Advancing Deep Packet Inspection in SDNs: A Comparative Analysis of P4 and OpenFlow Programmability

Anthony J. Bustamante

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Cybersecurity Engineering

University of Washington

2024

Committee:

Brent Lagesse

Geetha Thamilarasu

Marc J Dupuis

Department Authorized to Offer Degree:

Computing and Software System

Abstract

This thesis undertakes a critical examination of Deep Packet Inspection (DPI) capabilities within Software-Defined Networking (SDN) frameworks, emphasizing the comparative efficacy of P4 programming language against the conventional OpenFlow protocol.

OpenFlow, while foundational in SDN's evolution, exhibits notable constraints in DPI's domain, primarily due to its limited packet inspection depth, confined largely to the Transport, Network, Data Link and Physical layers.

In contrast, this research advocates for the adoption of P4 for its unique flexibility and programmability, potentially extending DPI functionalities to the application layer (Layer 7), thereby addressing and potentially surpassing OpenFlow's limitations.

Employing a methodical approach, this study harnesses Open vSwitch and BMv2 (Behavioral Model version 2) switches to emulate real-world network scenarios. These emulations facilitate a head-to-head comparison of OpenFlow and P4 in executing DPI tasks, particularly focusing on HTTP and SQL protocols — common vectors for network threats. Through a comprehensive suite of protocols including OpenFlow, gRPC (Google Remote Procedure Call), and P4Runtime, the research crafts a robust DPI framework, further complemented by a custom-developed controller designed for the BMv2 and P4 ecosystem.

The research culminates providing three different implementations to do Deep Packet Inspection within SDN domain, benchmarking each of them to measure their advantages and disadvantages. With these implementations and benchmarking, we not only aim to validate P4's superiority over OpenFlow in managing DPI tasks but we also seek to dynamically adapt packet-processing techniques to the ever-evolving landscape of network threats. By advancing SDN functionalities beyond traditional layer boundaries, this thesis contributes significantly to the discourse on network security, management, and optimization, paving the way for future innovations in increasingly complex network environments.

Contents

List of Figures

Chapter 1

Introduction

Software-Defined Networking (SDN) has changed how networks are managed by making data programmable, centralizing control, and allowing for quick adjustments [1]. SDN separates the control and data planes, which means network behaviors can be programmed without being limited by hardware. This thesis explores how SDN can improve Deep Packet Inspection (DPI), especially for inspecting data at the application layer, a level that has mostly been managed by traditional protocols like OpenFlow [2].

With the increase of cloud computing and other paradigms, networks are facing more advanced security threats [3]. This situation highlights the need for better network security measures. DPI, which usually looks at packet content up to the Application layer, becomes very important for improved security. However, the traditional protocols used in SDN, like OpenFlow, are not very good at analyzing application-layer traffic since it was mainly created to manage L1-L4 protocols. This makes it important to find new DPI methods that can examine deeper into packet payloads [4].

This research tries to fill a big gap in understanding how SDN can enhance network security with advanced DPI methods. It uses the P4 programming language to make data-plane more programmable, which allows for more thorough inspection of network traffic, beyond what current protocols can handle.

This study uses BMv2 switches (which were designed to work with P4) to test how effectively DPI can identify and stop network threats. The study uses advanced protocols like OpenFlow, gRPC (Google Remote Procedure Call), and P4Runtime to support a complex DPI framework within SDN environments [5].

Another major part of this thesis is crafting multiple implementations to do Deep Packet Inspection and test our hypotheses, as well as the design of a controller and applications to carry on our tests.

The final result of this research is conclusive results about how P4 implementations compare against OpenFlow implementations for Deep Packet Inspection, besides, we will also provide a DPI framework that goes beyond basic SDN principles.

1.1 Research Hypothesis

The main idea of this thesis is that using the P4 programming language can help us do a better job than the current OpenFlow method in inspecting application traffic because P4 is more flexible and can be adjusted more easily [6–8]. We think that P4 can make Deep Packet Inspection (DPI) better by allowing us to be more detailed in our analysis and faster in responding to network threats [5, 9, 10].

1.1.1 Hypothesis One: Flexibility and Application-layer Analysis

We believe that the flexibility offered by P4 can also be extended to analyze application-layer traffic beyond pure networking applications, in turn, helping us perform Deep Packet Inspection on Data Plane. This is actually an approach revalidation because in 2023 Ahad et al. [11] and S. Gupta et al. [5] proposed a recirculation algorithm to do Deep Packet Inspection on Data Plane, however, they did not intend to do the comparisons we are addressing in this thesis.

1.1.2 Hypothesis Two: Performance and Efficiency

Our second idea is that DPI using P4 will not only work better but also faster because it would give us the possibility to do DPI on Data Plane. This means quicker reactions to potential threats and less waiting time for data to be checked, all without losing the quality of our inspections [5, 10].

1.1.3 Hypothesis Three: Resource Usage

Lastly, we think that DPI with P4 will use less computing resources compared to methods that rely heavily on the control plane of the network. This would mean a more efficient way of managing network security [12, 13].

Each of these points will be tested by comparing how well P4 and OpenFlow handle different types of network security challenges. We will look at things like how much computer power they use, how quickly they process information, and how well they keep our networks safe, to evaluate the difference between P4 and Openflow we will calculate the percentage difference for P4 implementation's results vs Openflow results, and if we find that there is a increment of accuracy or reduction of latency and resource consumption, then, our hypotheses will be corroborated.

Another thing to keep in mind is that,we expect these differences to be observed better in Data plane implementations, due to P4 would allow us to use the Data Plane for Deep Packet Inspection which would mean that the traffic will not have to travel up to the control plane to be processed.

Chapter 2

SDN Ecosystem: Technologies and Protocols

The transformative paradigm of Software-Defined Networking (SDN) has redefined the orchestration of network resources. This chapter offers an overview of the technologies and protocols that are pivotal in shaping the SDN landscape, particularly focusing on their roles and interactions within the scope of Deep Packet Inspection (DPI) [14–17].

2.1 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) is an innovative networking paradigm that has introduced a significant shift in network design and management. By abstracting the control logic from the hardware layer, SDN introduces a level of flexibility and programmability previously unattainable in traditional network architectures [14–17].

2.1.1 Overview of SDN

SDN emerged as a response to the growing complexity and static nature of conventional networks. At its core, SDN separates the network's control logic (the control plane) from the actual traffic forwarding (the data plane). This separation enables network administrators to manage network behavior through abstracted and centralized control mechanisms, often realized through a programmable software application. The evolution of SDN is characterized by the progressive decoupling of control and data functionalities, culminating in a highly dynamic, manageable, and cost-effective network infrastructure that can quickly adapt to changing needs [14–17].

2.1.2 Architecture of SDN

The architecture of SDN can be divided into three fundamental layers as we can see in figure 2.1:

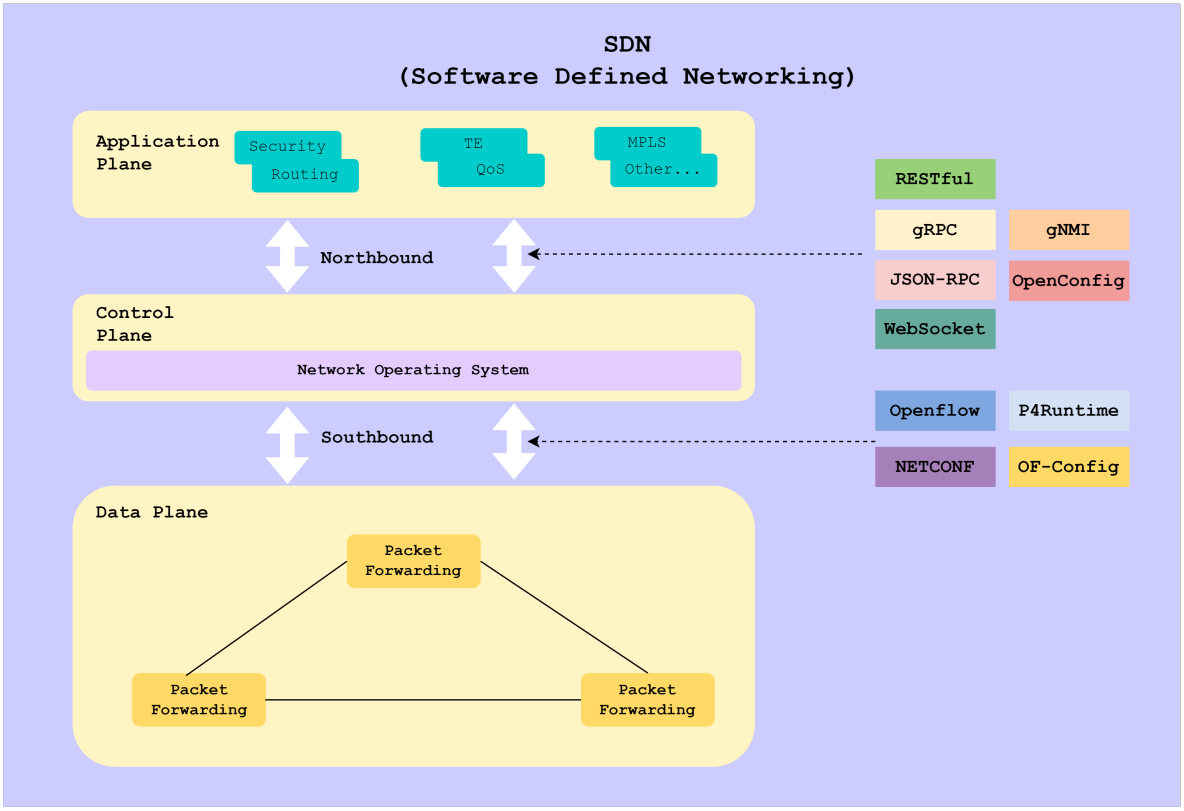

Figure 2.1: SDN Architecture

- **The Application Layer:** This layer is where business applications reside, interacting with the SDN controller via the northbound API to request network services and resources [18–22].
- **The Control Layer:** Occupying the central position is the SDN controller, the brain of the network that maintains a comprehensive view of the network state. It translates the requirements from the application layer into data plane configurations, managing the flow control across the network through the southbound API [18–22].
- **The Data Layer:** Also known as the infrastructure or forwarding layer, it consists of network devices such as switches and routers that are responsible for the actual movement of packets based on the configurations received from the controller [18–22].

The communication between these layers is facilitated by defined protocols, such as OpenFlow for the southbound API, which enables the controllers to direct traffic through the network switches dynamically [18–22].

2.1.3 Benefits and Challenges of SDN

SDN technology offers many benefits, some of them being **agility**, where network administrators can rapidly adjust network resources and policies to respond to varying workloads; **programmability**, which allows developers to customize network behavior to a granular level through software; and **centralized control**, which simplifies network design and operation by providing a single point of control for the entire network [20, 23–26].

However, with these benefits come significant challenges. The centralized nature of SDN poses potential **security risks**, as the control plane becomes a lucrative target for attackers seeking to compromise

the network. Additionally, **implementation complexities** can arise when integrating SDN into existing network infrastructures that are often heterogeneous and complex. These challenges must be carefully navigated to harness the full potential of SDN technologies [20, 23–26].

2.2 OpenFlow Protocol

As the most representative standard in SDN, the OpenFlow protocol has been pivotal in facilitating the communication between the control plane and the data plane. It defines a standardized way for the SDN controller to interact with the switch forwarding plane, thus enabling granular control over the network traffic routing and policies [27–31].

2.2.1 Fundamentals of OpenFlow

OpenFlow allows dynamic and programmable networking by providing a protocol through which the SDN controller can modify the flow table in network switches and routers. This capability is fundamental to the SDN paradigm, enabling the separation of the system that makes decisions about where traffic is sent (the controller) from the underlying systems that forward traffic to the selected destination (the switches and routers). OpenFlow supports various features, including defining flow entries to control packet forwarding, grouping multiple flows for easier management, and providing a communication channel for the controller to push changes to the network devices [27–31].

Field	Type	Description
version	uint8_t	The version field indicates the OpenFlow protocol version. The OFP_VERSION constant defines the specific version being used.
type	uint8_t	The type field specifies the kind of message being sent, with various constants (OFPT_*) representing packet-in, packet-out, flow-mod, and other message types.
length	uint16_t	This field gives the total length of the OpenFlow packet header and the payload in bytes, enabling the receiver to know how much data to process.
xid	uint32_t	A transaction identifier associated with this packet. It is primarily used to match messages and responses between a controller and a switch to facilitate proper pairing and sequence of operations.

Table 2.1: OpenFlow Packet Header Structure

2.2.2 OpenFlow in SDN Environment

In an SDN environment, OpenFlow serves as a holding element for network automation and intelligence. By allowing the controller to direct traffic through network switches, OpenFlow facilitates real-time adjustments to network topology and traffic flows, thereby enabling responsive and adaptive network configurations. The controller can reroute traffic, manage bandwidth, and implement network-wide policies, all through the centralized logic enabled by OpenFlow. This dynamic traffic management capability makes possible critical SDN applications, from load balancing and network virtualization to intrusion detection and energy-efficient networking [27–31].

In table 2.1, we can see the OpenFlow header that is sent between the Data plane and the Control Plane, the fields present in the header will tell the controller and switch how to operate in regards to the packet in evaluation [27–31].

2.2.3 Limitations and Extensions of OpenFlow

Despite its significant role in SDN, OpenFlow is not without limitations, particularly regarding its native support for the application-layer protocols essential for DPI. By design, OpenFlow focuses on the lower layers of the network stack, which constrains its efficacy for DPI that requires visibility into the application layer (Layer 7). This limitation has prompted the development of extensions and the exploration of alternative protocols/solutions that enhance OpenFlow's capabilities or circumvent its limitations. For instance, solutions such as P4 can be used to extend OpenFlow's match-action paradigm to support more functionalities. Similarly, solutions like OpenState or extensions that utilize eBPF (Extended Berkeley Packet Filter) for the Linux kernel have been proposed, offering the flexibility to potentially process packets at higher OSI layers [32–35]. These enhancements aim to empower OpenFlow to support sophisticated network functions, including but not limited to DPI, thereby augmenting its utility in contemporary network management and security contexts.

2.3 P4 Language

The P4 language (Programming Protocol-independent Packet Processors) represents a paradigm shift in how network devices process packets [36–39]. This domain-specific language enables programmers to specify how switches, routers, and other network devices process packets. Its design caters to the evolving needs of networks, including the growing demand for more flexible and programmable networks capable of supporting sophisticated data processing and analysis functionalities.

2.3.1 Principles of P4

P4 allows network engineers and researchers to describe packet forwarding behavior independently of the underlying hardware, promoting a new level of control over the data plane [36–39]. This language facilitates the precise definition of how packets are processed by the forwarding elements, including the parsing of packet headers and the execution of packet-processing algorithms. Unlike traditional networking where devices come with fixed functions, P4 makes it possible to program network devices to perform custom operations, adapt to new protocols, and modify their behavior dynamically based on the network requirements.

2.3.2 P4 and Data Plane Programmability

By enabling programmability directly within the data plane, P4 substantially enhances the flexibility and functionality of network infrastructures [36–39]. This programmability allows for the implementation of dynamic network policies and the ability to adapt traffic management strategies in real time. For instance, with P4, networks can efficiently implement load balancing, perform packet replication for multicast, and enforce security policies through dynamic access control lists. The granular control afforded by P4 over packet processing opens up new possibilities for optimizing network performance, implementing complex network functions, and rapidly prototyping new protocols without hardware changes.

2.3.3 P4 in the Context of DPI

In the realm of Deep Packet Inspection (DPI), the P4 language offers a transformative approach to analyzing and managing network traffic [40]. DPI, which involves examining the data part and the header of a packet as it passes an inspection point, requires significant flexibility and performance from the data plane [40]. With P4, network devices can theoretically be programmed to extract and inspect

specific packet fields beyond the traditional Layer 2 to Layer 4 headers, enabling DPI at Layer 7 [40]. This capability allows for the identification of application-level protocols, the monitoring of content for security purposes, and the enforcement of policies based on the actual data within packets [40]. Furthermore, the programmable nature of P4 enables the implementation of sophisticated DPI functions that can adapt to emerging threats and application behaviors, providing a powerful tool for enhancing network security and performance in SDN environments [40].

2.4 Protocol Buffers and gRPC

The modern network demands more efficient communication protocols, both in terms of performance and development workflow. Google's Protocol Buffers and gRPC are two technologies that respond to these demands by enabling fast, type-safe, and scalable communication between network components [41, 42].

2.4.1 Introduction to Protocol Buffers

Protocol Buffers, commonly known as Protobuf, is Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data, similar to XML or JSON. However, unlike these well-known data interchange formats, Protobuf is designed to be more efficient, both in terms of encoding size and speed of encoding/decoding operations. It enables developers to define data structures in a specialized language and then use generated source code in various languages to easily write and read the structured data to and from multiple data streams, while also providing robust backward and forward compatibility[41].

2.4.2 gRPC in Network Communications

gRPC is a high-performance, open-source universal RPC (Remote Procedure Call) framework that uses Protobuf as its interface definition language (IDL). It leverages HTTP/2 for transport, Protobuf for message serialization, and provides features such as authentication, load balancing, and blocking or non-blocking bindings. In the context of SDN, gRPC facilitates efficient and scalable communication between the SDN controller and network devices. With its support for bidirectional streaming and flow control, gRPC is particularly well-suited to handle the high volume and rapid exchange of messages that characterize control-plane communications in SDN[42].

2.4.3 gRPC's Role in SDN

Within an SDN architecture, gRPC can play a transformative role, particularly in facilitating communication between distributed control-plane services. It offers a more sophisticated alternative to traditional RESTful APIs by providing lower latency and higher throughput in communication. gRPC also supports server-side push, which aligns with the event-driven nature of SDN where the control layer needs to be promptly informed about changes in the network state[43]. Moreover, gRPC's strong type safety, when coupled with Protobuf's schema enforcement, ensures that the communication contracts between different network components are strictly adhered to, reducing the likelihood of miscommunication and related errors[44].

2.5 Data Plane Programmability

Data plane programmability represents a a very important shift in networking, making an era where the behavior of network devices can be dynamically programmed to meet specific network requirements. This capability is central to the advancement of Software-Defined Networking (SDN), allowing for unprecedented flexibility and control over how traffic is managed and processed within the network [45, 46].

2.5.1 Importance in SDN

In the context of SDN, programmability in the data plane has many advantages for several reasons. Firstly, it enables the implementation of custom traffic handling and forwarding logic, allowing networks to adapt to new protocols and services without requiring hardware upgrades. This flexibility is crucial for supporting the rapid evolution of network applications and services. Secondly, programmable data planes allow for more granular and dynamic network control, facilitating the creation of sophisticated traffic management policies that can be adjusted in real-time based on network conditions. This adaptability enhances network efficiency, security, and performance, aligning with the core objectives of SDN to create more agile and responsive networks.

Furthermore, data plane programmability empowers network operators to innovate and experiment with novel network functions and services directly in the network fabric, accelerating the deployment of new features and capabilities. This level of control and customization is essential for meeting the diverse and evolving needs of users and applications in modern networks.

2.5.2 Technologies Enabling Data Plane Programmability

Several technologies have emerged as enablers of data plane programmability, each contributing to the evolution of SDN in unique ways. The P4 language is at the forefront, providing a powerful and flexible platform for specifying packet processing behavior in network devices. By abstracting hardware specifics, P4 enables the definition of complex forwarding logic that can be deployed across a variety of hardware platforms, promoting innovation and interoperability in network device programming[47].

Beyond P4, other technologies play critical roles in enabling programmable data planes. Network processors, which are specialized hardware capable of executing complex packet processing algorithms at high speeds, provide the necessary computational resources for implementing programmable network functions. Additionally, Field-Programmable Gate Arrays (FPGAs) and Smart Network Interface Cards (SmartNICs) offer flexible and high-performance platforms for deploying custom packet processing logic, further expanding the possibilities for data plane programmability[48].

