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 curtail unauthorized web access by intercepting and analyzing HTTP requests based on their URLs. The control plane employs a 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
    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
    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 & 0xffffff000000000000000000000000000`	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 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.

6.3.1 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).

Accuracy Test Results

The accuracy metrics table (Table 6.5) details the effectiveness of each DPI implementation in filtering HTTP traffic. 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 6.10) compares the minimum, average, and maximum response times experienced by each DPI implementation when processing HTTP requests. The Data Plane-Centric DPI shows superior performance with significantly reduced latency, enhancing its suitability for environments where swift data processing and decision-making are paramount.

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 6.12 showcases the performance variances across different DPI implementations. The 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 Data Plane-Centric approach. This analysis highlights the 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 6.12 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 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. In summary, the experimental results validate the efficacy of the proposed DPI strategies, with each catering to different network conditions and operational priorities. Future work may explore further optimization techniques to enhance scalability and reduce latency across more diverse network topologies.

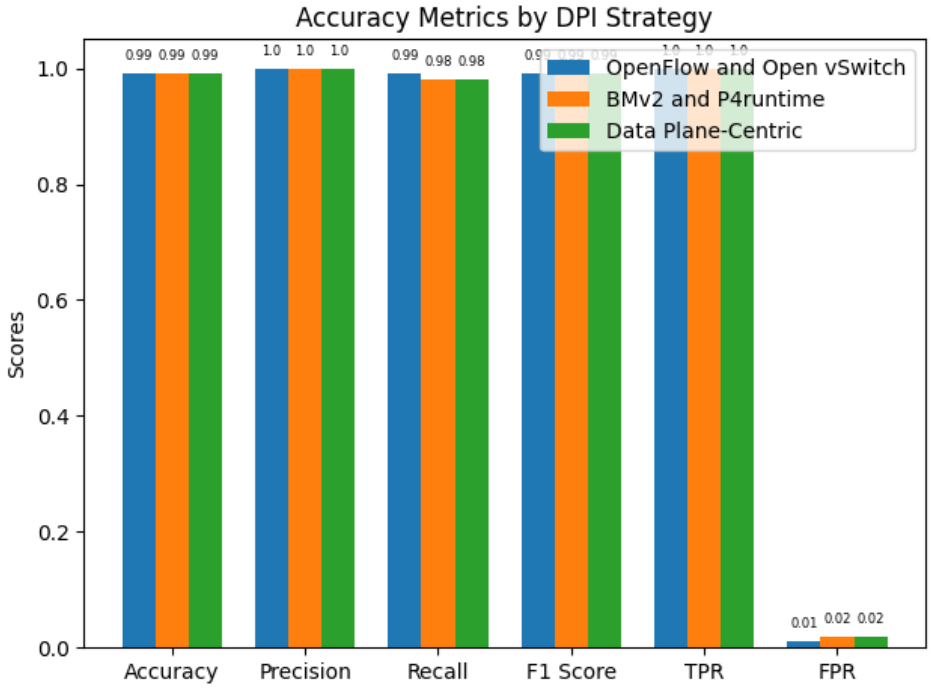

Figure 6.1: Accuracy comparison for all the implementations

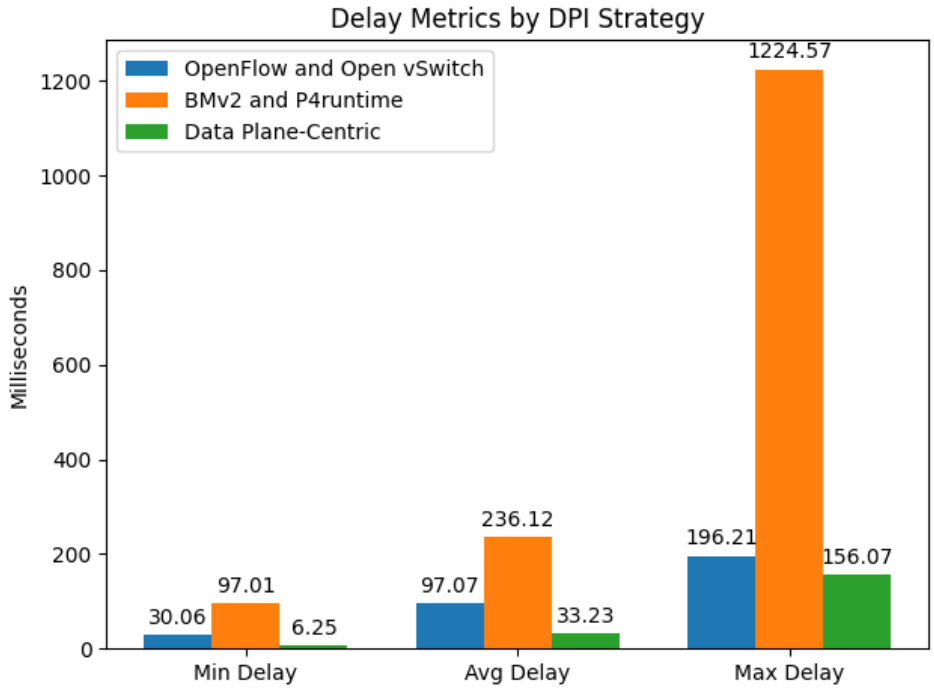

Figure 6.2: delay comparison for all the implementations

