-port Aware Monitoring in Software-Defined Networks. *IEEE Transactions on Network and Service Management* (2021).

101. Open Information Security Foundation. *Suricata* 2010. `https://suricata.io/`.

102. *Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions* Mar. 2021. `https://opennetworking.org/onos/`.

103. Open Networking Foundation. *Behavioural Model version 2* `https://github.com/p4lang/behavioral-model`. 2016.

104. Open Networking Lab. *P4Runtime-enabled Mininet Docker Image* `https://github.com/opennetworkinglab/p4mn-docker`. 2021.

105. Oppliger, R. Internet security: firewalls and beyond. *Communications of the ACM* **40,** 92–102 (1997).

106. Orebaugh, A., Ramirez, G. & Beale, J. *Wireshark & Ethereal network protocol analyzer toolkit* (Elsevier, 2006).

107. Orman, H. The Morris worm: a fifteen-year perspective. *IEEE Security Privacy* **1,** 35–43. doi:`10.1109/MSECP.2003.1236233` (2003).

108. Osiński, T., Tarasiuk, H., Rajewski, L. & Kowalczyk, E. *DPPx: A P4-based Data Plane Programmability and Exposure framework to enhance NFV services* in *2019 IEEE Conference on Network Softwarization (NetSoft)* (2019), 296–300.

109. Ozdag, R. White Paper: Intel Ethernet Switch FM6000 Series. `https://people.ucsc.edu/~warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf` (2012).

110. P4 Language Consortium. *P4Runtime GitHub Repository* 2017. `https://github.com/p4lang/PI`.

111. *P416 Language Specification* `https://p4.org/p4-spec/docs/P4-16-v1.2.1.html`.

112. Paolucci, F. *et al.* P4 edge node enabling stateful traffic engineering and cyber security. *Journal of Optical Communications and Networking* **11,** A84–A95 (2019).

113. Parizotto, R. *et al. ShadowFS: Speeding-up Data Plane Monitoring and Telemetry using P4* in *ICC 2020-2020 IEEE International Conference on Communications (ICC)* (2020), 1–6.

114. Park, T., Xu, Z. & Shin, S. *HEX Switch: Hardware-Assisted Security Extensions of OpenFlow* in *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges* (Association for Computing Machinery, Budapest, Hungary, 2018), 33–39. ISBN: 9781450359122. doi:10.1145/3229616.3229622. https://doi.org/10.1145/3229616.3229622.

115. Paxson, V. Bro: a system for detecting network intruders in real-time. *Computer networks* **31,** 2435–2463 (1999).

116. Paxson, V., Campbell, S., Lee, J., *et al. Bro intrusion detection system* tech. rep. (Lawrence Berkeley National Laboratory, 2006).

117. Pereira, F., Neves, N. & Ramos, F. M. *Secure network monitoring using programmable data planes* in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2017), 286–291.

118. Pfaff, B. *et al. The design and implementation of open vswitch* in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), 117–130.

119. Phaal, P., Panchen, S. & McKee, N. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks (2001).

120. Phan, X. T. & Fukuda, K. Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks. *Journal of Information Processing* **25,** 182–190 (2017).

121. Pontarelli, S. *et al. Flowblaze: Stateful packet processing in hardware* in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), 531–548.

122. Postel, J. & Reynolds, J. *Standard for the transmission of IP datagrams over IEEE 802 networks* STD 43 (RFC Editor, Feb. 1988). `http://www.rfc-editor.org/rfc/rfc1042.txt`.

123. Postel, J. *Internet Protocol* STD 5 (RFC Editor, Sept. 1981). `http://www.rfc-editor.org/rfc/rfc791.txt`.

124. Postel, J. *Transmission Control Protocol* STD 7 (RFC Editor, Sept. 1981). `http://www.rfc-editor.org/rfc/rfc793.txt`.

125. Psounis, K. Active networks: Applications, security, safety, and architectures. *IEEE Communications Surveys* **2,** 2–16 (1999).