Together, these technologies form the foundation of programmable data planes in SDN, enabling a shift towards more intelligent, adaptable, and efficient networks. As these technologies continue to evolve, they will undoubtedly unlock new capabilities and opportunities for innovation in SDN, driving the future of networking towards even greater levels of programmability and control.

2.6 Packet and Bit-level Analysis

Understanding packet structures and the operations performed at the bit level are fundamental for Deep Packet Inspection (DPI) in managing and securing network traffic efficiently.

2.6.1 Understanding Packet Structures

Packets, the basic units of communication in a network, consist of headers and payloads. Headers contain metadata about the packet, such as source and destination addresses, protocol type, and more, while the payload carries the actual data being transmitted. Framing protocols define the structure of packets for different network layers, ensuring that data can be correctly processed and routed across the network. DPI leverages this structured approach to analyze packets beyond the surface level, examining both header and payload content to make informed decisions about the nature of the traffic, such as identifying application types, detecting malware, or enforcing policies [45, 46].

2.6.2 Bit-level Operations for DPI

DPI involves complex bit-level operations to parse and examine the contents of each packet. This includes extracting specific fields from the packet header, inspecting payload signatures, and applying pattern-matching techniques to identify protocols, applications, or malicious content. By manipulating bits directly, DPI systems can perform these analyses with high precision and efficiency, enabling real-time decision-making regarding traffic handling, prioritization, blocking, or redirection based on the detailed inspection of packet contents. In this thesis, we are leveraging all the previous mechanisms to validate our hypotheses and provide our results.

Chapter 3

Related Work

This chapter makes a summary about some studies that pave the way for leveraging Software-Defined Networking (SDN) technologies for Deep Packet Inspection (DPI). These works collectively stress the transformative potential of SDN and DPI integration. Each selected study is closely examined to elucidate its contributions, methodologies, and findings in the context of DPI implementation, followed by a comparative analysis with the objectives and methodologies of this thesis.

3.1 Deep Packet Inspection in P4 using Packet Recirculation

Gupta et al. [5] in their pioneering work "DeeP4R: Deep Packet Inspection in P4 using Packet Recirculation" introduce a novel approach to implement DPI within P4 networks by leveraging packet recirculation. They demonstrate that it's feasible to inspect packet payloads deeply without significant performance penalties by recirculating packets through the processing pipeline for additional examination. This method effectively bypasses the limitations of traditional DPI mechanisms that primarily focus on header data, thereby enabling a more thorough analysis of packet payloads. However, while Gupta et al. focus on exploiting P4's capabilities for DPI through recirculation, this thesis aims for a broader integration of DPI functionalities across SDN components. Specifically, it explores leveraging not only P4 but also OpenFlow extensions and custom controller developments for comprehensive DPI, and validates whether P4 could offer significant advantages over OpenFlow solutions.

3.2 DPI-Based IPS for Security of P4 Network Data Plane

Ahad et al. [11] presented "DPIDNS:A Deep Packet Inspection Based IPS for Security Of P4 Network Data Plane," focusing on enhancing network security through DPI-enabled Intrusion Prevention Systems (IPS) within P4 programmable networks for DNS traffic. Even though this paper offers a solution to examine DNS traffic in the Data plane, it does not offer a border perspective about P4 as a tool for Deep Packet Inspection over other popular applications like HTTP or SQL as we intend to do it. Besides, it does not compare it to OpenFlow solutions to test P4's benefits over OpenFlow-based solutions.

3.3 Practical Deep Packet Inspection in the Data Plane

In "DeepMatch: practical deep packet inspection in the data plane using network processors," Hypolite et al. [49] propose DeepMatch, a system for implementing DPI directly within the data plane at

line rates using network processors. This research underscores the feasibility of conducting sophisticated DPI within the constraints of the data plane, a crucial consideration for real-time traffic analysis and decision-making. While Hypolite et al.'s DeepMatch system showcases the technical potential for in-data-plane DPI processing, this thesis differentiates itself by exploring DPI's integration within a broader SDN framework that includes not just data plane programmability but also the strategic use of OpenFlow and a custom controller for DPI. Additionally, this thesis places a significant emphasis on quantitatively evaluating the system's performance, including metrics like throughput, accuracy, and resource utilization, which are essential for practical deployments.

3.4 Application-Aware Traffic Control via DPI

Li et al. [50] in their study on "Deep Packet Inspection Based Application-Aware Traffic Control for Software Defined Networks," explore the utilization of DPI for dynamic traffic management based on application-level insights. Their work highlights the potential of DPI in facilitating application-aware network policies that can significantly enhance QoS and user experience. This approach mirrors the thesis's objective of employing DPI to achieve a nuanced understanding of traffic flows for better network management. However, this thesis goes further by integrating DPI with software-defined technologies across both the control and data planes, offering a comprehensive framework for DPI that includes detailed performance evaluation and the exploration of DPI for network optimization in addition to traffic control.

3.5 Selective Packet Inspection for DoS Attack Mitigation

Chin et al. [51] investigate the application of SDN principles to enhance network security, specifically targeting DoS (Denial of Service) flooding attacks. Their approach, utilizing selective packet inspection in conjunction with SDN's centralized control capabilities, proposes a collaborative mechanism between distributed monitors and a central controller to efficiently detect and mitigate TCP SYN flood attacks. This mechanism leverages the agility of SDN to dynamically inspect packets and adjust network configurations in real-time, aiming to balance the workload on network devices and achieve prompt attack detection and mitigation.

While Chin et al.'s study [51] presents a compelling use case of SDN for security enhancement, it fundamentally diverges from the objectives of this thesis in several key aspects:

- **Scope of DPI:** Chin et al. focus specifically on the detection of DoS flooding attacks, primarily utilizing packet volume and pattern anomalies as indicators. In contrast, this thesis explores DPI's broader capabilities within SDN, examining packet payloads for a comprehensive range of attack vectors, not limited to DoS. The intricate payload inspection aims to uncover not only volumetric anomalies but also sophisticated attack signatures embedded within the traffic.

- **Nature of Attacks:** DoS attacks, while disruptive, represent a specific subset of network threats characterized by their volume-based attack patterns. This thesis, however, delves into the nuances of payload-based threats, which encompass a wider variety of attack types, including but not limited to malware distribution, data exfiltration, and advanced persistent threats (APTs). The difference in attack nature necessitates distinct detection and mitigation strategies, with payload-based inspection demanding deeper analytical capabilities.

- **Methodological Approach:** The methodology employed by Chin et al. is heavily predicated on the collaboration between network monitors and the central controller, with a significant emphasis

on mitigating the detected threats. Conversely, this thesis proposes a comprehensive framework that integrates OpenFlow, P4runtime, and a custom controller for DPI, emphasizing not only detection and mitigation but also the performance evaluation of DPI across various metrics such as delay, jitter, packet loss, CPU, and memory consumption.

In summary, while Chin et al.'s work [51] provides valuable insights into leveraging SDN for DoS attack mitigation through selective packet inspection, the objectives and scope of this thesis extend beyond this focus. By examining payload data across a broader spectrum of attack types and employing an integrative framework of SDN technologies for DPI, this research aims to enhance network security, management, and optimization in more complex and varied network environments.

Overall, the discussed works collectively contribute to the foundational knowledge and technical capabilities necessary for advancing DPI within SDN contexts by presenting unique methodologies and discusses specific areas, from P4-based packet recirculation to IPS-focused security enhancements and practical DPI implementation strategies in the data plane. This thesis builds upon these contributions by proposing an integrative framework that leverages the strengths of SDN technologies for comprehensive DPI implementation. It not only aims to enhance network security and management but also rigorously evaluates the impact of DPI on various network performance metrics, setting a path for future research and deployment in increasingly complex network environments. Besides, we are looking to clearly compare P4-based vs OpenFlow-based solutions with and without a Control Plane component to validate or refute our hypotheses.

Chapter 4

Methodology

This thesis aims to systematically explore diverse methodologies for implementing Deep Packet Inspection (DPI) within Software-Defined Networking (SDN) infrastructures. Through meticulous experimentation and analysis, we intend to highlight the benefits and potential of various DPI techniques tailored to SDN ecosystems.

4.1 DPI Implementation via OpenFlow and Open vSwitch

4.1.1 Controller-Based DPI Approach

The first approach uses the OpenFlow protocol in tandem with Open vSwitch. Open vSwitch operates as a multifaceted virtual switch that relies on OpenFlow for its functional operations. In this paradigm, DPI is conducted predominantly at the controller level, owing to the limitations inherent in the data plane when utilizing OpenFlow and analogous protocols. Specifically, data plane layers governed by such protocols offer a static set of variables, limiting the creation and processing capabilities within flow tables to pre-defined network headers like Ethernet, IP, ARP, TCP, etc.(Protocols from layers 2 - 4). Similarly, the actions emanating from these flow tables are circumscribed by the predefined functions of the data plane.

Considering the necessity for DPI to parse dynamic-sized headers, such as those found in HTTP or another L7 protocol payload, the data plane's processing abilities of Open vSwitch are deemed inadequate. Consequently, this approach necessitates the redirection of pertinent traffic to the controller—specifically, a POX controller—wherein the traffic undergoes comprehensive analysis, leading to subsequent action based upon the processed information.

4.1.2 OpenFlow's Constraints on Dynamic Header Processing

The static nature of OpenFlow's data plane processing poses challenges in managing dynamic-size headers like HTTP with variable-length payloads. To circumvent this, our methodology involves rerouting suspect traffic to a higher-level processing entity capable of dynamic analysis.

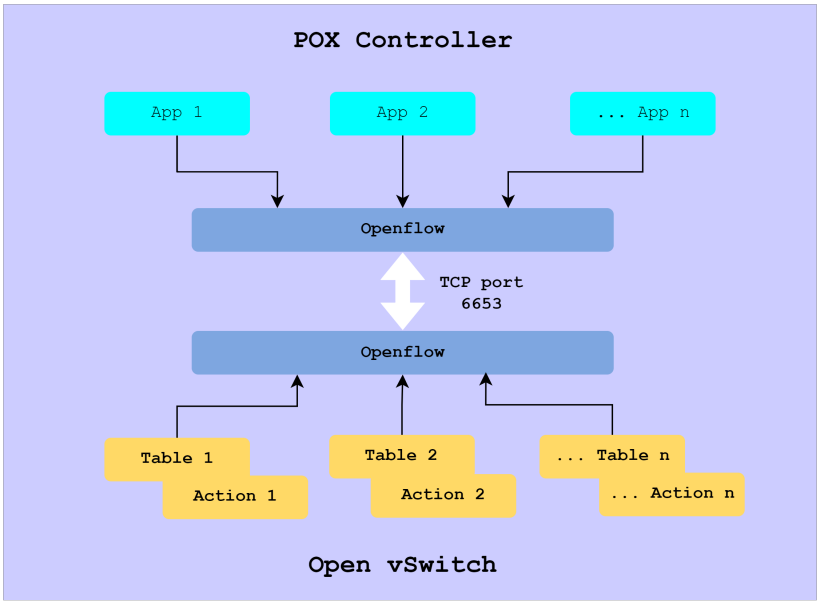

Figure 4.1: DPI implementation with OvSwitch, Openflow, and POX controller

4.2 Integration of BMv2 and P4runtime for DPI

4.2.1 P4-Enabled Data Plane Customization

Our secondary methodology leverages the BMv2 switch and P4runtime API. P4 offers substantial enhancements over OpenFlow-based approaches by granting the capability to custom-code the data plane, therefore facilitating the construction of flow tables that meet the requirements of DPI tasks. In this instance, DPI persists within the control plane; however, the transportation of data from the BMv2 switch to the control plane discards OpenFlow in favor of P4Runtime(which uses gRPC and protobuffers). The control plane, equipped with a bespoke controller, is responsible for parsing and processing this data, followed by action execution after the pertinent analysis.

4.2.2 Distinction Between Control and Data Plane Processing

While similar to the first approach in terms of control plane DPI processing, this method differentiates itself through its employment of P4 for data plane programming, usage of gRPC for data transport, and implementation of a custom-designed controller.

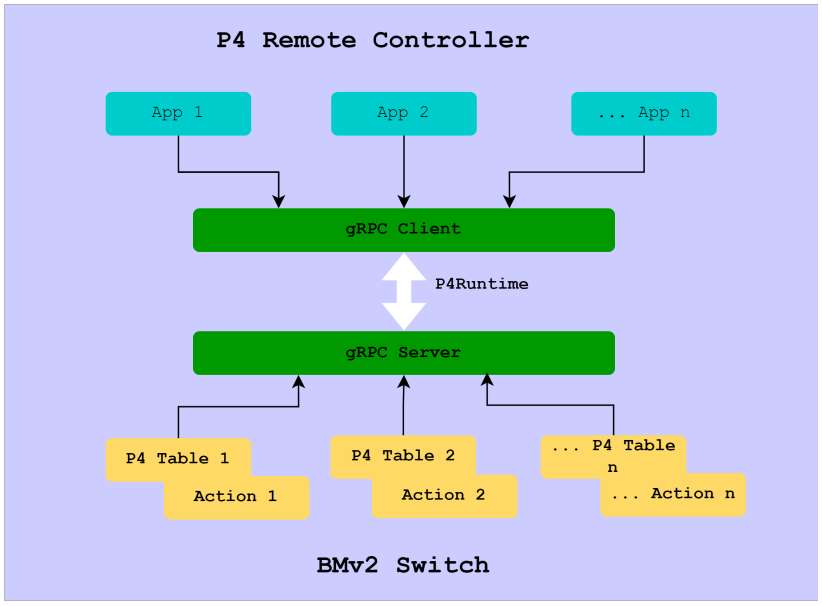

Figure 4.2: DPI implementation with BMv2, P4Runtime, and P4 controller

4.3 Data Plane-Centric DPI Strategy

4.3.1 In-situ Packet Analysis via P4

The third and final approach within our research project also utilizes the BMv2 switch alongside the P4 language; however, it fundamentally diverges in its execution of DPI. Instead of dispatching traffic for inspection to the control plane, DPI is executed in-situ within the data plane. This inline processing entails monitoring bit patterns corresponding to the protocol under analysis and executing predefined actions directly within the data plane, avoiding the need for control plane involvement.

4.3.2 Protocol-Specific Bit Tracking

By tracking protocol-specific bit patterns and executing DPI within the data plane, this method aims to optimize the inspection process and minimize latency introduced by controller-based analyses.

4.3.3 Thrift API for Data Plane Configuration

The Thrift API plays an instrumental role in the direct configuration and management of the BMv2 data plane used for DPI. It is a software framework for scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, and other languages.

Thrift in Network Device Configuration

In the context of P4 and BMv2, Thrift provides a set of interfaces generated from a common Thrift definition. This allows for remote procedure calls to configure various aspects of the switch, including table entries, from a control application written in a high-level language such as Python.

Table Entry Management with Thrift

Table entries dictate how packets are processed within the switch. Using Thrift, network developers can programmatically add, modify, and delete these entries, tailoring the data plane behavior to the desired DPI objectives. Through these interactions, Thrift ensures that DPI policies are enforced directly on the data plane with precision and agility. Through the application of the Thrift API, data plane-centric DPI becomes a dynamically adaptable and high-performance solution that aligns with the evolving landscape of network security challenges.

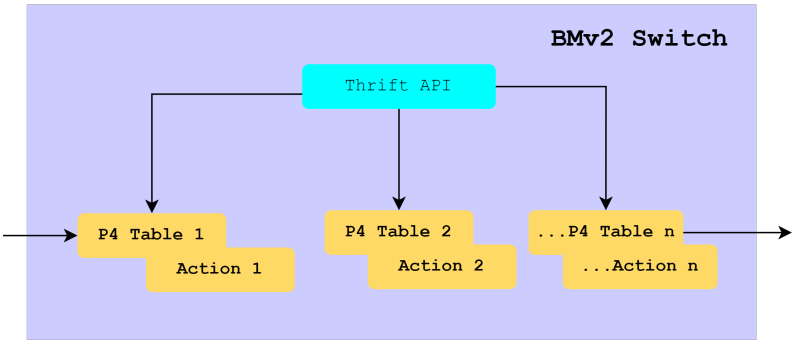

Figure 4.3: DPI implementation on Data plane only

4.4 Synthesis of DPI Methodologies for SDN Advancement

4.4.1 Facilitating Accelerated DPI Learning and Development

The proposed methodologies are intended to serve as instrumental tools for scholars and practitioners, expediting the learning curve and fostering the development of DPI solutions within SDN contexts.

4.4.2 Comparative Analysis of DPI Efficacies

A comparative examination of metrics such as inspection delay, accuracy, precision, recall, F1 score, and true and false positive rates is integral to this research. The outcomes of these measurements will critically inform the efficacy and performance viability of each DPI approach, guiding the identification of superior methodologies in terms of both performance and precision and finally validating or refuting our hypotheses.

4.5 Laboratory Configuration for Experimental Testing

4.5.1 Virtualized Testing Environment

For the experimental validation of our DPI methodologies within an SDN context, we utilized a virtualized environment to simulate the necessary network components, such as switches and controllers(see 4.4). Our virtualization platform of choice was VMware Workstation 16 Pro, version 16.2.3 build-19376536, serving as the hypervisor underpinning the laboratory setup. On VMware we set up two Ubuntu machines to carry out our tests.

4.5.2 Data Plane Emulation

Within the first Ubuntu operating system instance, we established the data plane using Mininet, a versatile network emulator, to create a representative network topology for our investigative purposes. The environment included virtual switches, specifically Open vSwitch and BMv2, and a network of interconnected hosts configured to facilitate client-server communications fundamental to DPI tests.

The specifications for the Ubuntu OS employed for the data plane emulation were as follows:

- OS: Ubuntu 22.02
- Memory: 4GB
- Processors: 2
- Hard Disk: 40GB
- Network Adapter: NAT
- IP Configuration: 192.168.30.149/24 with Gateway 192.168.30.1

To ensure reproducibility and foster future research, an OVF image of this virtual machine is archived and made available, encapsulating the configuration state used throughout our experiments.

4.5.3 Control and Application Plane Configuration

The second Ubuntu operating system instance was dedicated to hosting the control and application planes. The control plane was powered by the POX controller, an OpenFlow-compatible platform known for its simplicity and seamless integration with the Python programming language, and for the integration of our P4-based implementation, we employed a proprietary controller developed atop the p4_runtime.shell library.

The specifications for the Ubuntu OS designated for the control and application planes were as follows:

- OS: Ubuntu 22.02
- Memory: 4GB
- Processors: 2
- Hard Disk: 35GB
- Network Adapter: NAT
- IP Configuration: 192.168.30.148/24 with Gateway 192.168.30.1

Similar to the data plane instance, an OVF image of this setup is provided to support future expansion of our research and to facilitate independent verification of our experimental results.

4.6 Experimental Validation of DPI Configurations

Upon the successful deployment and development of the DPI configurations, as delineated in Figure 4.4, this study undertakes a rigorous evaluation using HTTP and SQL application protocols as primary vectors. It is pertinent to note that this investigation restricts its focus to unencrypted traffic to circumvent the complexities associated with encrypted data, which falls outside the purview of this thesis.

