Predictable Internet Clients and In-Switch Deep Packet Inspection

Sahil Gupta
Rochester Institute of Technology
Rochester, NY, USA
sg5414@rit.edu

Devashish Gosain
Max Planck Institute for Informatics
Saarbrucken, Germany
dgosain@mpi-inf.mpg.de

Minseok Kwon
Rochester Institute of Technology
Rochester, NY, USA
jmk@cs.rit.edu

Hrishikesh B Acharya
Rochester Institute of Technology
Rochester, NY, USA
acharya@mail.rit.edu

***Abstract*—Deep packet inspection (DPI) is important for network security and is currently provided by complex black-box firewalls. This raises the question: Can network administrators build their own DPI-capable filter using a standard programmable switch? The common answer is that standard switches support P4, which allows users to specify how to parse packet headers, but not packet payload fields (e.g. URL) – thus DPI tasks, like URL filtering, require dedicated middleboxes.**

In this paper, we challenge this common answer. First, we demonstrate that clients send packets with a predictable structure, so a P4 switch *can* perform some DPI (enough for URL filtering). Second, we demonstrate a URL-filtering firewall completely in the data plane, with no external help from the SDN controller, firewalls, *etc.* and no custom logic. Our proof-of-concept, P4Wall, handles multiple protocols (HTTP, HTTPS, DNS) with high performance – orders of magnitude faster than a standard Linux (netfilter) firewall.

***Index Terms*—P4 language, programmable switch, Firewall**

I. Introduction

Deep Packet Inspection (DPI), i.e., an inspection of the *payload* of network packets, is essential for complex security tasks such as URL filtering, detecting signatures of attacks such as Heartbleed [1], or matching a virus signature in a downloaded file. However, DPI is expensive: unlike simple IP-based filtering or flow analysis, which are efficiently performed by programmable switches[1], DPI requires dedicated middleboxes such as Cisco Firepower [4], SonicWALL [5], or Fortinet FortiGate [6]. And such firewalls introduce issues of their own: they are black boxes to the network administrator, they present a high-value target for attacks, and they compromise many users when they leak.

We note that there exists a standard language (P4) that allows users to specify the schema of a packet header, and give a programmable switch the ability to parse these headers[2]. If a switch can (extract and) filter traffic by application-layer headers, *e.g.* site URL or file type, it becomes an application layer firewall, *i.e.* performs DPI. The question immediately arises why such solutions do not replace black-box firewalls.

Indeed, such ideas have been proposed – for example, Sekar's CoMB architecture [7] built on the Click modular router [8]. But *current SDN platforms are not intended for Deep Packet Inspection*. The $P4_{16}$ standard makes this explicit [9].

- P4 is not a Turing-complete language; the P4 packet "parser" really just extracts slices of bits ("slice" meaning, a given length at a given offset). The parser cannot loop, and cannot properly handle the following cases:
  - Fields of variable length.
  - Fields which may or may not be present.
  - Fields present in random order.
- Headers of important application-layer protocols, such as HTTP, do present all the above cases. (HTTP has 47 fields, which are mostly optional; important fields for filtering, such as URL, are variable-length).

Thus while P4-compatible switches have some flexibility, *general DPI is beyond their scope*. If a network admin wishes to build their own DPI-capable infrastructure, the consensus is that they must *either* use specialized platforms – *e.g.* NVIDIA DPU [10], custom switches with non-standard extensions (`extern` logic implemented on NetFPGA), *etc.*– *or* they can have the switch outsource some work to an external server [11], and provide enough servers to process traffic at line rate. It is hardly surprising that enterprise and ISP admins prefer standard commercial middleboxes.

At this point, we make two important observations.

[1]Programmable switches can carry out computations on network and transport layer headers, such as source IP and port, destination IP and port, protocol, etc. This is sufficient for various network security tasks, such as the detection of port scans [2] and DoS attacks [3].

[2]In a P4 program, the user defines the structure of packets of a protocol. A switch loaded with the appropriate definition can parse headers of novel protocols just like TCP or IP headers, but *subject to some restrictions*, as we discuss in detail below.

- An application-layer firewall can be valuable even if it only performs a few simple cases of DPI. Content censorship *e.g.* social media or email, is usually performed with the help of an end-point on the provider or the client. For the common case in traffic inspection – blocklisting of websites – it is sufficient to detect the URL.
- The URL is usually present in plaintext in HTTP traffic, HTTPS traffic (the Server Name Indication field), and DNS traffic. If it is present *at a predictable position* in network packets, this common-case DPI can indeed be solved.