126. Rahimi, R. *et al. A high-performance OpenFlow software switch* in *2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR)* (2016), 93–99.

127. Ranum, M. J. *et al.* Implementing a generalized tool for network monitoring. *Information Security Technical Report* **3,** 53–64 (1998).

128. Roesch, M. *et al. Snort: Lightweight intrusion detection for networks.* in *Lisa* **99** (1999), 229–238.

129. Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R. & Moore, A. W. *OFLOPS: An open framework for OpenFlow switch evaluation* in *International Conference on Passive and Active Network Measurement* (2012), 85–95.

130. Salus, P. H. & Foreword By-Cerf, V. G. *Casting the Net: From ARPANET to Internet and Beyond...* (Addison-Wesley Longman Publishing Co., Inc., 1995).

131. Schwartz, B. *et al. Smart packets for active networks* in *1999 IEEE Second Conference on Open Architectures and Network Programming. Proceedings. OPENARCH'99 (Cat. No. 99EX252)* (1999), 90–97.

132. Sekar, V., Krishnaswamy, R., Gupta, A. & Reiter, M. K. *Network-wide deployment of intrusion detection and prevention systems* in *Proceedings of the 6th International COnference* (2010), 1–12.

133. Sharafaldin, I., Lashkari, A. H. & Ghorbani, A. A. *Toward generating a new intrusion detection dataset and intrusion traffic characterization.* in *ICISSp* (2018), 108–116.

134. Shekyan, S. *SlowHTTPTest* `https://github.com/shekyan/slowhttptest`. 2021.

135. Shin, S. W. *et al. Fresco: Modular composable security services for software-defined networks* in *20th Annual Network & Distributed System Security Symposium* (2013).

136. Shin, S. & Gu, G. *CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)* in *2012 20th IEEE international conference on network protocols (ICNP)* (2012), 1–6.

137. Shirali-Shahreza, S. & Ganjali, Y. *FleXam: flexible sampling extension for monitoring and security applications in openflow* in *Proceedings of the second*

ACM SIGCOMM workshop on Hot topics in software defined networking (2013), 167–168.

138. Shoch, J. F. & Hupp, J. A. The "worm" programs—early experience with a distributed computation. *Communications of the ACM* **25,** 172–180 (1982).

139. Shorey, T., Subbaiah, D., Goyal, A., Sakxena, A. & Mishra, A. K. *Performance comparison and analysis of slowloris, goldeneye and xerxes ddos attack tools* in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (2018), 318–322.

140. Silva, C., Sousa, B. & Vilela, J. P. CROCUS: An objective approach for SDN controllers security assessment (2021).

141. Stoll, C. Stalking the wily hacker. *Communications of the ACM* **31,** 484–497 (1988).

142. Su, Z., Wang, T., Xia, Y. & Hamdi, M. *FlowCover: Low-cost flow monitoring scheme in software defined networks* in *2014 IEEE Global Communications Conference* (2014), 1956–1961.

143. Suarez-Varela, J. & Barlet-Ros, P. *Towards a NetFlow implementation for OpenFlow software-defined networks* in *2017 29th International Teletraffic Congress (ITC 29)* **1** (2017), 187–195.

144. Suárez-Varela, J. & Barlet-Ros, P. Reinventing netflow for openflow software-defined networks. *arXiv preprint arXiv:1702.06803* (2017).

145. Sultana, N. *et al. Flightplan: Dataplane Disaggregation and Placement for P4 Programs* in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021), 571–592.

146. Tavares, K. & Ferreto, T. *P4-ONIDS: A P4-based NIDS optimized for constrained programmable data planes in SDN* in *Anais do XXXIX Simposio Brasileiro de Redes de Computadores e Sistemas Distribuidos* (2021), 434–447.

147. *The most popular database for modern apps* `https://www.mongodb.com/`.

148. Tjhai, G. C., Papadaki, M., Furnell, S. & Clarke, N. L. *Investigating the problem of IDS false alarms: An experimental study using Snort* in *IFIP International Information Security Conference* (2008), 253–267.