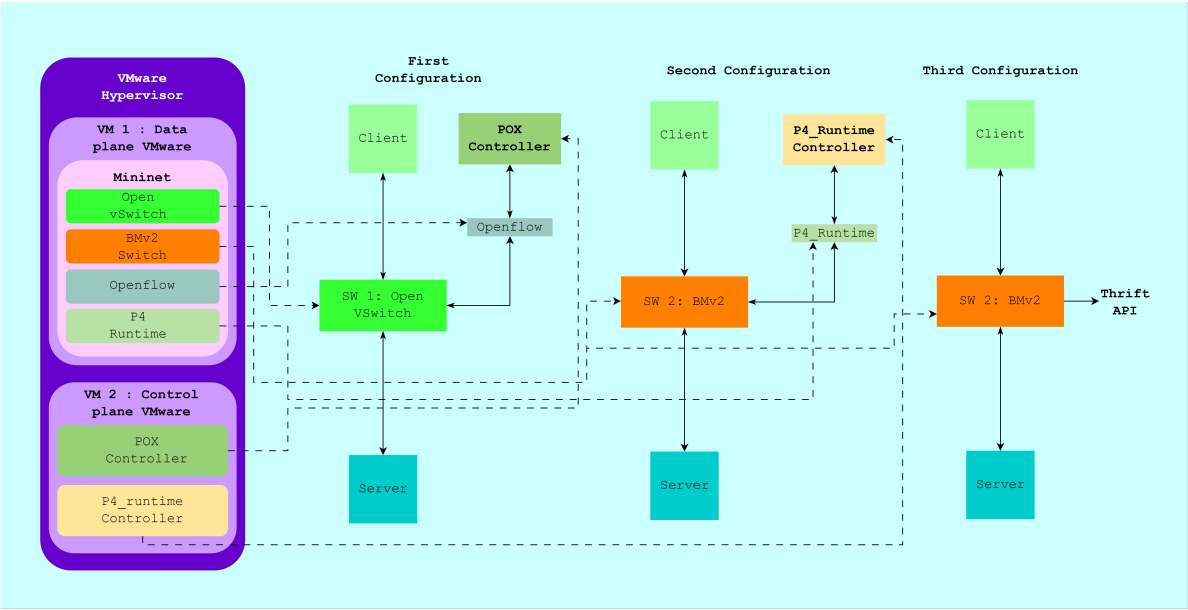

Figure 4.4: Virtualization utilized for testing our different configurations

4.6.1 HTTP Protocol Evaluation

The examination of HTTP traffic involves the orchestration of a server-client setup, The goal of this evaluation is to build a URL filtering solution to filter out domains or URLs that we consider unsafe.

Metrics for Evaluation: The evaluation metrics include accuracy, precision, recall, and F1 score, alongside the True Positive Rate (TP) and False Positive Rate (FP). These metrics will furnish insights into the DPI's proficiency in identifying potential security threats. The assessment will also encompass network performance metrics such as delay to complete the transmission of the HTTP GET method, additionally we will evaluate the resources consumption by our tests.

4.6.2 SQL Protocol Evaluation

For the SQL protocol, the experimental setup comprises a SQL server connected with a client machine. This setup facilitates the creation of databases and tables on the server, while the client is tasked with executing queries and attempting SQL injection attacks. The DPI's objective is to prevent such injection attempts effectively.

Metrics for Evaluation: Similar to the HTTP tests, the evaluation for SQL tests will also leverage accuracy as key metric. These will provide a quantitative measure of the DPI's accuracy in classifying and mitigating attack vectors. Performance metrics, including delay, and latency, will be scrutinized to ascertain the DPI solution's operational efficiency, lastly, like for our URL filtering mechanism, here we will also measure the resource consumption by our tests.

Chapter 5

Development of Components

This chapter delineates the systematic development of three distinct configurations, each tailored to explore the capabilities and limitations of Deep Packet Inspection (DPI) within Software-Defined Networking (SDN) environments. The configurations are designed to facilitate a comprehensive understanding of DPI's applicability across various network architectures and protocols.

5.1 DPI Implementation via OpenFlow and Open vSwitch (Open flow-based Implementation)

The initial phase of our exploration commenced with the deployment of a basic network topology using Mininet, a versatile network emulator. This topology comprised a singular switch connected to two host machines, one designated as a client and the other as a server, to simulate client-server interactions. The command utilized for this setup is as follows:

```
sudo mn --controller remote,ip='192.168.30.148' --switch ovs,
protocols=OpenFlow1.0
```

Command Explanation: This command instantiates a Mininet environment with a remote controller specification, where ''ip='192.168.30.148'' denotes the controller's IP address. The ''–switch ovs,protocols=OpenFlow1.0'' argument configures the network to use Open vSwitch (OVS) with OpenFlow version 1.0 as the communication protocol between the switch and the controller, we used OpenFlow v1.0 for compatibility reasons across all the dependencies for this project.

In the absence of a specified controller IP, Open vSwitch operates as a conventional Hub broadcasting the packet by all the ports except the one from which the packet was received. However, in our configured scenario, the switch is programmed to forward all incoming traffic to the controller. This is achieved through the transmission of `packet_in` messages, which the controller receives, processes, and responds to with appropriate actions encapsulated in `packet_out` messages. The previous command can be run after just installing mininet, however, for our tests we run Python scripts to tailor the Mininet topology to out needs.

The core of our controller logic resides within the POX framework, chosen for its simplicity and Python programming compatibility. The following is a conceptual representation of the algorithm used to process packets arriving at the switch:

Controller's Deep Packet Inspection Algorithm for Processing Packets:

1. Listen for `Packet_In` events from the switch.

2. Upon receiving a packet, determine if it is an IP packet; if not, proceed with normal processing.

3. For IP packets, further identify if they encapsulate TCP segments. Non-TCP packets are processed normally.

4. For TCP packets, particularly those on destination port 80 (HTTP) and 3306 (SQL), attempt to decode the TCP payload to interpret the traffic (either HTTP or SQL).

5. Conduct a series of checks within the payload for predefined conditions, such as host matching or command matching for the presence of blacklisted terms.

6. Depending on the outcome of these checks, either drop the packet or continue with standard processing, which includes flooding the packet to all ports except the incoming port(DEFAULT_ACTION).

Algorithm 1 Packet Processing in DPI via OpenFlow and POX Controller

```
Require: Incoming Packet: packet, Controller IP: controller_IP
Ensure: Proper Action for Packet based on DPI
    //First we decode the payload
 1: function DECODE_PAYLOADpacket
 2:     return Decoded payload of packet
 3: end function
    //Then we look for applications HTTP and SQL and if that happends, we proceed to invoke our
    DPI application which is imported as dependency
 4: procedure HANDLE_PACKET_IN(packet)
 5:     if packet.type = IP_TYPE then
 6:         IP_Packet ← packet.payload
 7:         if IP_Packet.protocol = TCP_PROTOCOL then
 8:             TCP_Segment ← IP_Packet.payload
 9:             if TCP_Segment.dstport = 80 or TCP_Segment.dstport = 3306 then
10:                 http_Payload ← DECODE_PAYLOAD(TCP_Segment.payload)
11:                 sql_Payload ← DECODE_PAYLOAD(TCP_Segment.payload)
12:                 if contains_Blacklisted_Terms(http_Payload or sql_Payload) then
13:                     return DROP_ACTION
14:                 end if
15:                 return FORWARD_ACTION
16:             else
17:                 return DEFAULT_ACTION
18:             end if
19:         else
20:             return DEFAULT_ACTION
21:         end if
22:     else
23:         return DEFAULT_ACTION
24:     end if
25: end procedure
    //Invoking the main function and running the program
26: procedure MAIN
27:     Listen for packet_in messages
28:     action ← HANDLE_PACKET_IN(packet)
29:     Execute action on the switch
30: end procedure
```

This DPI implementation strategy via OpenFlow and Open vSwitch sets the groundwork for subsequent configurations, aiming to elevate the comprehension and application of DPI in SDN contexts. In algorithm 1 we can see the algorithm that we used for our tests with Openflow for better understanding.

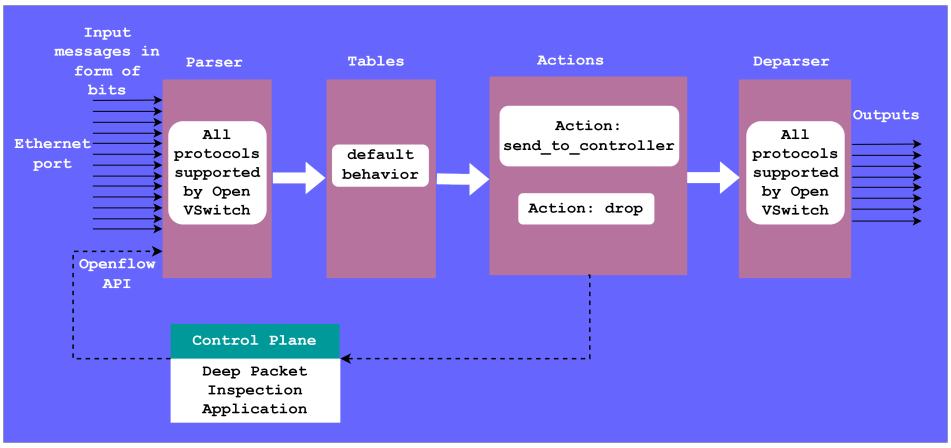

Figure 5.1: Pipeline to process packets arriving at the switch

In figure 5.2 we show a captured packet that represents the interaction between data plane and control plane involving OpenFlow protocol, specifically an `OFPT_PACKET_IN` message, alongside network layer (IP) and transport layer (TCP) headers encapsulating HTTP traffic. Below we have a detailed description of elements within the frame:

- **Ethernet II Header**: This layer identifies the source and destination MAC addresses, indicating the frame's traversal between two endpoints on a local network segment.
- **Internet Protocol Version 4 (IPv4)**: The IP header specifies the source and destination IP addresses, enabling the routing of packets across IP networks.
- **Transmission Control Protocol (TCP)**: The TCP header details the source and destination ports (indicative of the service in use, in this case, HTTP), sequence and acknowledgment numbers essential for data ordering and reliability, and the length of the TCP segment.
- **OpenFlow Header**: OpenFlow, a fundamental protocol for SDN (Software Defined Networking), uses specific message types for communication between the controller and the switch. The header includes:
  - *Version*: The OpenFlow protocol version in use, depicted by `.0000 0001 = Version: 1.0 (0x01)`.
  - *Type*: The message type, here `OFPT_PACKET_IN (10)`, signals that this is a packet-in message typically sent by the switch to the controller when a packet arrives at a switch port and no matching flow entry is found.
  - *Length*: The total length of the OpenFlow message.
  - *Transaction ID (xid)*: A unique identifier for this OpenFlow transaction, allowing for request-response pairing between the switch and the controller.
  - *Buffer Id*: A field often used by the switch to indicate that the packet has been buffered and to reference the buffered packet within subsequent interactions.
  - *Total Length*: The length of the encapsulated packet that triggered the `OFPT_PACKET_IN` event.
  - *In Port*: The switch port number on which the original packet arrived.
  - *Reason*: The reason for sending the `OFPT_PACKET_IN` message, often due to the absence of a matching flow for the packet within the switch's flow table.

– *Pad*: A padding field to align the header size as per the protocol specification.

- **Encapsulated HTTP Traffic**: The TCP segment contains payload data associated with HTTP traffic, identifiable by the destination port 80, the standard port for HTTP communication. The encapsulation within the data plane implies that the traffic is subject to DPI based on the OpenFlow and Open vSwitch capabilities.

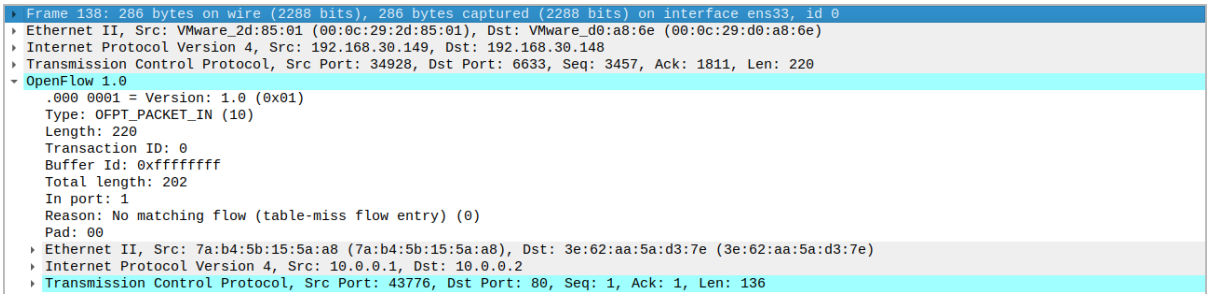

Figure 5.2: Packet_in message traveling from data plane to control plane

5.2 Integration of BMv2 and P4runtime for Deep Packet Inspection (P4runtime-based Implementation)

To explore the advanced capabilities of Deep Packet Inspection (DPI) within a Software-Defined Networking (SDN) framework, we extended the use of Mininet—a popular network emulator—to incorporate the Behavioral Model version 2 (BMv2) switch, which is emblematic of the P4 language's runtime environment. This modification enables a profound exploration of DPI's potential when interfaced with the granular control and flexibility offered by P4 programmability.

5.2.1 Customizing Mininet for BMv2 Integration

To facilitate the BMv2 switch's integration into our experimental topology, a customized class within the Mininet environment was developed. This adaptation allows for the instantiation of a P4-programmable switch within the traditionally Open vSwitch-dominated Mininet emulations. The algorithm 2 outlines the foundational steps taken to achieve this customization:

Algorithmic Description:

1. Instantiate a Mininet network with a remote controller specification, signifying the pivot towards a more flexible and programmable network emulation.
2. Introduce a BMv2 switch class, embedding gRPC support to enable real-time control and monitoring capabilities, crucial for DPI activities.
3. Dynamically add network components, including hosts and the BMv2 switch, thereby crafting a emulation environment reflective of real-world SDN topologies.
4. Configure network links and initiate the network, setting the stage for subsequent DPI-related experiments.

Algorithm 2 shows us the exact adaptation that we did in Mininet to use BMv2 and be able to use it with P4runtime.

Algorithm 2 Setting up a Mininet Network with BMv2 Switches

Require: None
Ensure: A Mininet network with BMv2 switch and gRPC support is created and started
//Initialization of BMv2 switch
function INITIALIZE_BMV2_SWITCHname, sw_path, json_path, thrift_port, grpc_port, pcap_dump
    Create BMv2Switch instance with provided parameters
    if json_path is provided **then**
        Prepare command with logging, debug level, gRPC server address
        if thrift_port is provided **then**
            Add thrift port to command
        end if
        if pcap_dump is enabled **then**
            Enable packet capture
        end if
        Attach switch interfaces
    end if
    Execute command and redirect output to log file
end function
//Creating network components(hosts, switches) and linking them
procedure CREATE_NETWORK
    Initialize Mininet with a remote controller
    Add a remote controller with specified IP and port
    Add a BMv2 switch to the network with gRPC support
    Add two hosts with static IP and MAC addresses
    Create links between hosts and the BMv2 switch
    Start the network
    Open Mininet CLI for interactive commands
    Stop the network upon CLI exit
end procedure
//Invoking functions and running Mininet
procedure MAIN
    Set logging level to information
    Invoke CREATE_NETWORK
end procedure

5.2.2 BMv2 Command Line Execution and P4 Program Loading

Following the establishment of the Mininet topology featuring the BMv2 switch, the next critical step involves initializing the switch with P4Runtime support and loading the P4 program for our data plane. The command and description to initiate the BMv2 switch is detailed below:

```
sudo simple_switch_grpc
     --log-file ss-log \
     --log-flush \
     --dump-packet-data 10000 \
     -i 0@s1-eth1 \
     -i 1@s1-eth2 \
     --no-p4 \
     --thrift-port 9090 \
     -- --cpu-port 510 &
```

`sudo simple_switch_grpc` This initiates the BMv2 switch executable with gRPC support, running it with superuser privileges to ensure sufficient permissions for network manipulation.

`--log-file ss-log` Specifies the creation of a log file, named `ss-log`, where operational logs of the switch will be stored. This facilitates debugging and analysis of the switch's behavior over time.

`--log-flush` Ensures immediate flushing of log outputs to the `ss-log` file, enhancing the real-time tracking of switch activities and anomalies.

--dump-packet-data 10000 Activates the dumping of packet data for packets processed by the switch, with a maximum size limit set to 10,000 bytes.

-i 0@s1-eth1 -i 1@s1-eth2 Attaches the switch's ports to specific network interfaces (`s1-eth1` and `s1-eth2`), effectively defining the physical or virtual connections through which the switch will receive and transmit network traffic.

--no-p4 Indicates that the switch will operate without loading a P4 program. For our tests, we are uploading manually.

--thrift-port 9090 Assigns port 9090 for Thrift connections, we used thrift to debug the packet coming in and out of the switch.

-- --cpu-port 510 Designates port 510 as the CPU port, a special port used for sending and receiving control messages directly between the switch and the controller. This setup is essential for processing Packet-In and Packet-Out messages that are fundamental to DPI tasks.

& Ensures the command runs in the background, allowing the terminal session to be used for other commands while the BMv2 switch continues operating.

In summary, this command intricately configures a BMv2 switch to support gRPC interfaces, equipping it with the necessary capabilities for extensive logging, packet data inspection, and specific interface assignments. Such configuration is integral to conducting sophisticated DPI analyses within a Software-Defined Networking (SDN) environment.

Upon executing the aforementioned command, the DPI-focused P4 program is loaded into the switch. This program shown in algorithm 3 encompasses the definitions and logic required to dissect and analyze network traffic at a granular level:

5.2.3 P4 Program Overview

The provided P4 program in algorithm 3 encapsulates a sophisticated DPI mechanism aimed at enhancing network security through deep packet inspection at various protocol layers. The program integrates multiple headers, including Ethernet,TCP and IPv4, to facilitate the precise analysis of traffic flows. Also, special headers such as `packet_out` and `packet_in` are crucial for managing DPI-related packet transfers between the network devices and the controller. To have a better understanding of the P4 pipeline we can observe figure 5.3 and take a closer look of this.

The information interchanged between the data plane and the control plane for this approach can be seen in figure 5.4, the captured packet in Frame 116 exemplifies a comprehensive communication sequence from the data plane to the control plane facilitated by P4Runtime. The packet's structure, as delineated in the capture, reveals an amalgamation of metadata and payload, embodying both the original packet information and additional control plane directives.

Packet Constituents

- **Ethernet II**: This header includes source and destination MAC addresses, providing essential layer 2 network information.
- **Internet Protocol Version 4**: IP headers comprise source and destination IP addresses, encapsulating the layer 3 routing data.

Algorithm 3 P4 Program for DPI with BMv2 and P4runtime

Require: None
Ensure: Detailed packet inspection and handling within the BMv2 switch
//P4 Header Definitions
Ethernet Header: Includes source and destination MAC addresses, and EtherType.
IPv4 Header: Contains fields like source and destination IP addresses, TTL, protocol type, etc.
TCP Header: Contains all the standard fields of a TCP header.
//Parser Implementation
procedure ParseEthernet
    Extract Ethernet header
    if EtherType is IPv4 **then**
        Proceed to parse IPv4 header
    else
        Accept packet as non-IPv4
    end if
end procedure
procedure ParseIPv4
    Extract IPv4 header
    Validate checksum and necessary fields
    Decide next parsing steps based on protocol field
end procedure
//Ingress Control Logic
procedure IngressProcessing
    Apply LPM for destination IP to determine routing
    Apply actions based on table matches (set port, drop, etc.)
    if Packet destined for CPU **then**
        Encapsulate and send to CPU port
    end if
end procedure
//Packet Operations
Define actions to modify packets, such as setting TTL, modifying ports, and handling metadata for packet-in and packet-out operations.
//Checksum and Queue Management
procedure ManageChecksums
    Verify and update IPv4 checksums before packet forwarding
end procedure
procedure QueueManagement
    Manage queue sizes and handle congestion
end procedure
//Deparser
procedure DeparsePacket
    Emit headers in order: Ethernet, IPv4
    Prepare packet for egress processing
end procedure
//Overall Switch Configuration
Combine parser, control logic, and deparser into a cohesive BMv2 model.
Utilize P4Runtime to dynamically manage the switch configuration.