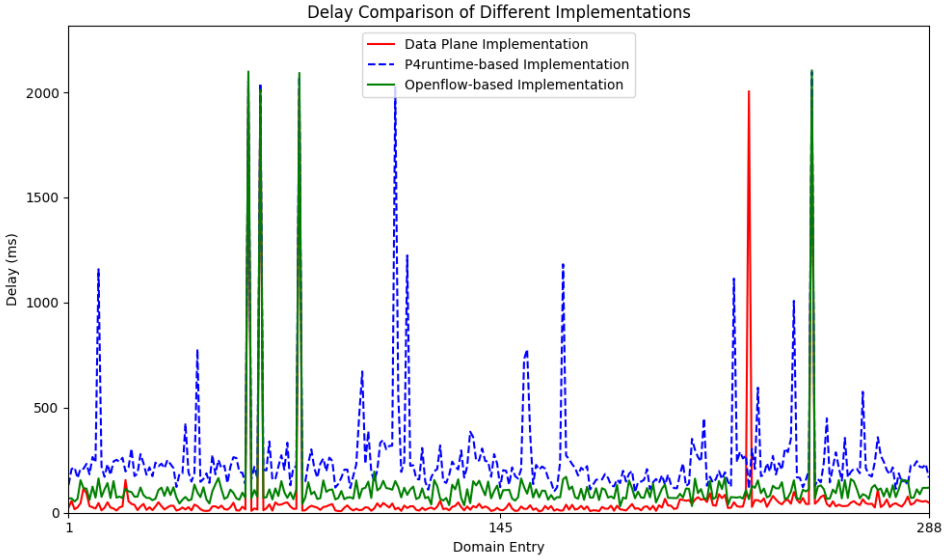

Figure 6.3: Delay comparison for our URL filtering application for the different implementations

6.3.2 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 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 6.8

Table 6.8: 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 filter or not. To simplify the test we blocked all the SQL commands in out applications 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 6.9:

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.

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

Delay Test Results The delay introduced by SQL command filtering is tabulated below in table 6.10 for each DPI implementation. These metrics include the minimum, average, and maximum response times recorded during the tests.

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 6.10: 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 6.11 summarizes these metrics.

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

Table 6.11: 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 6.12 reveals that while the OpenFlow-based DPI is the most responsive due to its shortest execution time and lowest CPU time, the 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 Data Plane-Centric DPI implementation is recommended as the better choice.

Table 6.12: 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 6.12 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 Data plane implementation is the one that introduces less delay in 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 Data plane implementation produced the best results.

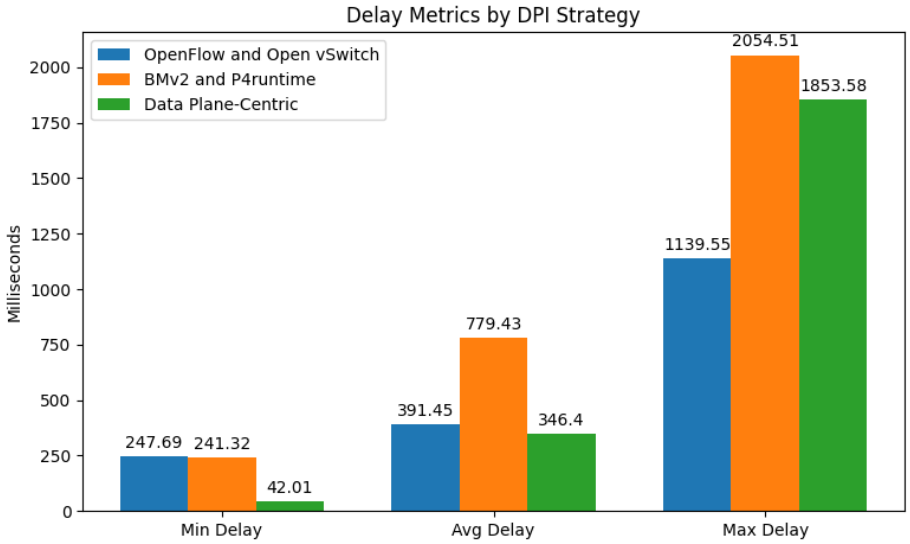

Figure 6.4: Minimum, Average, and Maximum delay comparison for the different implementations

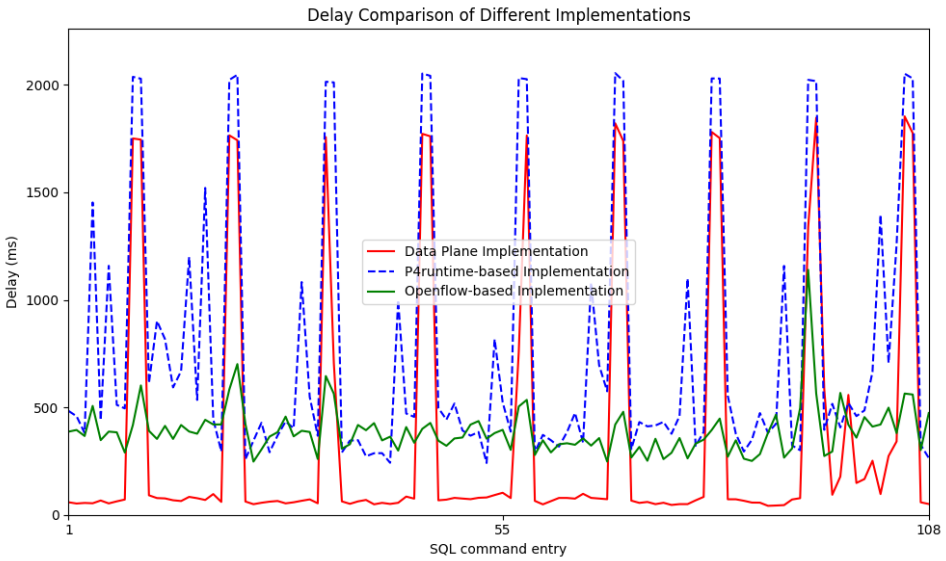

Figure 6.5: Delay comparison for our SQL command filtering application for the different implementations

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 6.9: SQL Commands Categorized by Their Properties