In other words: even if it is challenging to build a *fully general application-layer firewall* in the data plane, it may be possible to build a *URL filter for traffic from practical Internet clients*. But to build such a filter using a standard P4-programmable SDN switch, we must be sure that *the URL consistently appears at a predictable location in packets*. This brings us to the two contributions of this paper.

1) We demonstrate with a field study (Section IV) that in actual web traffic, the header has a predictable structure, even for theoretically "free-form" protocols such as HTTP(S). These protocols *can* reliably be parsed in the data-plane.
   In other words, even simple SDN switches (not designed for DPI) can perform URL filtering, thanks to the predictability of browser clients (and the lack of adoption of secure protocols such as encrypted SNI and DNS-over-TLS).
2) We develop P4Wall, a pure data-plane firewall capable of simple deep packet inspection, on a simple consumer-grade SDN switch (Netberg Aurora 710) [12] [3]. P4Wall is able to block URLs directly in the data plane for multiple protocols – HTTP, HTTPS, and DNS.
   In terms of *scalability* and *performance*, P4Wall greatly outperforms a standard software firewall (Linux netfilter): it works smoothly when filtering 1000 URLs from 10k parallel traffic flows, with a near-zero packet processing delay ($< 0.05ms$), and showing no degradation under 10 Gbps cross-traffic load.

P4Wall currently matches URLs from various protocols (HTTP, HTTPS, DNS) with line-rate performance, and could in future be extended to match keywords or other specific strings. As we will make both our P4 code (for the Tofino switch) and all related scripts *etc.* available to researchers for future study or extension, we expect that SDN-based firewalls with DPI capability may indeed see practical use in the future.

This paper begins with background about P4 and programmable switches in Section II, and the overview and challenges for our system in Section III. We then move on to our main sections: our field study in Section IV, system implementation in Section V, and evaluation results in Section VI. To wrap up, we discuss limitations and future work in Section VII and related work in Section VIII, and end with some concluding remarks.

[3]Our implementation runs on the Intel P4-based Tofino ASIC [13], but it can be ported very simply to another platform such as the Broadcom NPL ASIC [14], as we *only use standard P4 functionality* to parse packets, extract fields, and trigger actions.

II. Background

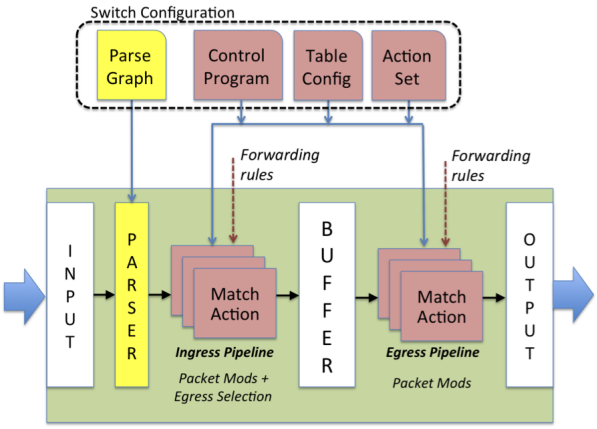

Fig. 1. Pipeline for a switch supporting P4 language [15].

A programmable data plane allows the user to write a program to parse packets from different protocols. Notably, the user can define custom headers for their own protocols – they supply a schema describing the layout of the packet headers, and the switch extracts these header fields from packets in traffic and uses them for network processing tasks (routing, load balancing *etc.*). This greatly reduces the barrier to entry when deploying new protocols. The basic architecture is provided in Figure 1.

- *Parser.* This is the programmable block where the user can specify a schema for packet parsing.
  The parser treats the packet as a string and extracts header fields as sub-strings (of a given length and starting at a given offset). It handles multiple protocols, in the same traffic, through stateful parsing of a packet. As it passes down the packet extracting headers, the information seen so far determines what headers it expects next.
  While the parser allows the user to define a protocol header schema as they choose (hence the full form of P4: programming protocol-independent packet processors), *such a schema does not expect optional, variable-length, or variable-position fields*. This constraint makes it very difficult to parse application layer protocols, in which the header fields are indeed of variable length and position.
  Another challenge is that if the schema must match exactly. If the packet is shorter than the parse length expected for the protocol, it is dropped by the switch.
- *Control block (Match-action tables).* The control block implements user-defined policies for packet classification. As seen in the figure, the control block mainly consists of match-action tables, where keys – fields extracted by the parser, as well as *metadata* – are used to look up the appropriate action for the packet. The action can be to drop a packet, to set a target egress port for output, *etc.* These tables are set by the control plane.
  The control block also provides some facilities for storage and computation, namely registers, counters, and meters.