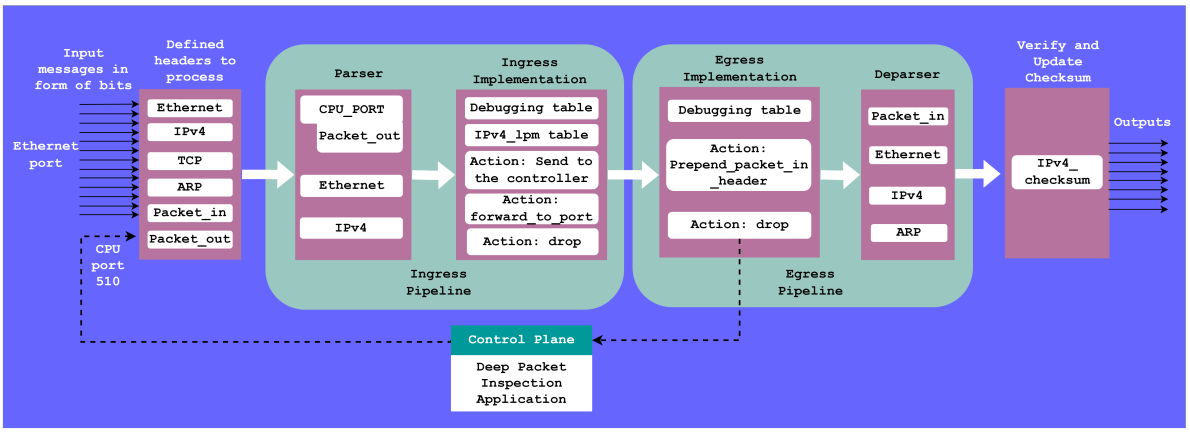

Figure 5.3: Pipeline for the integration of P4-based BMv2 switch with the control plane

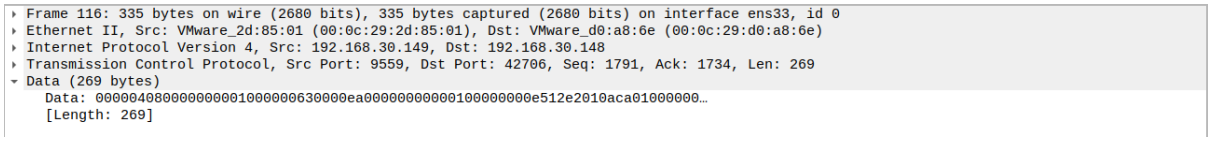

Figure 5.4: Packet_in message traveling from data plane to control plane

- **Transmission Control Protocol**: The TCP segment captures transport layer details such as source and destination ports, sequence and acknowledgment numbers, and segment length.
- **Data (269 bytes)**: The payload, encoded using Protocol Buffers—a language-agnostic binary serialization tool—encapsulates the actual data alongside metadata. Protocol Buffers ensure efficient, extensible encoding of structured data.

Data Plane to Control Plane Communication

The packet snapshot explicitly indicates a transition of data from the switch (data plane) to the controller (control plane), a process governed by P4Runtime. The packet's payload is bifurcated into:

- **Metadata**: This section encompasses information about the packet's journey within the network infrastructure, such as ingress port and timestamps, and is typically used by the control plane for decision-making and logging.
- **Original Packet Payload**: Retaining the integrity of the original packet, this segment aids the controller in determining the necessary actions, such as modifying table entries to handle similar future packets.

Data Encapsulation Protocol

The payload's encoding with Protocol Buffers substantiates its association with the control plane instruction set. It signifies a seamless communication standard where the controller can interpret and manipulate the data plane behaviors dynamically, based on the network policies and the packet's attributes.

In essence, this packet is a typical representation of a control message in an SDN environment, where P4Runtime operates as the conduit between high-level network objectives and low-level data plane functionalities.

5.2.4 Algorithm for P4Runtime-based DPI

The controller leverages the P4Runtime to interact with the programmable BMv2 switch, capable of dynamically injecting P4 rules and managing network traffic based on real-time analysis. Algorithm 4 outlines the steps involved in establishing the controller, monitoring network traffic, and responding to network events dynamically.

5.2.5 Libraries and Utilities Utilized in the DPI Controller

In the development of the P4Runtime-based DPI Controller, several specialized libraries and utilities are employed to facilitate the interaction with P4-enabled devices, packet crafting, and operational management. These libraries provide essential functions that enable the dynamic and efficient handling of network traffic for Deep Packet Inspection within a Software-Defined Networking environment.

Algorithm 4 Setup and Operation of P4Runtime-based DPI Controller

Require: None
Ensure: Real-time packet processing and dynamic network response

// Initialization of necessary libraries and data structures
Import necessary libraries and configure logging facilities.
Define global data structures to maintain state, mappings, and configurations.
// Establish communication with the P4Runtime, the controller behaves as **"P4Runtime Client"** opposite to how a controller normally behaves
procedure SET_UP(grpc_addr, p4info_txt_fname)
    Establish connection with the P4Runtime server using GRPC address.
    Load P4Info file to set up device and controller communications.
    Initialize mappings for enum types and metadata fields using utility functions.
    Prepare the PacketIn channel to receive packets from the switch.
end procedure
// Duplication of packets for analysis with our DPI applications
procedure WRITE_CLONE_SESSION(clone_session_id, port_list)
    Configure cloning sessions to duplicate packets for analysis, specifying egress ports.
end procedure
// Fetching the cloned packets and parsing with Scapy
procedure MONITOR_PACKETS
    Continuously fetch packet-in messages and process each packet based on the rules for DPI.
    Decode packets using Scapy to analyze headers and payload.
    Apply DPI logic to determine whether to modify, drop, or forward the packet.
    If action required, construct and send packet-out messages to modify flow directly.
end procedure
// Finishing session
procedure TEAR_DOWN
    Clean up and close connections, ensuring all sessions are terminated properly.
end procedure
// Function to clone, retrieve and parse packet in real time as the Data plane sends information for analysis on the control plane
Main Loop:
while True **do**
    Monitor incoming packets and handle according to DPI logic.
    Adjust flows and handle network events as they occur.
end while

- **logging:** This library is crucial for managing logging activities across the Python application. It aids in debugging and provides operational insights by recording runtime events and errors, which is invaluable in a network security context where real-time data analysis is crucial.
- **p4runtime_sh:** Provides the necessary Python bindings for interacting with the P4Runtime interface of network devices such as the BMv2 switch. This library allows the controller to send commands, modify device state, and manage network flows dynamically, essential for implementing DPI strategies.
- **scapy:** A powerful Python library used for crafting and interpreting packets at multiple layers of the network stack. Scapy enables detailed inspection and manipulation of network traffic, which supports the complex packet analysis required for DPI.
- **struct:** This module offers functionality to pack and unpack complex binary data formats, which is fundamental when working with network protocols and data structures. It ensures that the data extracted from or sent to the network is in the correct format, thus maintaining the integrity of DPI processes.
- **p4runtime_sh.shell:** Enhances the usability of the p4runtime_sh library by providing higher-level abstractions for P4 device management. This component simplifies the deployment and modification of P4 programs on the switch, facilitating more agile DPI rule updates and network responses.
- **p4runtime_shell_utils (p4runtime_shell_utils):** A utility library that streamlines common tasks associated with the use of P4Runtime, such as managing serialization of enum types, handling packet metadata, and simplifying table entry manipulations. This library is crucial for reducing complexity and increasing the efficiency of the controller script.

These tools collectively provide a robust framework for the DPI controller, enabling it to perform sophisticated network packet analyses and apply DPI rules dynamically based on real-time network conditions.

5.3 Data Plane-Centric DPI Strategy (Data plane-based Implementation)

This configuration leverages the data plane exclusively to perform Deep Packet Inspection (DPI), bypassing the control plane's involvement in the DPI process. This method tries to optimize performance by minimizing latency associated with control plane communication and as well as maximize the processing capabilities at the data plane level.

5.3.1 Operational Methodology

The DPI is conducted directly within the data plane by using P4 programming to parse and inspect packets as they pass through the network device. This approach involves identifying and parsing various packet headers—Ethernet, IP, TCP, and application-level headers like HTTP or SQL based on the destination port in the TCP header.

To accomplish the objectives with this approach we had to define network headers, set up parsing sequences, and implement logical conditions for traffic management based on header content. Figure 5.5 offers a detailed example for what the pipeline for Data Plane inspection looks like, whose description is:

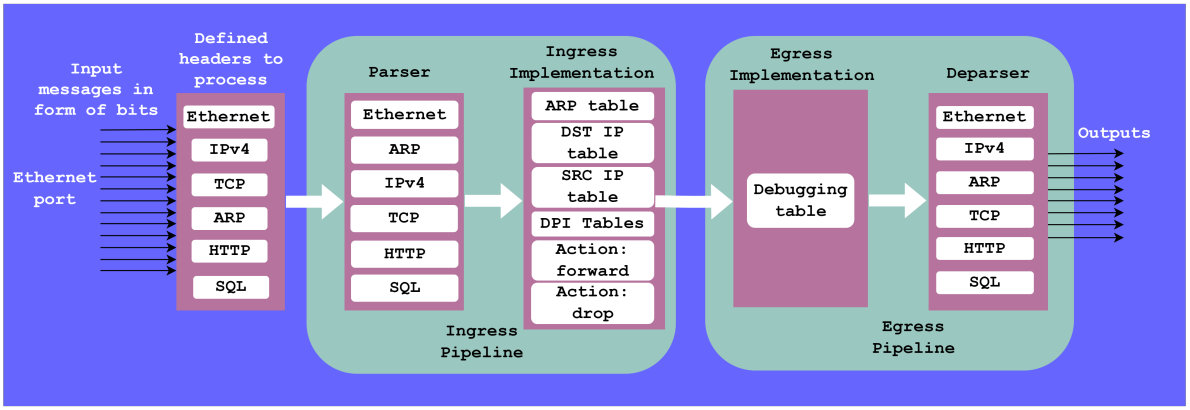

Figure 5.5: Pipeline for the processing on Data Plane only

Header Definitions We would start defining standard P4 libraries and defining network headers like Ethernet, IPv4, TCP, and ARP. These headers enable the parsing of standard network layers. Additionally, custom headers such as `http_t` and `http_payload_header_t` are defined to handle specific DPI tasks related to HTTP traffic.

Parser Implementation The parser is the critical component where packet headers are sequentially extracted and inspected. Depending on the etherType, the parser decides whether to parse IPv4, ARP, or skip to the next relevant header. For IPv4 packets, if the protocol indicates TCP (protocol number 6), it proceeds to parse TCP headers. For TCP packets destined to port 80, it additionally parses custom HTTP headers, facilitating application-level DPI for HTTP traffic.

Ingress Processing In the ingress control block, actions and tables are defined to specify how packets should be handled based on their header content. This includes forwarding rules, dropping conditions, and specialized handling for ARP and HTTP traffic.

Egress Processing Egress processing is an important stage in the data plane-centric DPI strategy where decisions made at the ingress are enforced. After packets have been scrutinized and corresponding actions determined, they arrive at the egress pipeline. Here, decisions like forwarding to the appropriate port are executed. In the context of our DPI application, egress processing is relatively light—mostly it ensures that packets are sent out without additional inspection, as significant DPI work occurs during ingress processing. For efficiency, the egress stage is designed to be simple, ensuring that DPI does not introduce unnecessary latency at this stage of packet handling.

Deparsing Once the egress processing is complete, the packet must be reconstructed from the parsed header structures before transmission. This phase, known as deparsing, is where headers are reassembled in the correct sequence of bits to form a standard network packet. It is critical to maintain the integrity of packet structure modifications made during ingress processing. In our DPI framework, deparsing carefully reassembles headers, including any custom headers that were inserted or altered during the DPI analysis, ensuring the packet is correctly formatted for network delivery.

Algorithm for DPI Using P4

The DPI functionality is realized through a series of programmed steps in the P4 device:

1. **Initialization:** Start by parsing Ethernet headers, determining the packet type, and directing the parsing process accordingly.
2. **Header Parsing:** Sequentially parse IPv4, TCP headers, and potentially HTTP or SQL information based on the packet's protocol and destination port.
3. **DPI Logic Execution:** Utilize defined tables to apply DPI logic, such as filtering HTTP methods, SQL command filtering or managing TCP ports.
4. **Action Execution:** Based on the parsed information and DPI rules, execute actions like forwarding or dropping packets.
5. **Packet Emission:** Reconstruct the packet by emitting all parsed headers in the deparser, preparing it for exit from the switch.

For a more technical review of our methodology to do Data Plane Deep Packet Inspection we can look into Algorithm 5.

5.3.2 Performance Considerations

By implementing DPI directly in the data plane, we expect to reduce the DPI system's overall latency without losing accuracy while identifying threats. This would tell us whether our second and third hypotheses are correct or not. There are however some limitations that we also expect, such as:

- **Complexity:** It requires intricate programming and understanding of network protocols at a granular level.
- **Flexibility:** It is less flexible in dynamically changing DPI logic compared to control plane-based solutions.

This approach exemplifies the advanced capabilities of P4 programming in facilitating sophisticated network functions like DPI directly within the data plane, showcasing significant potential for future network management and security infrastructures.

Algorithm 5 P4 Data Plane Processing

```
// Parser
function MyParserpacket, headers, metadata, standard_metadata
    Extract Ethernet headers
    if Ethernet type is IPv4 then
        Extract IPv4 headers
        if IPv4 protocol is TCP then
            Extract TCP headers
            if TCP destination port is 3306 then
                Extract SQL headers
            end if
        else if IPv4 protocol is ICMP then
            Extract ICMP headers
        end if
    else if Ethernet type is ARP then
        Extract ARP headers
    end if
end function
// Ingress Processing
function MyIngressheaders, metadata, standard_metadata
    if ARP header is valid then
        Apply ARP table
    else if IPv4 and ICMP headers are valid then
        Apply ICMP table
    else if TCP header is valid then
        if TCP source port is 3306 then
            Apply source TCP port 3306 table
        else if TCP destination port is 3306 then
            Apply destination TCP port 3306 table
        else
            Drop packet
        end if
    end if
end function
// Egress Processing
function MyEgressheaders, metadata, standard_metadata
    // Usually minimal processing here
end function
// Checksum Computation
function MyComputeChecksumheaders, metadata
    // Implemented as needed
end function
// Deparsing
function MyDeparserpacket, headers
    Emit all headers in the correct order
end function
```

Chapter 6

Tests and Results

This chapter presents the experimental framework and evaluates the results of two sophisticated Deep Packet Inspection (DPI) techniques designed for network security enforcement within Software-Defined Networking (SDN) environments. The tests focus on URL filtering for HTTP traffic and command filtering for SQL sessions to mitigate unauthorized activities.

6.1 HTTP URL Filtering

This mechanism aims to prevent unauthorized web access by intercepting and analyzing HTTP requests based on their URLs. The control plane employs an URL filtering application that compares GET requests against a set of blacklisted domains that we have predefined for our tests.

6.1.1 HTTP Filtering Algorithmic Implementation for Openflow-based implementation and P4runtime-based Implementation

The core functionality of this application that is going to run at the controller level is detailed in Algorithm 6, which illustrates the process of intercepting HTTP packets and determining whether they should be blocked based on the presence of blacklisted hosts. This is a straightforward application/dependency that is going to be used by the controllers in Algorithms 1 and 3 (either the POX controller or the controller we developed for P4runtime). In our real Python code, we are importing this dependency from our controller to evaluate the packet received from the data plane.

Rationale for Utilizing HTTP GET Messages in URL Filtering

HTTP GET messages are fundamental for web browsing activities, representing requests made by clients to servers. They contain essential information such as the requested URL, which is pivotal for a URL filtering application. The reasoning for employing GET messages in DPI-based URL filtering includes:

- GET requests encapsulate the URL of the requested resource, providing immediate insight into the web content being accessed.
- These messages are typically sent in plain text, making them easily interpretable by DPI mechanisms.
- The ubiquity of GET requests across all web browsing activities ensures that the filtering mechanism is comprehensive and far-reaching.

Given their significance, GET messages become the primary target for inspection in DPI to enforce security policies effectively. By examining the hostname and path specified in the GET request, a DPI system can readily determine whether to permit or deny the traffic based on predefined security rules.

HTTP GET Message Structure

An HTTP GET request comprises several fields that are useful for DPI. The following table describes the key fields found within a GET message:

Field	Description
Method	The HTTP method used, typically "GET" for retrieval of data.
URL	The Uniform Resource Locator specifying the path to the requested resource.
Host	The domain name of the server from which the resource is requested.
User-Agent	Information about the client software initiating the request.
Accept	The types of content that the client can process.

Table 6.1: Key components of an HTTP GET message relevant for DPI-based URL filtering.

The information encapsulated within these fields forms the basis for DPI processes that assess the legitimacy and safety of web traffic, thereby safeguarding the network from potential threats hidden within web requests.

Algorithm 6 HTTP URL Filtering Process

```
Require: Packet packet, Set blacklist_hosts
Ensure: Decision on whether to block the packet
// Application that is used by our controller as dependency
function PROCESS_HTTP packet, blacklist_hosts
    if packet has layer Raw then
        payload_data ← packet[Raw].load
        http_payload ← decode payload_data to UTF-8
        if "GET " in http_payload then
            host ← extract host from http_payload
            if host in blacklist_hosts then
                print "Dropping packet due to host match."
                return True                          ▷ Indicates packet should be dropped
            end if
        end if
    end if
    return False                                     ▷ Indicates packet should not be dropped
end function
```

6.1.2 Data Plane Integration

The URL filtering logic is implemented directly within the data plane using P4 programming. This approach leverages fast, in-line processing to examine and act upon HTTP requests as they traverse the network.

Algorithm 7 Ingress Processing in P4

```
// Ingress_port processing after the packets have been parsed
function INGRESS_PROCESSING: hdr, meta, stdmeta
    action DROP
        mark_to_drop(stdmeta)
    end action
    action FORWARD_TO_port
        stdmeta.egress_spec ← port
    end action
    action FORWARD_METHOD_TO_port
        stdmeta.egress_spec ← port
    end action
    action ARP_FORWARD_TO_port
        stdmeta.egress_spec ← port
    end action
    // We created multiple tables that are initiated below
    Initialize tables: filter_method, src_tcp_port_80, dst_tcp_port_80, arp_table, icmp_table
    // Then we apply the tables depending on the traffic type
    if hdr.arp.isValid() then
        arp_table.apply()
    else if hdr.ipv4.isValid() and hdr.icmp.isValid() then
        icmp_table.apply()
    else if hdr.ipv4.isValid() and hdr.tcp.isValid() then
        if hdr.tcp.srcPort = 80 then
            src_tcp_port_80.apply()
        else if hdr.tcp.dstPort = 80 then
            filter_method.apply()
            if meta.apply_dst_tcp_port_80 then
                dst_tcp_port_80.apply()
            end if
        else
            drop()
        end if
    end if
end function
```

Defining P4 Actions and Table for URL Filtering

To facilitate URL filtering, several actions and a decision table are defined within the P4 program seen in Algorithm 7 as follows:

- **drop:** This action marks packets for dropping, effectively blocking them from further transmission within the network.
- **forward:** Directs packets to a specified port, allowing permissible traffic to continue to its destination.
- **set_apply_dst_tcp_port_80:** Flags HTTP traffic destined for TCP port 80, indicating it should undergo further inspection.

The *filter_method* table utilizes these actions based on matching criteria including ingress port, HTTP method, and host field within HTTP headers. The default action, *set_apply_dst_tcp_port_80*, is configured to handle typical web traffic, while specific rules are designed to intercept and evaluate traffic against the blacklist.

Table Entries for Blacklisted Domains

The 'filter_method' table in Table 6.2 is crucial for implementing URL filtering directly in the data plane. It contains entries that match HTTP requests based on method and host header fields to determine whether a request should be blocked. Each entry is designed to match the HTTP GET method and specific host names that have been identified as malicious or unwanted. Here's a detailed look at how these entries function:

- The key for each entry combines the ingress port, HTTP method, and the host field in the HTTP header. The method and host are matched using ternary matching which allows for wildcard and exact matches.
- The actions associated with these entries include dropping the packet (effectively blocking the request) or forwarding it, depending on whether the host matches a blacklisted value.