149. Tofino, B. *World's fastest P4-programmable Ethernet switch ASICs* 2018.

150. Toh, A. *Azure Ddos Protection-2021 Q3 and Q4 ddos attack trends* `https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q3-and-q4-ddos-attack-trends/`.

151. Tomonori, F. Introduction to ryu sdn framework. *Open Networking Summit,* 1–14 (2013).

152. Turner, A. Tcpreplay. *http://tcpreplay. synfin. net/trac/* (2011).

153. Van Adrichem, N. L., Doerr, C. & Kuipers, F. A. *Opennetmon: Network monitoring in openflow software-defined networks* in *2014 IEEE Network Operations and Management Symposium (NOMS)* (2014), 1–8.

154. Van Tu, N., Hyun, J. & Hong, J. W.-K. *Towards onos-based sdn monitoring using in-band network telemetry* in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)* (2017), 76–81.

155. Van Tu, N., Hyun, J., Kim, G. Y., Yoo, J.-H. & Hong, J. W.-K. *Intcollector: A high-performance collector for in-band network telemetry* in *2018 14th*

International Conference on Network and Service Management (CNSM) (2018), 10–18.

156. Vestin, J. *et al. Programmable Event Detection for In-Band Network Telemetry* in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)* (2019), 1–6. doi:`10.1109/CloudNet47604.2019.9064137`.

157. Vörös, P. & Kiss, A. *Security middleware programming using P4* in *International Conference on Human Aspects of Information Security, Privacy, and Trust* (2016), 277–287.

158. Wang, S., Chavez, K. G. & Kandeepan, S. *SECO: SDN sEcure COntroller algorithm for detecting and defending denial of service attacks* in *2017 5th International Conference on Information and Communication Technology (ICoIC7)* (2017), 1–6. doi:`10.1109/ICoICT.2017.8074692`.

159. White, J., Fitsimmons, T. & Matthews, J. *Quantitative Analysis of Intrusion Detection Systems: Snort and Suricata* in. **8757** (Apr. 2013). doi:`10.1117/12.2015616`.

160. White, J. S., Fitzsimmons, T. & Matthews, J. N. *Quantitative analysis of intrusion detection systems: Snort and Suricata* in *Cyber sensing 2013* **8757** (2013), 875704.

161. Wood, R. *Damn Vulnerable Web App* `https://github.com/digininja/DVWA`. 2021.

162. Xing, J., Wu, W. & Chen, A. *Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries* in *30th USENIX Security Symposium (USENIX Security 21)* (2021).

163. Xing, T., Huang, D., Xu, L., Chung, C.-J. & Khatkar, P. *Snortflow: A openflow-based intrusion prevention system in cloud environment* in *2013 second GENI research and educational experiment workshop* (2013), 89–92.

164. Yu, C. *et al. Flowsense: Monitoring network utilization with zero measurement cost* in *International Conference on Passive and Active Network Measurement* (2013), 31–41.

165. Yu, L., Sonchack, J. & Liu, V. Mantis: Reactive Programmable Switches, 296–309. doi:`10.1145/3387514.3405870` (2020).

166. Yu, M., Jose, L. & Miao, R. *Software Defined Traffic Measurement with OpenSketch* in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), 29–42.

167. Yu, M., Rexford, J., Freedman, M. J. & Wang, J. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review* **40,** 351–362 (2010).

168. Yu, T., Fayaz, S. K., Collins, M. P., Sekar, V. & Seshan, S. *PSI: Precise Security Instrumentation for Enterprise Networks.* in *NDSS* (2017).

169. Zaballa, E. O., Franco, D., Zhou, Z. & Berger, M. S. *P4Knocking: Offloading host-based firewall functionalities to the network* in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)* (2020), 7–12.

170. Zhang, K., Zhuo, D. & Krishnamurthy, A. *Gallium: Automated Software Middlebox Offloading to Programmable Switches* in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the*

applications, technologies, architectures, and protocols for computer communication (2020), 283–295.