Table Entries for Blacklisted Domains

The 'filter_method' table contains entries that are crucial for the deep packet inspection system to identify and block HTTP requests based on domain names. Each entry specifies a pattern to match against the HTTP GET request's method and host header fields, utilizing wildcard masks to allow for flexible matching. Packets matching these specifications are dropped, enhancing network security by preventing access to blacklisted domains.

Entry No.	Method Match	Host Match	Wildcard Mask	Action
1	0x47455420	0x0000000000006430647179317765307133662e636f6d0000000000000000	0x00000000000ffffffffffffffffffffffffffffffffffff000000000000000000	Drop
2	0x47455420	0x0000000000063752e636f6d0000000000000000000000000000000000000000	0x00000000000ffffffffffff000000000000000000000000000000000000000000	Drop
3	0x47455420	0x0000000000006d756966362e636f6d0000000000000000000000000000000000	0x00000000000ffffffffffffffffffff000000000000000000000000000000000000	Drop
4	0x47455420	0x000000000000773065656f2e636f6d0000000000000000000000000000000000	0x00000000000ffffffffffffffffffff000000000000000000000000000000000000	Drop
5	0x47455420	0x00000000000072623374677474322e636f6d0000000000000000000000000000	0x00000000000ffffffffffffffffffffffffff000000000000000000000000000000	Drop
6	0x47455420	0x0000000000006e65337968347a7967316f34346a2e636f6d0000000000000000	0x00000000000ffffffffffffffffffffffffffffffffffffff000000000000000000	Drop

Table 6.2: Examples of entries in the `filter_method` table used to identify and block HTTP requests based on domain blacklisting.

This table lists specific entries designed to block HTTP GET requests to certain domains. Each entry uses a method match to target the 'GET' method and a host match for the domain, with a wildcard mask to account for variations in the packet's format. The action column specifies that matching packets are to be dropped, thereby preventing unauthorized access to potentially harmful domains. This granularity allows for precise control over network traffic, critical for maintaining security in network environments. Among the domains that we blocked for our test, we can see 6 in table 6.2 out of 212 in total, converting the hexadecimal hosts in table 6.2 to utf8 we can see that the domains are: d0dqy1we0q3f.com, cu.com, muif6.com, w0eeo.com, rb3tgtt2.com, ne3yh4zyg1o44j.com.

6.2 SQL Command Filtering

This section introduces a SQL command filtering application developed to restrict database command access based on user permissions and their originating IP addresses. The application was evaluated across our three distinct DPI frameworks: Data plane-based DPI, OpenFlow-based DPI, and P4Runtime-based

DPI. Each framework utilized distinct mechanisms for enforcing SQL command filters, leveraging both control and data plane resources.

6.2.1 MySQL Packet Analysis

The Table 6.3 includes several fields that can be found in a SQL request query message, each of them taking part in the SQL command filtering application.

Field	Description
MySQL Protocol - Packet Length	Size of the MySQL-specific data within a packet.
MySQL Protocol - Packet Number	Sequence identifier of the MySQL packet within the communication session.
MySQL Protocol - Request Command	Type of MySQL command; 'Query (3)' indicates an SQL query.
MySQL Protocol - Statement	The SQL statement being executed, visible if not encrypted.
MySQL Protocol - Payload	Hexadecimal and ASCII representation of the SQL statement, which is used for SQL command filtering.

Table 6.3: Analysis of a MySQL "Request Query" Message

Each field, especially the Payload, is scrutinized by the SQL command filtering application. If the Payload matches a blacklisted command, the packet is dropped to prevent unauthorized database operations.

6.2.2 SQL Command Filtering using OpenFlow and P4Runtime

The SQL command filtering application for OpenFlow and P4Runtime environments employed the POX controller and our own controller respectively to dynamically intercept and analyze SQL traffic. The core functionality is centered on detecting and filtering SQL commands based on a predefined blacklist.

Algorithm Description

The algorithm operates within the controller framework to process incoming SQL traffic, specifically targeting packets directed to MySQL's default port (3306).

Algorithm 8 SQL Command Filtering application

Require: tcp_segment *parameter*, blacklist_command_list *parameter*
Ensure: Decision on whether to drop or forward a packet based on SQL command analysis

```
1: // Application that is used by our controller as dependency
2: function PROCESS_SQL tcp_segment, blacklist_commands
3:     sql_payload ← tcp_segment.payload.decode('utf-8', errors='ignore')
4:     Clean sql_payload to remove non-printable characters
5:     first_word ← First word of sql_payload, lowercased
6:     if first_word in blacklist_commands then
7:         print("Dropping packet due to command filter.")
8:         return True                          ▷ Indicates the packet should be dropped
9:     end if
10:    return False                         ▷ Indicates the packet should not be dropped
11: end function
```

The algorithm 8 is integrated within the controller to intercept network packets destined for SQL services. It utilizes a blacklist mechanism, reading prohibited SQL commands from a specified file. Each packet destined for the MySQL port is reviewed and if it contains SQL commands, these are checked

against the blacklist. If a match is found, the packet is dropped to prevent potentially malicious database operations. This proactive filtering helps in maintaining database integrity and preventing unauthorized data manipulations.

6.2.3 Data Plane-Based Implementation

The data plane-based DPI utilizes P4 programming to implement SQL command filtering directly within the network fabric, reducing the reliance on external control applications and enhancing performance by localizing data processing.

P4 Program Overview The P4 program parses Ethernet, ARP, IPv4, and TCP headers to inspect and process SQL traffic. It applies a series of tables that match on TCP port numbers and SQL command keywords extracted from packet payloads. Commands identified as restricted are subsequently dropped; others are permitted to proceed.

Algorithm 9 Ingress Processing for SQL Command Filtering in P4

```
// Ingress_port processing after the packets have been parsed
function IngressProcessing: hdr, meta, stdmeta
    action drop
        mark_to_drop(stdmeta)
    end action
    action forward_to_port
        stdmeta.egress_spec ← port
    end action
    action forward_method_to_port
        stdmeta.egress_spec ← port
    end action
    action arp_forward_to_port
        stdmeta.egress_spec ← port
    end action
    action set_apply_dst_tcp_port_3306
        meta.apply_dst_tcp_port_3306 ← true
    end action
    // We created multiple tables that are initiated below
    Initialize tables: filter_method, src_tcp_port_3306, dst_tcp_port_3306, arp_table, icmp_table
    // Then we apply the tables depending on the traffic type
    if hdr.arp.isValid() then
        arp_table.apply()
    else if hdr.ipv4.isValid() and hdr.icmp.isValid() then
        icmp_table.apply()
    else if hdr.tcp.isValid() then
        if hdr.tcp.srcPort = 3306 then
            src_tcp_port_3306.apply()
        else if hdr.tcp.dstPort = 3306 then
            filter_method.apply()
            if meta.apply_dst_tcp_port_3306 then
                dst_tcp_port_3306.apply()
            end if
        else
            drop()
        end if
    end if
end function
```

Table Entries for Blacklisted Commands

The `filter_method` table defined in algorithm 9 within our DPI system utilizes specific entries to identify and filter out unauthorized SQL commands. These entries are pivotal in enforcing command-level security policies, preventing the execution of potentially malicious or unauthorized SQL operations that could compromise the database integrity. Each entry is configured to recognize a particular SQL command by its hexadecimal representation in the network traffic.

Table 6.4: Example of table entries for the `filter_method` P4 table in algorithm 9 used to identify and drop SQL commands based on their encoded signatures in packet payloads.

Entry No.	Command Signature	Hexadecimal Mask	Action
1	DROP	`0x44524F500000000000000000000000000 & 0xffffffff000000000000000000000000`	drop
2	SHOW	`0x53484F570000000000000000000000000 & 0xffffffff000000000000000000000000`	drop
3	GRANT	`0x4752414E540000000000000000000000000 & 0xffffffffff000000000000000000000000`	drop
4	REVOKE	`0x5245564F4B450000000000000000000000 & 0xffffffffffff000000000000000000000000`	drop
5	CREATE	`0x435245415445000000000000000000000 & 0xffffffffffff000000000000000000000000`	drop
6	USE	`0x555345000000000000000000000000000 & 0xffffff000000000000000000000000000000`	drop

The hex masks associated with each command enable precise filtering by matching specific patterns in the packet data. This method effectively blocks the execution of blacklisted commands such as *DROP*, *SHOW*, *GRANT*, *REVOKE*, *CREATE*, *USE*, etc. which are critical to maintaining control over database operations. Such granular control helps prevent SQL injection attacks and restricts database access to authorized operations only, significantly enhancing the security framework.

6.3 Benchmarking

The benchmarking process to compare the different implementations, OpenFlow-based, P4runtime-based, and Data Plane-Centric involves comparing various performance metrics, including delay, accuracy, and resource usage. To quantify the relative performance differences between the implementations, we use the Percentage Change method, which is a standard approach in performance evaluation studies.

6.3.1 Percentage Change Method

The Percentage Change method is used to express the relative difference between the performance metrics of different implementations. This method provides a clear and intuitive way to understand how much one implementation outperforms or underperforms another. The percentage change is calculated using the following formula:

$$PercentageChange = \left(\frac{ImplementationValue - BaselineValue}{BaselineValue} \right) \times 100 \tag{6.1}$$

where:

- **Implementation Value** is the metric value for the DPI implementation being evaluated.
- **Baseline Value** is the metric value for the baseline DPI implementation (in our case, the OpenFlow-based implementation).

Depending on the metric to be evaluated we should interpret the results in the following way:

- **For the accuracy metric** A positive percentage change indicates that the Implementation Value is higher than the Baseline Value, signifying a performance improvement, while a negative percentage change indicates a performance degradation.

- **Delay and resource consumption metrics** A positive percentage change indicates that the Implementation Value is higher than the Baseline Value, signifying a performance degradation, while a negative percentage change indicates a performance improvement.

6.4 Results

This section delineates the outcomes derived from the experimental validation of the proposed Deep Packet Inspection (DPI) strategies as described in Chapter 5. Each DPI implementation was subjected to rigorous testing to evaluate its effectiveness and efficiency in filtering HTTP URLs.

6.4.1 HTTP URL Filtering Application

The evaluation protocol involved querying 500 distinct domains hosted on a single HTTP server, with 42% of these domains flagged as malicious and targeted by our URL filtering applications. The objective was to assess the performance of each DPI strategy in terms of latency and accuracy metrics such as precision, recall, F1 score, True Positive Rate (TPR), and False Positive Rate (FPR). Common tests would test just 20% of the dataset, but in our case we decided to double that percentage to have more conclusive results since the objective of the application is to block blacklisted domains.

Accuracy Test Results

The accuracy metrics table (Table 6.5) details the effectiveness of each DPI implementation in filtering HTTP traffic, that it was obtained by using the "scikit-learn" library. . All DPI strategies maintain high accuracy and precision, illustrating their capability to discern between legitimate and malicious web traffic effectively. Slight variations in recall and false positive rates suggest differences in how stringent or lenient each system is in classifying domains as malicious, which could influence overall security and user experience.

Implementation	Accuracy	Precision	Recall	F1 Score	TPR	FPR
OpenFlow-based DPI	0.99	1.00	0.99	0.99	1.00	0.01
P4runtime-based DPI	0.99	1.00	0.98	0.99	1.00	0.02
Data Plane-Centric DPI	0.99	1.00	0.98	0.99	1.00	0.02

Table 6.5: Accuracy metrics for HTTP URL filtering across different DPI implementations.

Delay Test Results

Delay metrics are crucial for network operations, particularly when real-time or near-real-time processing is required. The delay metrics table (Table 6.11) compares the minimum, average, and maximum response times experienced by each DPI implementation when processing HTTP requests. The P4 Data Plane-Centric DPI shows superior performance with significantly reduced latency, enhancing its suitability for environments where speedy data processing and decision-making are fundamental.

Implementation	Min Delay (ms)	Avg Delay (ms)	Max Delay (ms)
OpenFlow-based DPI	30.06	97.07	196.21
P4runtime-based DPI	97.01	236.12	1224.57
Data Plane-Centric DPI	6.25	33.23	156.07

Table 6.6: Response time metrics for HTTP URL filtering application across different DPI implementations.

Performance Metrics During The Test Execution Across DPI Implementations Results

The data presented in Table 6.13 showcases the performance variances across different DPI implementations. The P4 Data Plane-Centric DPI is notably the most efficient, with the lowest execution time of 286865 ms, reflecting superior responsiveness and less computational delay. This implementation also has the lowest peak CPU usage at 34.62%, and a corresponding CPU Usage Increment Factor of 33.61, which indicates a moderate increase in CPU load during operation. In comparison, the P4runtime-based DPI demands the highest computational resources, evident from its execution time of 491242 ms, peak CPU usage at 57%, and the highest memory usage at 99072 KB. The OpenFlow-based DPI, while slightly more efficient than P4runtime in terms of execution time and CPU usage, still falls short of the performance exhibited by the P4 Data Plane-Centric approach. This analysis highlights the P4 Data Plane-Centric DPI's effectiveness in managing resource utilization while maintaining lower operational latencies, which is crucial for high-performance environments requiring real-time data processing.

Table 6.7: Performance Metrics for DPI Implementations

Implementation	Execution Time (ms)	Peak CPU Usage (%)	CPU Usage Increment Factor	CPU Time (s)	Max Memory Usage (KB)
P4runtime-based DPI	491242	57%	57.00	2.44s	99072
OpenFlow-based DPI	464454	43.03%	43.03	2.42s	98876
Data Plane-Centric DPI	286865	34.62%	33.61	1.79s	98860

Table 6.13 represents:

- **Execution Time:** Measures the duration from the start to the end of the DPI script, with shorter times reflecting better responsiveness.
- **Peak CPU Usage:** Indicates the highest CPU load during execution, highlighting the computational demand of the DPI process.
- **CPU Usage Increment Factor:** Shows the ratio of peak CPU usage to the idle state, important for understanding the increase in load due to DPI.
- **CPU Time:** Represents the actual CPU time consumed by the process, focusing on processor usage excluding any external delays.
- **Maximum Memory Usage:** Details the highest memory requirement during execution, critical for managing resources and preventing overallocation.

Comparative Analysis

A comparative analysis in figure 6.1 reveals that while all three DPI implementations maintain high accuracy and precision , the P4 data plane-centric approach offers superior performance in terms of latency reduction as we can see in figure 6.2. This finding underscores the advantages of localized data processing in reducing network strain and improving overall system responsiveness. The extended delays observed in the BMv2 and P4runtime setup could be attributed to the overhead introduced by more complex processing and control logic, which may affect scalability in larger network environments.

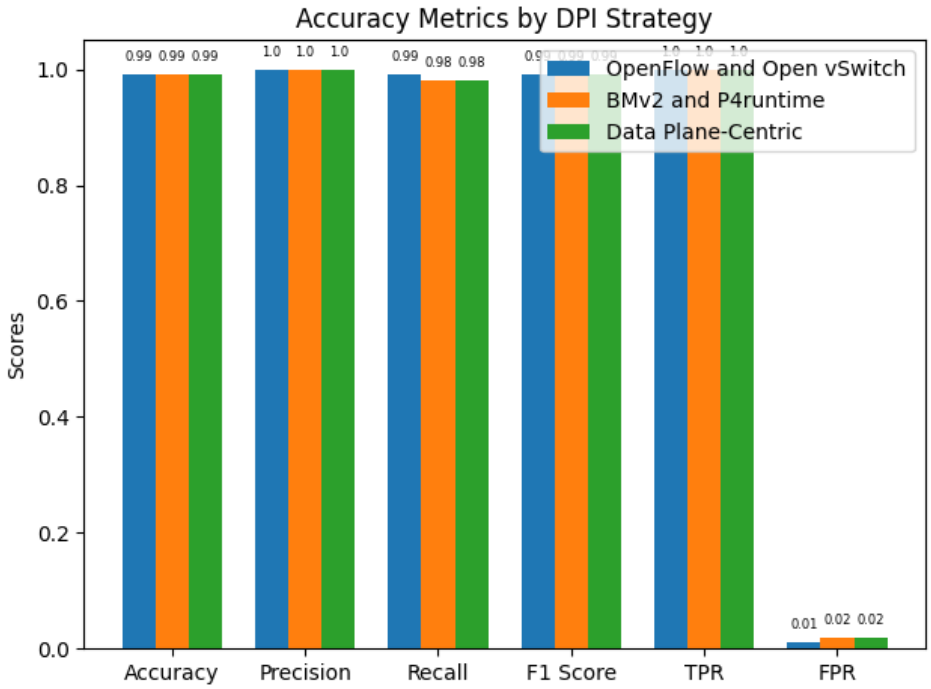

Figure 6.1: Accuracy comparison for all the implementations

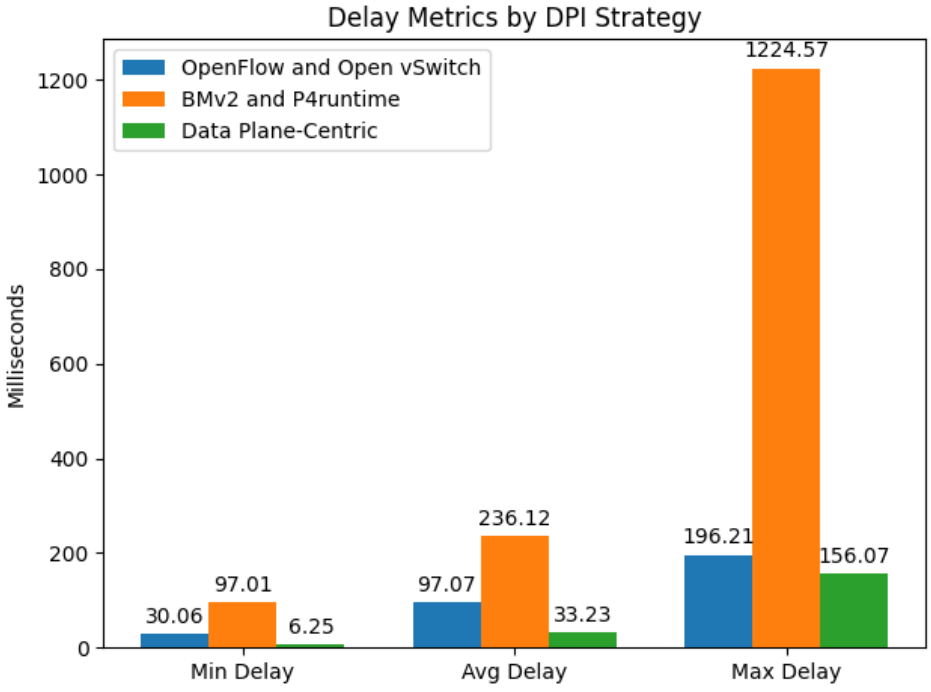

Figure 6.2: delay comparison for all the implementations

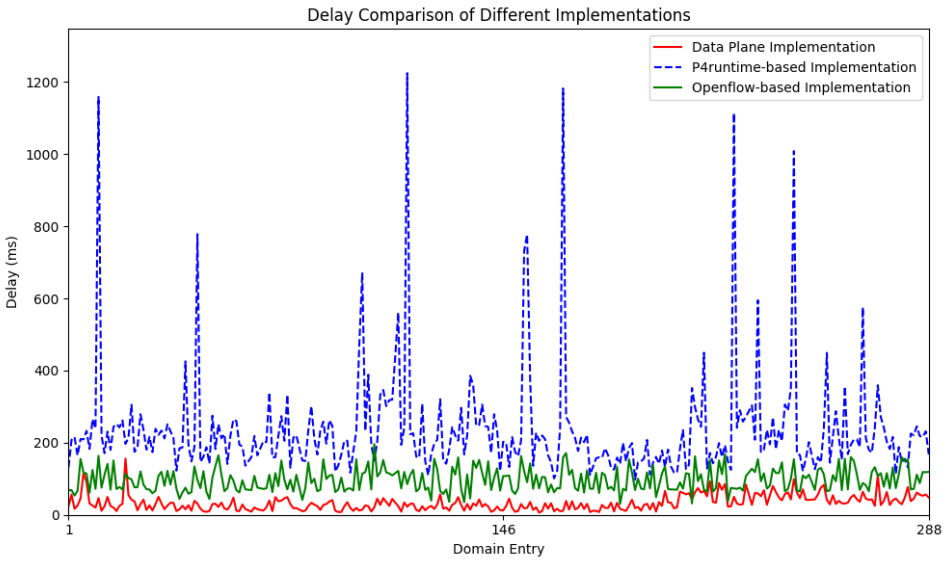

Figure 6.3: Delay comparison for our URL filtering application for the different implementations

Besides, in 6.14 we can see the results of the Percentage Change analysis for the accuracy, average delay, execution time, peak CPU usage, CPU time, and maximum memory usage metrics across different DPI implementations. As described previously, the OpenFlow-based DPI implementation is used as the baseline for comparison.

Table 6.8: Percentage Change in Performance Metrics Compared to OpenFlow-based DPI

Implementation	Accuracy (%)	Avg Delay (%)	Execution Time (%)	Peak CPU Usage (%)	CPU Time (%)	Max Memory Usage (%)
P4runtime-based DPI	0.00%	+143.34%	+5.76%	+32.43%	+0.83%	+0.20%
Data Plane-Centric DPI	0.00%	-65.77%	-38.25%	-19.55%	-26.03%	-0.02%

The accuracy for both P4runtime-based DPI and P4 Data Plane-Centric DPI is the same as the OpenFlow-based DPI, resulting in a 0.00% percentage change. This indicates that all implementations maintain the same level of accuracy.

Average Delay: The average delay for P4runtime-based DPI is 143.34% higher than the OpenFlow-based DPI, indicating significantly higher latency. In contrast, the Data Plane-Centric DPI shows a 65.77% reduction in average delay, highlighting its superior performance in terms of latency.

Execution Time: The P4runtime-based DPI implementation has an execution time 5.76% higher than the OpenFlow-based DPI, while the Data Plane-Centric DPI has a 38.25% lower execution time, demonstrating its efficiency.

Peak CPU Usage: The P4runtime-based DPI shows a 32.43% increase in peak CPU usage compared to the OpenFlow-based DPI. In contrast, the Data Plane-Centric DPI has a 19.55% lower peak CPU usage, indicating better resource utilization.

CPU Time: The CPU time for the P4runtime-based DPI is 0.83% higher than the OpenFlow-based DPI, whereas the Data Plane-Centric DPI has a 26.03% lower CPU time, further supporting its efficiency.

Max Memory Usage: The maximum memory usage for the P4runtime-based DPI is 0.20% higher, and for the Data Plane-Centric DPI is 0.02% lower than the OpenFlow-based DPI, suggesting that memory usage differences are minimal.

All in All, the Percentage Change analysis highlights that the P4 Data Plane-Centric DPI implementation consistently outperforms the OpenFlow-based DPI implementation in terms of lower average delay, execution time, peak CPU usage, and CPU time. The P4runtime-based DPI implementation,

while maintaining the same accuracy, incurs significantly higher delays and resource usage suggesting that P4 does better than OpenFlow for Data-plane implementations, but when it is linked to a controller it can produce worst results.

6.4.2 SQL Command Filtering Application

This section of the thesis focuses on evaluating the effectiveness of SQL command filtering across three distinct DPI implementations: OpenFlow-based DPI, P4runtime-based DPI, and P4 Data Plane-Centric DPI. This application is critical for preventing SQL injections and privilege escalation by restricting SQL commands based on user permissions.

Testing Methodology

We developed some Python scripts to execute a series of SQL commands against a test database, measuring both the induced delay and the accuracy of command filtering. The script executes each command and records the response time, allowing us to quantify the impact of the filtering mechanism on database accessibility.

Delay Measurement

The Python script utilized subprocesses to send SQL commands to a MySQL server and measured the time taken since each command is executed until we it receives a response from the server. This method simulates an operational environment where SQL commands are issued sequentially, capturing the realistic impacts of command filtering on performance. The commands we fed to algorithm 10 to measure delay can be observed in Tabke 6.9

Table 6.9: MySQL Commands Used in Delay Measurement

Command Number	**Command**
1	SHOW DATABASES
2	SHOW TABLES
3	SHOW COLUMNS FROM "table_name"
4	SHOW INDEX FROM "table_name"
5	SHOW TABLE STATUS LIKE "table_name"
6	SHOW PROCESSLIST
7	SHOW GRANTS FOR 'username'@'localhost'
8	SHOW CREATE TABLE "table_name"
9	SHOW VARIABLES
10	SHOW STATUS
11	SHOW ERRORS
12	SHOW WARNINGS

Accuracy Measurement

To assess the accuracy of the SQL command filtering, we use "scikit-learn" library. Also, the accuracy was calculated based on the binary classification of command execution outcomes as 'reachable' or 'unreachable,' reflecting whether a command was blocked by the application or not. To simplify the test we blocked all the SQL commands in our application(considering that the goal of the application is

Algorithm 10 MySQL Command Execution and Response Time Measurement

Require: List of *commands*, Configuration $config$
Ensure: CSV file with command response times

// Defined function that is going to require the SQL commands to test
function EXECUTE_MYSQL_COMMAND*command*, *timeout*
    $start\_time \leftarrow$ current time
    $process \leftarrow$ start subprocess with *command* and *timeout*
    wait for *process* to complete
    $end\_time \leftarrow$ current time
    $delay\_ms \leftarrow$ calculate elapsed time in ms
    return ($delay\_ms$, 'reachable') subprocess.Called_Process_Error
    return ($timeout \times 1000$, 'unreachable')
end function
//Feed the commands to test into the previous function to calculate the delay
Initialize *delays* list
Open 'command_response_times.csv' as $csvfile$ for writing
$writer \leftarrow$ CSV writer for $csvfile$
Write headers to $csvfile$
Then i in range 1 to 9 **do**
    for all *cmd* in *commands* **do**
        $(delay, status) \leftarrow$ EXECUTE_MYSQL_COMMAND(*cmd*, $config$)
        Write $(cmd, delay, status)$ to $csvfile$
        if *status* is 'reachable' **then**
            Append *delay* to *delays*
        end if
    end Then
end Then
// We then calculate that was the commands with the least delay, the average delay and the slowest command
if *delays* is not empty **then**
    Calculate min_delay, avg_delay, max_delay from *delays*
    Write summary statistics to $csvfile$
else
    Print no reachable commands found
end if

to block blacklisted commands) and measure how many of them were effectively blocked or passed the filter. The commands that we tested to evaluate the accuracy of our application is listed in table 6.10:

We present the results of the delay and accuracy measurements for each DPI implementation in structured tables, providing a clear comparison of performance and efficacy across the different setups.

Delay Test Results The delay introduced by SQL command filtering is tabulated below in table 6.11 for each DPI implementation. These metrics include the minimum, average, and maximum response times recorded during the tests. We can clearly see that the P4 Data Plane-Centric Implementation is once again the one that introduces the least delay to the DPI solution.

Implementation	Min Delay (ms)	Avg Delay (ms)	Max Delay (ms)
OpenFlow-based DPI	247.69	391.45	1139.55
P4runtime-based DPI	241.32	779.43	2054.51
Data Plane-Centric DPI	42.01	346.40	1853.58

Table 6.11: Response time metrics for our SQL command filtering application across different DPI implementations.

Accuracy Test Results The accuracy of the SQL command filtering process is also essential, reflecting the system's ability to correctly identify and block unauthorized commands. The table 6.12 summarizes these metrics, in here we can see that the P4 Data Plane-Centric Implementation is also the best one out of all the considered solutions.

Implementation	Accuracy
OpenFlow-based DPI	0.6061
P4runtime-based DPI	0.5152
Data Plane-Centric DPI	0.7576

Table 6.12: Accuracy metrics for SQL command filtering across different DPI implementations.

The results clearly indicate the variations in delay and accuracy across implementations, highlighting the trade-offs between different DPI approaches. These insights are critical for network administrators to choose the appropriate DPI strategy based on their specific security needs and operational constraints.

Performance Metrics During The Test Execution Across DPI Implementations Results

Table 6.13 reveals that while the OpenFlow-based DPI is the most responsive due to its shortest execution time and lowest CPU time, the P4 Data Plane-Centric DPI offers the best performance in terms of CPU efficiency and balanced memory usage. This makes it a strong candidate for resource-constrained environments where CPU and memory efficiency are critical. The P4runtime-based DPI, despite its higher execution time, excels in maintaining a low CPU usage increment and minimal memory usage, making it a viable option where maintaining a minimal increase in computational load is essential. Therefore, for environments prioritizing CPU and memory efficiency, the P4 Data Plane-Centric DPI implementation is recommended as the better choice.

Table 6.13: Performance Metrics for DPI Implementations

Implementation	Execution Time (ms)	Peak CPU Usage (%)	CPU Usage Increment Factor	CPU Time (s)	Max Memory Usage (KB)
P4runtime-based DPI	63535	65.44%	40.90	2.38s	92368
OpenFlow-based DPI	46544	65.2%	65.20	2.31 s	98272
Data Plane-Centric DPI	54430	63.3%	63.30	2.43s	92488

Table 6.13 represents:

- **Execution Time:** Measures the duration from the start to the end of the DPI script, with shorter times reflecting better responsiveness.
- **Peak CPU Usage:** Indicates the highest CPU load during execution, highlighting the computational demand of the DPI process.
- **CPU Usage Increment Factor:** Shows the ratio of peak CPU usage to the idle state, important for understanding the increase in load due to DPI.
- **CPU Time:** Represents the actual CPU time consumed by the process, focusing on processor usage excluding any external delays.
- **Maximum Memory Usage:** Details the highest memory requirement during execution, critical for managing resources and preventing overallocation.

Comparative Analysis

From the results in Figure 6.4 and 6.5 we can see that the P4 Data plane implementation is the one that introduces less delay on average and it has the lowest delay out of all the implementations for an individual command. Besides, in terms of accuracy for SQL commands we also can see that the P4 Data plane implementation produced the best results.

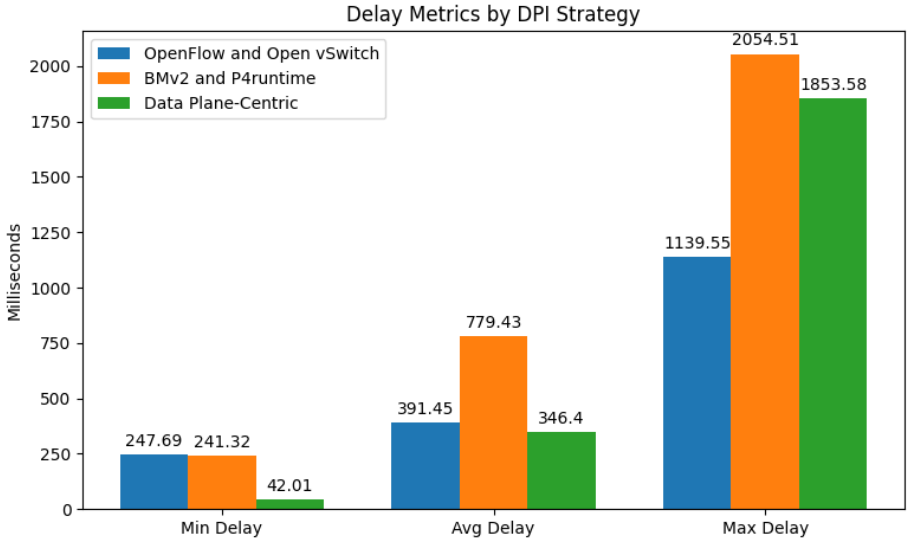

Figure 6.4: Minimum, Average, and Maximum delay comparison for the different implementations

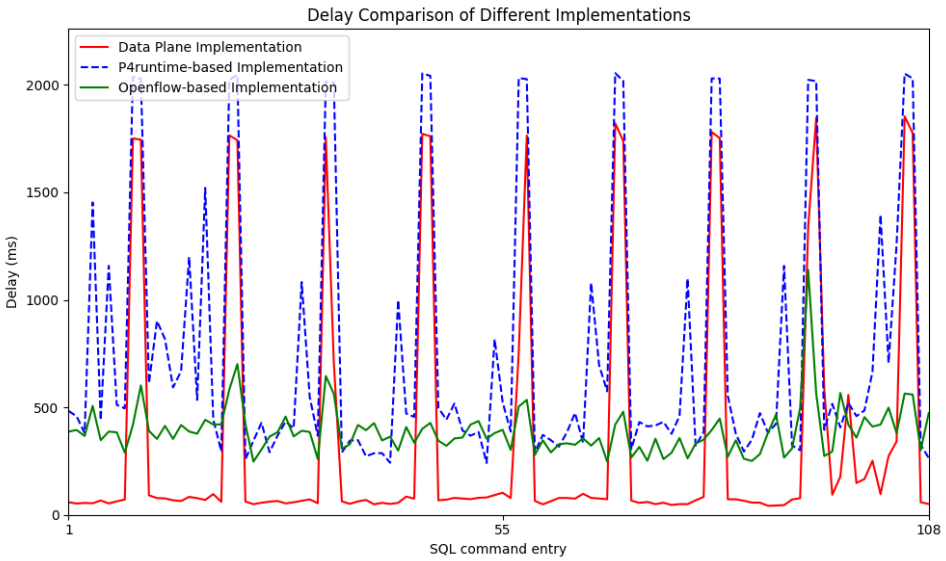

Figure 6.5: Delay comparison for our SQL command filtering application for the different implementations

Additionally, in table 6.14 we can see the Percentage Change analysis for the accuracy, average delay, execution time, peak CPU usage, CPU time, and maximum memory usage metrics across different DPI implementations. Once again, the OpenFlow-based DPI implementation is used as the baseline for comparison.

Table 6.14: Percentage Change in Performance Metrics Compared to OpenFlow-based DPI

Implementation	Accuracy (%)	Avg Delay (%)	Execution Time (%)	Peak CPU Usage (%)	CPU Time (%)	Max Memory Usage (%)
P4runtime-based DPI	-15.00%	+99.13%	+36.51%	+0.37%	+3.03%	-6.02%
Data Plane-Centric DPI	+25.00%	-11.51%	+16.95%	-2.91%	+5.19%	-5.89%

From table 6.14 we observe that the accuracy for the P4runtime-based DPI is 15.00% lower than the OpenFlow-based DPI, whereas the P4 Data Plane-Centric DPI has a 25.00% higher accuracy. This indicates that the P4 Data Plane-Centric DPI implementation performs significantly better in terms of accuracy.

Average Delay: The average delay for P4runtime-based DPI is 99.13% higher than the OpenFlow-based DPI, indicating significantly higher latency. In contrast, the Data Plane-Centric DPI shows an 11.51% reduction in average delay, highlighting its superior performance in terms of latency.

Execution Time: The P4runtime-based DPI implementation has an execution time 36.51% higher than the OpenFlow-based DPI, while the Data Plane-Centric DPI has a 16.95% higher execution time, which in this case means the Openflow was faster than both P4-based implementations.

Peak CPU Usage: The P4runtime-based DPI shows a 0.37% increase in peak CPU usage compared to the OpenFlow-based DPI. In contrast, the Data Plane-Centric DPI has a 2.91% lower peak CPU usage, indicating better resource utilization.

CPU Time: The CPU time for the P4runtime-based DPI is 3.03% higher than the OpenFlow-based DPI, whereas the Data Plane-Centric DPI has a 5.19% higher CPU time, which again gives the edge to Openflow for the minimum.

Max Memory Usage: The maximum memory usage for the P4runtime-based DPI is 6.02% lower, and for the Data Plane-Centric DPI is 5.89% lower than the OpenFlow-based DPI, suggesting that memory usage differences are minimal, but still P4-based implementations were better than Openflow's.

In conclusion, for this application, the Percentage Change analysis highlights that the P4 Data Plane-Centric DPI implementation consistently outperforms the OpenFlow-based DPI implementation in terms of higher accuracy, lower average delay, and better resource utilization. The P4runtime-based DPI implementation, while maintaining some performance improvements, incurs significantly higher delays and resource usage. This analysis provides a clear and quantitative comparison of the DPI implementations, validating the superior performance of the P4 Data Plane-Centric DPI approach.

Data Control Language (DCL) commands
GRANT SELECT ON countries TO 'other_user'@'localhost'
REVOKE SELECT ON countries FROM 'other_user'@'localhost'
Data Definition Language (DDL) commands
CREATE DATABASE IF NOT EXISTS temp_db
USE temp_db
CREATE TABLE IF NOT EXISTS temp_table (id INT AUTO_INCREMENT PRIMARY KEY, data VARCHAR(100))
CREATE INDEX idx_data ON temp_table(data)
CREATE VIEW view_data AS SELECT data FROM temp_table
CREATE PROCEDURE SelectAll() BEGIN SELECT * FROM temp_table; END
CREATE FUNCTION GetDataCount() RETURNS INT BEGIN DECLARE cnt INT; SELECT COUNT(*) INTO cnt FROM temp_table; RETURN cnt; END
CREATE TRIGGER BeforeInsert BEFORE INSERT ON temp_table FOR EACH ROW SET NEW.data = CONCAT('Prefix_', NEW.data)
ALTER TABLE temp_table ADD COLUMN new_column VARCHAR(100)
ALTER TABLE temp_table DROP COLUMN new_column
DROP INDEX idx_data ON temp_table
DROP TRIGGER BeforeInsert
DROP FUNCTION GetDataCount
DROP PROCEDURE SelectAll
DROP VIEW view_data
DROP TABLE temp_table
DROP DATABASE temp_db
Data Manipulation Language (DML) commands
INSERT INTO countries (name, population) VALUES ('Testland', 123456)
UPDATE countries SET population = 654321 WHERE name = 'Testland'
DELETE FROM countries WHERE name = 'Testland'
INSERT INTO countries (name, population) VALUES ('Testland', 123456) ON DUPLICATE KEY UPDATE population = VALUES(population)
SELECT * FROM countries
Show Commands
SHOW DATABASES
SHOW TABLES
SHOW COLUMNS FROM countries
SHOW INDEX FROM countries
SHOW TABLE STATUS LIKE 'countries'
SHOW PROCESSLIST
SHOW GRANTS FOR 'username'@'localhost'
SHOW CREATE TABLE countries
SHOW VARIABLES
SHOW STATUS
SHOW ERRORS
SHOW WARNINGS
Transaction Control (TC) commands
START TRANSACTION
SAVEPOINT svpt1
INSERT INTO countries (name, population) VALUES ('TempCountry', 100)
UPDATE countries SET population = 200 WHERE name = 'TempCountry'
ROLLBACK TO svpt1
COMMIT
Utility Commands
USE sqli
SET @test_variable = 'test_value'
SELECT @test_variable
EXPLAIN SELECT * FROM countries
LOCK TABLES countries READ
UNLOCK TABLES

Table 6.10: SQL Commands Categorized by Their Properties

Chapter 7

Discussion

7.1 Interpretation of Results

This chapter delves into the implications of the results obtained from the experimental evaluations discussed in the previous chapter.

7.1.1 Efficiency of DPI Implementations

In this thesis, we are comparing Openflow with P4 implementations such as P4 based only on Data Plane (Data Plane-centric approach) and P4 interconnected with a controller (P4Runtime-based approach).

The comparative analysis of DPI with the OpenFlow vs. P4 approaches revealed significant differences in their performance and accuracy. The P4 Data Plane-centric DPI not only demonstrated the least delay but also the highest accuracy in filtering HTTP and SQL commands, besides, it also consumed the least computing resources. These results underscore the potential of P4 to leverage Data Plane processing capabilities to enhance real-time traffic analysis without burdening the control plane. On the other hand, using P4 with a controller introduces more delay in the network despite having very good accuracy.

For HTTP tests we can see that we achieved almost ideal Accuracy, Precision, Recall, F1 Score, TPR, and FPR for all implementations, which mainly indicates that our Web-filtering application algorithm is very well-designed for this purpose. On the delay measurement side on the other hand we can see the real difference for the different implementations, we can see that on average the P4 Data plane implementation is 65.77% faster than the Openflow-based implementation. However, P4 working with a controller(P4runtime implementation) has 143.34% higher latency, which highlights the benefits and drawbacks of using P4 with only Data Plane and with Control Plane vs the Openflow implementation. On the resource consumption part, the P4 Data Plane-centric DPI implementation was superior to Openflow in every way, nonetheless, the P4Runtime-based approach consumed more resources than Openflow showing again that P4 on data plane produces good results, but that is not the case for P4 control-based approaches.

For SQL tests we showed that even though the P4 Data plane implementation is the highest, it is not close to being top-notch. This is because in this thesis the SQL command filtering algorithm was not produced to be a top-notch algorithm, but, to compare that algorithm on different architecture paradigms, independently of the previous statement we still can see that the P4 Data plane implementation achieved a 25.00% better result for accuracy than the OpenFlow implementation. However, the P4 implementa-

tion with a controller produced 15.00% less accurate results than the OpenFlow implementation. On the delay measurement side, once again we can see more clearly the difference between the different implementations, we see that on average the P4 Data plane implementation was 11.51% better than the Openflow-based implementation, but P4 linked to the controller was 99.13% slower. In terms of resource consumption, the P4 implementations even though it consumed fewer resources in some categories, was not notoriously better than Openflow.

7.1.2 Trade-offs and Practical Considerations

While the P4 Data Plane-centric approach offers substantial benefits in terms of latency reduction and processing efficiency without decreasing accuracy, it requires sophisticated programming capabilities and deep understanding of network packet structures. Conversely, the OpenFlow-based DPI, despite its relatively simpler implementation and compatibility with existing SDN environments, suffers from higher latency and resource consumption, highlighting the trade-offs between ease of deployment and performance efficacy, the same happens for P4runtime implementations that rely on a controller, which makes evident that using P4 for DPI on Data Plane can add many benefits, but using it with a controller can produce worst results.

7.2 Contextual Comparison with Existing Work

Previous studies have often emphasized the scalability and flexibility of OpenFlow-based DPI solutions; however, findings from this thesis suggest that when latency and accuracy are critical, advanced data plane programming with P4 might offer superior alternatives. This aligns with recent shifts in network management paradigms towards more distributed and intelligent edge processing architectures.

7.3 Limitations of the Study

This research was conducted within a controlled environment which may not perfectly simulate real-world network conditions. The scalability of the tested DPI approaches under varied and unpredictable traffic patterns remains to be verified. Furthermore, the complexity of P4 programming and the need for specialized hardware could limit the widespread adoption of Data Plane-centric DPI strategies.

Besides, the results in the previous section and chapters is for implementations in which the Data plane and the Control plane are in separate environments(virtual machines). We also tested our algorithms when the Data plane and Control plane were hosted in the same Operating system, and the results were very similar, the OVF image provided in the appendices covers this part, where the OVF image hosts both control and data plane. These last findings support the universality of our conclusions.
This research is also limited to unencrypted data, to test encrypted messages, it is necessary to unencrypt the messages, which is out of scope of this research thesis.

7.4 Future Research Directions

Future work could explore the integration of machine learning algorithms into DPI tasks at the Data Plane level to dynamically adapt to changing network behaviors and threat landscapes.

In this research project we provided two Deep Packet Inspection applications to block unsafe domains and SQL commands previously defined, however, we have not addressed the security of the implementation components, for example, in this project, we did not harden the switches, components or applications.

Hence, if the provided framework by this research is implemented in production, this concern should be addressed.

Also, this research project has not conducted Throughput tests for the different implementations, reason for which in case of using this framework in production, Throughput studies should be carried on to make sure the different DPI approaches meet the requirements of the intended project.

Additionally, another approach that we wanted to address in this thesis, and however we could not do it for lack of time is the potential use of hybrid DPI approaches that intelligently distribute processing tasks between the Data and Control planes. This hybrid approach might yield optimal configurations that balance performance, accuracy, and resource utilization.

Lastly, another approach that seems very exciting as well is the possibility of having the control plane in the same appliance that is in charge of packet forwarding(Control plane embedded in the same appliance as the data plane). Even though this might seem like going back to the standard approach followed until before SDN, it could also be very beneficial for Deep Packet Inspection and the development of defensive appliances tailored to our necessities. We can see in figure 7.1 what an embedded controller would look like in a P4 environment.

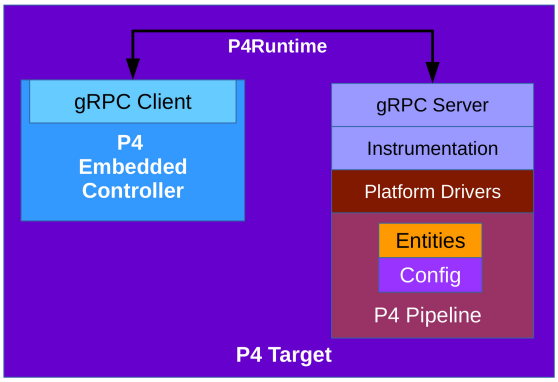

Figure 7.1: Embedded controller approach [47]

Chapter 8

Conclusions

This thesis critically examined the capabilities of Deep Packet Inspection (DPI) within Software-Defined Networking (SDN) frameworks, comparing the efficacy of the P4 programming language against the conventional OpenFlow protocol. The primary goal was to validate the hypothesis that P4, due to its flexibility and programmability, can significantly enhance DPI functionalities, especially at the application layer, compared to OpenFlow.

Through rigorous benchmarking and emulations using Open vSwitch and BMv2 switches, various DPI implementations were evaluated on metrics such as accuracy, delay, execution time, peak CPU usage, CPU time, memory usage, etc. The results provided clear insights into the performance differences between OpenFlow-based and P4-based DPI implementations.

The findings validated our hypotheses in several key areas:

1. **Flexibility and Application-layer Analysis**: The P4 Data Plane-centric DPI implementation demonstrated either the same (for the URL filtering application) or superior accuracy, achieving a 25.00% higher accuracy in SQL command filtering compared to the OpenFlow-based DPI. This confirms that P4's flexibility allows to do DPI on Data Plane without quality drops.

2. **Performance and Efficiency**: The P4-based DPI showed significant improvements in latency, with the P4 Data Plane-centric implementation being 65.77% faster for HTTP filtering and 11.51% faster for SQL filtering compared to OpenFlow. This supports the hypothesis that P4 can enhance DPI performance by enabling real-time traffic analysis with minimal delay.

3. **Resource Usage**: The P4 Data Plane-centric DPI implementation consumed fewer computing resources, particularly in terms of peak CPU usage and CPU time than the Openflow-based implementation, corroborating in this way our third hypothesis.

Having said this, we must also indicate that the P4Runtime-based DPI, which involves a controller, exhibited a worse performance in general than the Openflow-based DPI. This indicates that while P4 offers substantial benefits in a Data Plane-centric approach, its advantages diminish when linked to a controller.

Also, the Percentage Change analysis provided a quantitative comparison, highlighting that the P4 Data Plane-centric DPI consistently outperformed the OpenFlow-based DPI in terms of lower average delay, reduced resource consumption, and higher accuracy. Conversely, the P4Runtime-based DPI, while maintaining good accuracy, incurred higher delays and resource usage.

These findings suggest that advanced data plane programming with P4 offers a superior alternative to traditional OpenFlow-based approaches for DPI tasks. The ability to perform in-depth packet inspection at the data plane level can enhance real-time traffic analysis and network security without overburdening

the control plane.

To conclude, this thesis validates the potential of P4 to enhance DPI within SDN frameworks, particularly through data plane capabilities. The insights gained from this research pave the way for future innovations in network security and management, contributing to the ongoing evolution of SDN technologies.

Bibliography

[1] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," in *Proceedings of the IEEE*, vol. 103, 2015, pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[2] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," vol. 44, Apr. 2014, pp. 44–51. DOI: 10.1145/2602204.2602211.

[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 13–24, 2012. DOI: 10.1145/2342356.2342359.

[4] Enea, *The future of deep packet inspection: Key findings from the enea dpi survey*, 2020. [Online]. Available: https://www.enea.com/insights/the-future-of-deep-packet-inspection-key-findings-from-the-enea-dpi-survey.

[5] S. Gupta, D. Gosain, M. Kwon, and H. B. Acharya, "Deep4r: Deep packet inspection in p4 using packet recirculation," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10. DOI: 10.1109/INFOCOM53939.2023.10228996.

[6] R. Carvalho, L. Costa, J. Bordim, and E. Alchieri, "New programmable data plane architecture based on p4 openflow agent," pp. 1355–1367, Mar. 2020. DOI: 10.1007/978-3-030-44041-1_115.

[7] F. Hauser *et al.*, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103 561, Dec. 2022. DOI: 10.1016/j.jnca.2022.103561.

[8] W. Cordeiro, J. Marques, and L. Gaspary, "Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management," *Journal of Network and Systems Management*, vol. 25, Oct. 2017. DOI: 10.1007/s10922-017-9423-2.

[9] S. Gupta, D. Gosain, G. Grigoryan, M. Kwon, and H. B. Acharya, "Demo: Simple deep packet inspection with p4," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, 2021, pp. 1–2. DOI: 10.1109/ICNP52444.2021.9651973.

[10] A. Alsabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection," Mar. 2022. DOI: 10.14722/madweb.2022.23012.

[11] A. Ahad, R. A. Bakar, M. Arslan, and M. H. Ali, "Dpidns:a deep packet inspection based ips for security of p4 network data plane," in *2023 International Conference on Smart Computing and Application (ICSCA)*, 2023, pp. 1–8. DOI: 10.1109/ICSCA57840.2023.10087377.

[12] T. Systems, "Your all-in-one guide to p4 (programming protocol-independent packet processors)," *Trenton Systems*, 2023. [Online]. Available: https://www.trentonsystems.com/en-us/resource-hub/blog/your-all-in-one-guide-to-p4.

[13] K. Kumazoe and M. Tsuru, "P4-based implementation and evaluation of adaptive early packet discarding scheme," pp. 460–469, Jan. 2021. DOI: 10.1007/978-3-030-57796-4_44.

[14] Wikipedia, "Software-defined networking - wikipedia," *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Software-defined_networking.

[15] Cloudflare, "What is software-defined networking (sdn)? — cloudflare," *Cloudflare*, 2024. [Online]. Available: https://www.cloudflare.com/learning/network-layer/what-is-sdn/.

[16] BMC, "What is software defined networking? sdn explained," *BMC*, 2024. [Online]. Available: https://www.bmc.com/blogs/software-defined-networking/.

[17] G2, "A complete guide to software-defined networking - g2," *G2*, 2024. [Online]. Available: https://www.g2.com/articles/software-defined-networking.

[18] IBM, "What is software-defined networking (sdn)? - ibm," *IBM*, 2024. [Online]. Available: https://www.ibm.com/topics/sdn.

[19] GeeksforGeeks, "Architecture of software defined networks (sdn) - geeksforgeeks," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/architecture-of-software-defined-networks-sdn/.

[20] JavaTpoint, "Software defined networking (sdn): Benefits and challenges ... - ieom," *JavaTpoint*, 2024. [Online]. Available: https://www.javatpoint.com/software-defined-networking-sdn-benefits-and-challenges-of-network-virtualization.

[21] SDxCentral, "Understanding the sdn architecture and sdn control plane - sdxcentral," *SDxCentral*, 2024. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/inside-sdn-architecture/.

[22] GeeksforGeeks, "Software defined networking(sdn) - geeksforgeeks," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/software-defined-networking/.

[23] Comparitech, "Software-defined networking (sdn) guide: Sdn advantages - comparitech," *Comparitech*, 2024. [Online]. Available: https://www.comparitech.com/net-admin/software-defined-networking/.

[24] IEOM, "Software-defined networking: Current trends, challenges, and ... - ieom," *IEOM*, 2024. [Online]. Available: http://ieomsociety.org/dc2018/papers/435.pdf.

[25] Deliveredsocial, "Software defined networking (sdns) - benefits, challenges & applications," *Deliveredsocial*, 2024. [Online]. Available: https://deliveredsocial.com/software-defined-networking-sdns-benefits-challenges-applications/.

[26] Hitechwhizz, "5 advantages and disadvantages of sdn — drawbacks & benefits of sdn," *Hitechwhizz*, 2024. [Online]. Available: https://www.hitechwhizz.com/2021/06/5-advantages-and-disadvantages-drawbacks-benefits-of-sdn.html.

[27] Huawei, "What is openflow? how does it relate to sdn? - huawei," *Huawei*, 2024. [Online]. Available: https://www.huawei.com/en/industry-insights/technology/digital-transformation/what-is-openflow.

[28] NoviFlow, "The basics of sdn and the openflow network architecture," *NoviFlow*, 2024. [Online]. Available: https://noviflow.com/the-basics-of-sdn-and-the-openflow-network-architecture/.

[29] HowToForge, "Software defined networking (sdn) - architecture and role of openflow," *HowToForge*, 2024. [Online]. Available: https://www.howtoforge.com/tutorial/software-defined-networking-sdn-architecture-and-role-of-openflow/.

[30] Wikipedia, "Openflow - wikipedia," *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/OpenFlow.

[31] SDxCentral, "What is an openflow controller? - sdxcentral .com," *SDxCentral*, 2024. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/what-is-sdn-controller/openflow-controller/.

[32] Cisco, "Openflow benefits vs openflow drawbacks. - cisco learning network," *Cisco Learning Network*, 2024. [Online]. Available: https://learningnetwork.cisco.com/s/question/0D53i00000KsoLvCAJ/openflow-benefits-vs-openflow-drawbacks.

[33] 1Library, "Openflow limitations - project and openflow limitations - 1library," *1Library*, 2024. [Online]. Available: https://1library.net/article/openflow-limitations-project-and-openflow-limitations.y44pvg9y.

[34] W. Braun and M. Menth, "Software-defined networking using openflow: Protocols, applications and architectural design choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014, ISSN: 1999-5903. DOI: 10.3390/fi6020302. [Online]. Available: https://www.mdpi.com/1999-5903/6/2/302.

[35] A. Azzouni, N. T. Mai Trang, R. Boutaba, and G. Pujolle, "Limitations of openflow topology discovery protocol," pp. 1–3, 2017. DOI: 10.1109/MedHocNet.2017.8001642.

[36] N. World, "What p4 programming is and why it's such a big deal for software defined networking — network world," *Network World*, 2024. [Online]. Available: https://www.networkworld.com/article/959851/what-p4-programming-is-and-why-it-s-such-a-big-deal-for-software-defined-networking.html.

[37] PLVision, "Is p4 programming the future of sdn? - plvision," *PLVision*, 2024. [Online]. Available: https://plvision.eu/blog/sdn/p4-programming-future-sdn.

[38] O. Research, "Onrc research," *ONRC Research*, 2024. [Online]. Available: https://onrc.stanford.edu/p4.html.

[39] VMware, "Programming networks with p4 — office of the cto blog," *VMware*, 2024. [Online]. Available: https://octo.vmware.com/programming-networks-with-p4/.

[40] Y. Gao and Z. Wang, "A review of p4 programmable data planes for network security," *Hindawi*, 2024. [Online]. Available: https://www.hindawi.com/journals/misy/2021/1257046/.

[41] Google, "Protocol buffers," [Online]. Available: https://developers.google.com/protocol-buffers.

[42] Google, "Grpc," [Online]. Available: https://grpc.io.

[43] Vaadin, "Role of grpc," 2022. [Online]. Available: https://vaadin.com/docs/v8/framework/gwt/gwt-rpc.

[44] OSEDEA, "Communication with protobuf and grpc," [Online]. Available: https://www.osedea.com/insight/streamlining-communication-with-protobuf-and-grpc.

[45] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," 2014, P4: Programming Protocol-Independent Packet Processors introduces the P4 language, a high-level language for programming the data plane of network devices. P4 enables network operators to specify the behavior of packet processing devices, allowing for flexible and programmable forwarding logic. [Online]. Available: https://dl.acm.org/doi/10.1145/2656877.2656890.

[46] R. Giladi, "Network processors: Architecture, programming, and implementation," *Morgan Kaufmann; 1st edition*, 2008, Network Processors: Architecture, Programming, and Implementation provides an in-depth overview of network processors, specialized hardware components designed for executing packet processing tasks in network devices. The paper discusses the architecture, programming models, and implementation considerations of network processors.

[47] The P4.org API Working Group, *P4runtime specification*, https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html#fig-embedded-plus-two-remote-ha-controllers, Version 1.3.0, Jul. 2021.

[48] E. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez, and J. Crichigno, *A comprehensive survey on smartnics: Architectures, development models, applications, and research directions*, 2024. arXiv: 2405.09499 [cs.NI].

[49] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, and J. M. Smith, "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*, ser. CoNEXT '20, Barcelona, Spain: Association for Computing Machinery, 2020, pp. 336–350, ISBN: 9781450379489. DOI: 10.1145/3386367.3431290. [Online]. Available: https://doi.org/10.1145/3386367.3431290.

[50] G. Li, M. Dong, K. Ota, J. Wu, J. Li, and T. Ye, "Deep packet inspection based application-aware traffic control for software defined networks," in *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841721.

[51] T. Chin, X. Mountrouidou, X. Li, and K. Xiong, "Selective packet inspection to detect dos flooding using software defined networking (sdn)," in *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*, 2015, pp. 95–99. DOI: 10.1109/ICDCSW.2015.27.

Chapter 9

Appendices

This chapter describes how to reproduce the results of this thesis. The developed code, along with other necessary files, is available through the following GitHub link Access_Thesis_on_Github that contains the developed framework. Additionally, we offer an OVF image containing all code, dependencies, and scripts prepared for immediate execution. Alternatively, you can set up your environment by cloning the main directories from the repository.

9.1 OVF Image Route To Reproduce The Results

The provided OVF image is based on Ubuntu, with the password for all users set as `uwbothell`. Within the directory `/home/anthony/`, you will find two directories containing all the necessary code, dependencies, and scripts to conduct the tests discussed in this thesis.

9.1.1 HTTP_TEST Directory

This directory includes three subdirectories corresponding to the three different implementations explored in the thesis.

Figure 9.1: Overview of HTTP project directory structure

9.1.2 Testing The P4runtime-based Implementation (Directory p4runtime_based_implementation)

The process to test the P4runtime_based implementation involves several steps:

1. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.
2. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1 h2` to open terminal windows for hosts h1 and h2.
3. In each console, disable receive and transmit offload using: `ethtool --offload h1-eth0 rx off tx off` and similarly for `h2-eth0`.
4. On h2's console, start the HTTP server with: `python3 http_dpi_server.py`.

5. Execute `run_controller.sh` to start the p4runtime controller that contains the HTTP filtering application as dependency to filter out domains.

6. On h1's console, execute `benchmarking.sh` to run the tests.

7. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

Note: To exist the p4_runtime controller you must press "s", be careful not to press it if you do not want to exit.

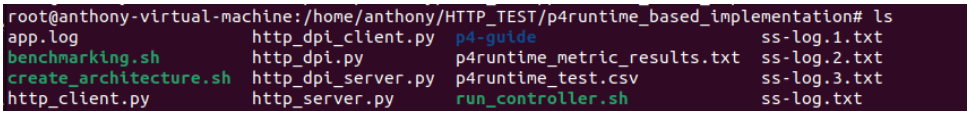

Figure 9.2: Needed files for the P4runtime-based Implementation

The results will look like figure 9.3

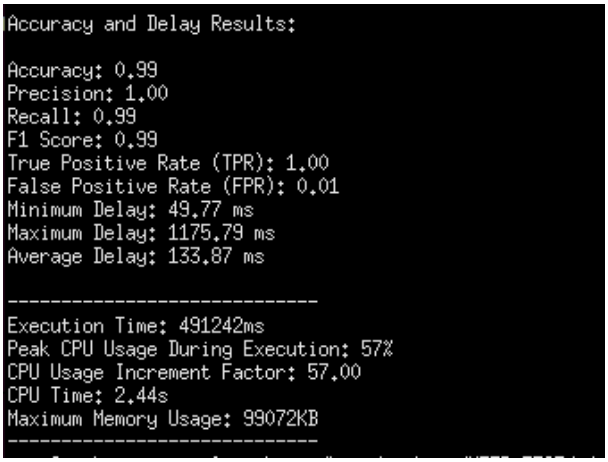

Figure 9.3: P4runtime_based implementation results

9.1.3 Testing The Openflow-based Implementation (Directory openflow_based_implementation)

The process to test the Openflow_based implementation involves several steps:

1. Execute `run_controller.sh` to start the POX controller that contains the HTTP filtering application as a dependency to filter out domains.

2. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.

3. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1 h2` to open terminal windows for hosts h1 and h2.

4. On h2's console, start the HTTP server with: `python3 http_dpi_server.py`.

5. On h1's console, execute `benchmarking.sh` to run the tests.

6. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

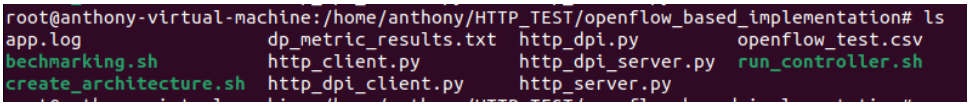

Figure 9.4: Needed files for the OpenFlow-based Implementation

The results will look like figure 9.5

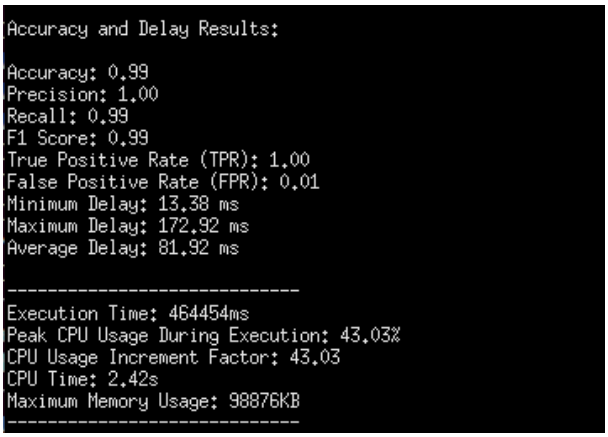

Figure 9.5: Openflow_based implementation results

9.1.4 Testing The Data Plane Implementation (Directory data_plane_implementation)

The process to test the Data Plane implementation involves several steps:

1. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.
2. Execute `architecture_dependencies.sh` to connect interfaces to the network and add necessary table entries for the BMv2 switch.
3. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1 h2` to open terminal windows for hosts h1 and h2.
4. In each console, disable receive and transmit offload using: `ethtool --offload h1-eth0 rx off tx off` and similarly for `h2-eth0`.
5. On h2's console, start the HTTP server with: `python3 http_dpi_server.py`.
6. On h1's console, execute `benchmarking.sh` to run the tests.
7. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

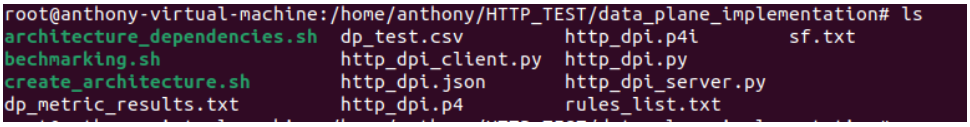

Figure 9.6: Needed files for the Data Plane Implementation

The results will look like figure 9.7

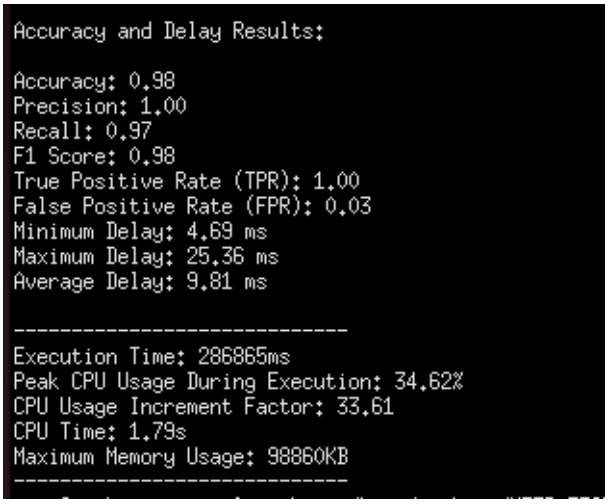

Figure 9.7: Data Plane Implementation Results

9.1.5 SQL_TEST Directory

This directory includes three subdirectories corresponding to the three different implementations explored in the thesis.

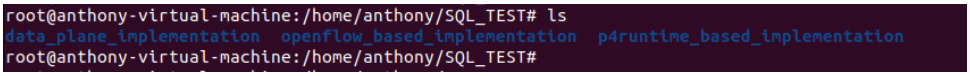

Figure 9.8: Overview of SQL project directory structure

9.1.6 Testing The P4runtime-based Implementation (Directory p4runtime_based_implementation)

The process to test the P4runtime$_{b}asedimplementationinvolvesseveralsteps$:

1. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.
2. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1` to open terminal windows for hosts h1. Note: In this step, we are not opening the h2 console, because we are connecting our BMv2 switch to a VMware machine that is hosting the MySQL Database, we could use any other solution to host our database like a real machine or docker.
3. In h1 console, disable receive and transmit offload using: `ethtool --offload h1-eth0 rx off tx off`.
4. Execute `run_controller.sh` to start the p4runtime controller that contains the SQL filtering application as a dependency to filter out domains.
5. On h1's console, execute `benchmarking.sh` to run the tests.
6. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

Note: To exist the p4_runtime controller you must press "s", be careful not to press it if you do not want to exit. Additionally to collect the results for accuracy and delay you have to read the .csv files

"accuracy_sql.csv" and "command_response_times.csv", make sure to not have these two .csv files before executing the script `benchmarking.sh`.

```
root@anthony-virtual-machine:/home/anthony/SQL_TEST/p4runtime_based_implementation# ls
accuracy_sql.csv      accuracy_sql_utility.py     delay_test.py       __pycache__           ss-log.2.txt
accuracy_sql_dcl.py   accuracy_test.py            http_client.py      run_controller.sh     ss-log.3.txt
accuracy_sql_ddl.py   app.log                     http_dpi_client.py  sql_ddl_commands.py   ss-log.txt
accuracy_sql_dml.py   benchmarking.sh             http_dpi_server.py  sql_dpi.py
accuracy_sql_show.py  command_response_times.csv  http_server.py      sql_show_commands.py
accuracy_sql_tc.py    create_architecture.sh      p4-guide            ss-log.1.txt
root@anthony-virtual-machine:/home/anthony/SQL_TEST/p4runtime_based_implementation#
```

Figure 9.9: Needed files for the P4runtime-based Implementation

The results will look like figure 9.10

```
----------------------------
Execution Time: 63535ms
Peak CPU Usage During Execution: 65.44%
CPU Usage Increment Factor: 40.90
CPU Time: 2.38s
Maximum Memory Usage: 92368KB
----------------------------
```

Figure 9.10: Enter Caption

9.1.7 Testing The Openflow-based Implementation (Directory openflow_based_implementation)

The process to test the Openflow$_{b}asedimplementationinvolvesseveralsteps$:

1. Execute `run_controller.sh` to start the POX controller that contains the SQL filtering application as dependency to filter out domains.
2. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.
3. Then execute the command `ovs-vsctl add-port s1 ens37` that is going to add the interface ens37 to the OpenV Switch, you should add whichever interfaces connecting your Database instead.
4. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1` to open terminal windows for hosts h1. Note: In this step, we are not opening the h2 console, because we are connecting our Openv Switch to a VMware machine that is hosting the MySQL Database, we could use any other solution to host our database like a real machine or docker.
5. On h1's console, execute `benchmarking.sh` to run the tests.
6. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

Note: To collect the results for accuracy and delay you have to read the .csv file "accuracy_sql.csv" and "command_response_times.csv", make sure to not have these two .csv files before executing the script `benchmarking.sh`.

```
root@anthony-virtual-machine:/home/anthony/SQL_TEST/openflow_based_implementation# ls
 accuracy_sql.csv       accuracy_sql_utility.py     'delay_test copy.py'   __pycache__
 accuracy_sql_dcl.py    accuracy_test.py            delay_test.py          run_controller.sh
 accuracy_sql_ddl.py    app.log                     http_client.py         sql_ddl_commands.py
 accuracy_sql_dml.py    benchmarking.sh             http_dpi_client.py     sql_dpi.py
 accuracy_sql_show.py   command_response_times.csv  http_dpi_server.py     sql_show_commands.py
 accuracy_sql_tc.py     create_architecture.sh      http_server.py         sql-topology.py
root@anthony-virtual-machine:/home/anthony/SQL_TEST/openflow_based_implementation#
```

Figure 9.11: Needed files for the OpenFlow-based Implementation

The results will look like figure 9.12

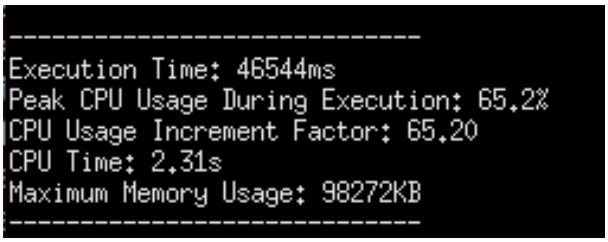

Figure 9.12: Openflow_based implementation results

9.1.8 Testing The Data Plane Implementation (Directory data_plane_implementation)

The process to test the Data Plane implementation involves several steps:

1. Run the bash script `create_architecture.sh` to set up the network architecture for the tests.
2. Execute `architecture_dependencies.sh` to connect interfaces to the network and add necessary table entries for the BMv2 switch.
3. Upon executing `create_architecture.sh`, a Mininet console will appear. Use the command `xterm h1` to open terminal windows for hosts h1.
4. In the console, disable receive and transmit offload using: `ethtool --offload h1-eth0 rx off tx off`. Note: In this step, we are not opening the h2 console, because we are connecting our BMv2 switch to a VMware machine that is hosting the MySQL Database, we could use any other solution to host our database like a real machine or docker.
5. On the virtual machine we have running our MySQL database.
6. On h1's console, execute `benchmarking.sh` to run the tests.
7. Once `benchmarking.sh` completes, collect the results for accuracy, delay, CPU consumption, memory consumption, etc.

Note: To collect the results for accuracy and delay you have to read the .csv file "accuracy_sql.csv" and "command_response_times.csv", make sure to not have these two .csv files before executing the script `benchmarking.sh`.

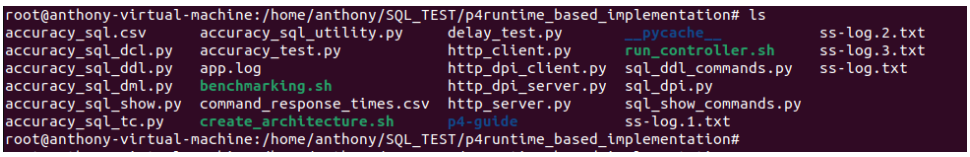

Figure 9.13: Needed files for the Data Plane Implementation

The results will look like figure 9.14

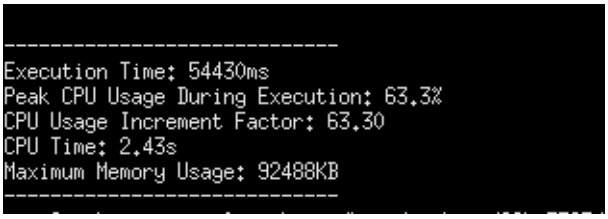

Figure 9.14: Data Plane Implementation Results

9.1.9 About MySQL Database

The Database that we used for out tests is not in the provided OVF image, therefore, to execute the SQL tests successfully we will have to import the dump file that contains the squema, tables, etc. in regards to the DB. Besides, when importing the database make sure to set up this information:

1. 'user': 'username'
2. 'password': 'password'
3. 'host': '192.168.118.136' (IP of the host containing the DB)
4. 'database': 'sqli'

If you decide to opt for configuring different username, password, host, or database name, you will have to update the parameters in the python programs. Also, note that the MySQL Database is not encrypting the communication. Hence, you will have to disable SSL/TLS for the tests.

Appendix A: Database Schema Dump

Below is the complete MySQL dump from the database 'sqli', which includes detailed table structures, data insertion commands, and system settings. This dump is crucial for reproducing the database environment necessary for the experiments conducted in this thesis. Note: You can download the file from the provided GitHub repository.

```
-- MySQL dump 10.13 Distrib 8.0.36, for Linux (x86_64)
-- Host: localhost Database: sqli
-- ------------------------------------------------------
-- Server version 8.0.36-0ubuntu0.22.04.1

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

-- Table structure for table `coffee_table`
DROP TABLE IF EXISTS `coffee_table`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
```

```
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `coffee_table` (
  `id` int DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  `region` varchar(255) DEFAULT NULL,
  `roast` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

-- Dumping data for table `coffee_table`
LOCK TABLES `coffee_table` WRITE;
/*!40000 ALTER TABLE `coffee_table` DISABLE KEYS */;
INSERT INTO `coffee_table` VALUES
(1, 'default route', 'Ethiopia', 'light'),
(1, 'Colombian', 'South America', 'Medium'),
(2, 'Ethiopian Yirgacheffe', 'Africa', 'Light'),
(3, 'Sumatra Mandheling', 'Indonesia', 'Dark'),
(4, 'Guatemala Antigua', 'Central America', 'Medium'),
(5, 'Kenya AA', 'Africa', 'Medium'),
(6, 'Costa Rica Tarrazu', 'Central America', 'Medium');
/*!40000 ALTER TABLE `coffee_table` ENABLE KEYS */;
UNLOCK TABLES;

-- Table structure for table `countries`
DROP TABLE IF EXISTS `countries`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `countries` (
  `id` int NOT NULL AUTO_INCREMENT,
  `country_name` varchar(255) DEFAULT NULL,
  `capital` varchar(255) DEFAULT NULL,
  `government_type` varchar(255) DEFAULT NULL,
  `population` int DEFAULT NULL,
  `area_km2` decimal(10,2) DEFAULT NULL,
  `currency` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
    ↪ ;
/*!40101 SET character_set_client = @saved_cs_client */;

-- Dumping data for table `countries`
LOCK TABLES `countries` WRITE;
/*!40000 ALTER TABLE `countries` DISABLE KEYS */;
INSERT INTO `countries` VALUES
(1, 'United States', 'Washington D.C.', 'Federal Republic', 331002651, 9833517.85,
    ↪ 'USD'),
(2, 'United Kingdom', 'London', 'Constitutional Monarchy', 67886011, 242495.00, '
    ↪ GBP'),
(3, 'France', 'Paris', 'Semi-Presidential Republic', 65273511, 551695.00, 'EUR'),
(4, 'Germany', 'Berlin', 'Federal Republic', 83783942, 357022.00, 'EUR'),
(5, 'Japan', 'Tokyo', 'Constitutional Monarchy', 126476461, 377975.00, 'JPY');
```

```
/*!40000 ALTER TABLE `countries` ENABLE KEYS */;
UNLOCK TABLES;

-- Table structure for table `dpi2`
DROP TABLE IF EXISTS `dpi2`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `dpi2` (
  `id` int NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

-- Dumping data for table `dpi2`
LOCK TABLES `dpi2` WRITE;
/*!40000 ALTER TABLE `dpi2` DISABLE KEYS */;
/*!40000 ALTER TABLE `dpi2` ENABLE KEYS */;
UNLOCK TABLES;

-- Table structure for table `test1`
DROP TABLE IF EXISTS `test1`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `test1` (
  `id` int NOT NULL AUTO_INCREMENT,
  `column1` varchar(255) DEFAULT NULL,
  `column2` varchar(255) DEFAULT NULL,
  `column3` varchar(255) DEFAULT NULL,
  `column4` varchar(255) DEFAULT NULL,
  `column5` varchar(255) DEFAULT NULL,
  `column6` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
    ↪ ;
/*!40101 SET character_set_client = @saved_cs_client */;

-- Dumping data for table `test1`
LOCK TABLES `test1` WRITE;
/*!40000 ALTER TABLE `test1` DISABLE KEYS */;
INSERT INTO `test1` VALUES
(1, 'data1_1', 'data1_2', 'data1_3', 'data1_4', 'data1_5', 'data1_6'),
(2, 'data2_1', 'data2_2', 'data2_3', 'data2_4', 'data2_5', 'data2_6'),
(3, 'data3_1', 'data3_2', 'data3_3', 'data3_4', 'data3_5', 'data3_6'),
(4, 'data4_1', 'data4_2', 'data4_3', 'data4_4', 'data4_5', 'data4_6'),
(5, 'data5_1', 'data5_2', 'data5_3', 'data5_4', 'data5_5', 'data5_6');
/*!40000 ALTER TABLE `test1` ENABLE KEYS */;
UNLOCK TABLES;

-- Footer settings to restore system settings
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
```

```
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2024-05-19 19:39:03
```

9.2 Create Your Own Environment To Reproduce The Results

If on the contrary you don't want to use the provided OVF image, you can also set up your own environment.

Software Installation Guide

These are some of the things to install to have the adequate environment for the tests.

1. **Basic Tools Installation**
   - Command: `sudo apt install git wget vim ethtool`
2. **Mininet Installation**
   - GitHub: https://github.com/mininet/mininet
3. **POX Controller Installation**
4. **Thrift Installation**
   - Required for BMv2.
   - Installation guide: Install Thrift
5. **BMv2 (Behavioral Model v2) Installation**
   - Necessary for running P4 environments.
   - GitHub: https://github.com/p4lang/behavioral-model
6. **gRPC and Protobuf Installation**
   - Needed for communication between P4 devices and controllers.
   - Protobuf GitHub: https://github.com/protocolbuffers/protobuf
   - gRPC GitHub: https://github.com/grpc/grpc
   - Installation steps are available on their respective GitHub repositories.
7. **Docker Installation**
   - For container management.
   - Installation guide: Install Docker
8. **Additional Commands and Software**
   - Configure and manage SDN controllers, switches, and environments using provided scripts and command-line utilities.

Once we have out environment we will have to clone the provided GitHub repository GitHub_repository.

Post-Repository Cloning Instructions

1. **P4-Guide Setup:** After cloning the repository, check for a sub-folder named "p4-guide" within the directories "HTTP_TEST/p4runtime_based_implementation" and "SQL_TEST/p4runtime_based_implementation". If this sub-folder is absent, you should clone it from this link. Once downloaded, replace the "flowcache" folder within the newly downloaded "p4-guide" with the one from "p4runtime_based_implementation".

2. **POX Controller Configuration:** Upon successful installation of the POX controller on your OS, you will have to add all the files contained in the folder "POX_Applications" in the GitHub repository to the pox project you just installed. Add the files to the corresponding sub-folders.

3. **Final Integration Steps:** Once the above configurations are complete, follow the steps analogous to those provided for the OVF image to ensure a consistent setup across different environments. This includes running the scripts and others as detailed in the first section of the appendices.