Registers are essentially variables, storing key data elements derived from the packet; counters maintain packets and byte counts; and meters are used to shape traffic flow.

- *Deparser.* A component not shown in the diagram, the deparser re-combines all the bytes of the (possibly modified) packet headers back into a packet. The user can choose to leave a particular header out of the reconstituted packet (by setting its validity bit to zero). Packets can also be targeted to multiple destinations here, *i.e.* mirrored.

We note that an actual implementation, such as our Tofino switch, provides some additional facilities. The "buffer" component in Figure 1 can be expanded into a traffic manager – which is *not* programmable using the P4 language, and also helps decide which egress port to forward the packet to. In many architectures such as PSA [16] each pipeline (ingress or egress) has its own parser, control block, and deparser, so the switch effectively gets two passes over the packet. (Note: a P4 parser cannot loop over the packet.) And most powerful of all: modular functionality can be added to the data plane, through *user defined* control blocks, which may be incorporated into either the ingress or the egress control block.[4]

As our aim is to perform deep packet inspection using only standard P4, we do not make use of any of these advanced features, user-defined control blocks, or external help from servers and firewalls. Once the switch is set up (*i.e.* it has received its P4 protocol definitions and the match-action tables from the controller), it operates independently to perform traffic filtering, using only its parser and the match-action tables of the input control block.

III. Overview and Challenges

This section introduces how we build a generic (multi-protocol) application-layer firewall in the data plane, the challenges we face, and the assumptions we make.

Our firewall takes two inputs from the network administrator: a *filtering policy i.e.* list of blocklisted URL's, and a *data definition* in the P4 language, setting out the fields that the parser should extract from packets. We begin by considering how the system would operate, if we could specify it like an IP filter, but using a different field of interest (*i.e.* URL).

- The data definition specifies the position in a packet where the field of interest (*i.e.* the URL) is present.
- The switch extracts the URL from this location, using the ingress parser, and matches it using the match-action table.
- The match-action table triggers the action, `drop`, if the URL is indeed on the block list. Otherwise, the packet does not have a URL we are blocking; it is allowed to pass.

This starting design needs to be modified because of two challenges, which we present below.

[4]Such a block has the same ingress and egress-stage resources and data available to them, but the logic is relatively free-form, so they are able to perform more complex packet processing tasks. They are generally implemented using reprogrammable hardware (NetFPGA); high-performance switches use dedicated ASIC implementations of architectures such as Tofino TNA [17], so it is difficult and expensive to provide user-defined control logic on such a switch.

A. Challenge 1. Parsing

The "parsers" in P4 are not recursive-descent parsers able to match a regular grammar. They are strictly limited to extracting bit slices from a packet, *i.e.* fixed-length header fields from known locations. This makes it difficult to extract variable-length domain names with variable start positions.

- Our first attempt was to use the P4 `varbit` data type, which the parser can use to extract a field of variable length (*e.g.* for variable-length IP or TCP headers). This approach failed, as varbits cannot be used as keys in a match-action table.
- Our next idea was to keep track of state in the parser. (The P4 parser is stateful: for example, it keeps track when it removes an Ethernet header, so it can check for IPv4, IPv6 *etc.* headers next.) Might it be possible to match an entire URL byte-by-byte, using the parser as a Finite State Machine? We found that this is very challenging: the number of parser states is small, and the number of states required for a URL-matching automaton would be very large for a non-trivial firewall.
- Our final idea was to check the range of bytes in the packet payload that could possibly contain the domain name. While the protocol definitions for say HTTP are very liberal, we suggested that *in practice, the position of URL in a HTTP(S) or DNS packet is highly predictable*. We then checked this hypothesis with a field study (Section IV).
  Our study demonstrated that while URL positions in packets were not *rigidly* predictable – there was a *range* of positions for the domain name, in DNS response, HTTP GET, and TLS client hello packets – the range was small enough to handle using case-by-case enumeration. This is the approach we adopt in this paper.

B. Challenge 2. Platform Constraints

A switch has limited computing power as well as memory. In particular, the ternary content-addressable memory (TCAM) used for match-action tables is limited, expensive, and enforces a limit on the length of the key used in rule lookup.

Our system originally used different match-action tables for HTTP, HTTPS and DNS. This system strongly limited the number of URL's filtered by P4Wall (< 200), and we considered adding the restriction that P4Wall can work only one protocol at a time, to accommodate more rules.

However, we are happy to report that a simple optimization – extracting the URL, *i.e.* match key, from different protocols with the parser *as a slice that always has the same length* (32 bytes, whether HTTP, HTTPS, or DNS) – allowed us to implement P4Wall as a single match-action table, filtering over 1000 domains per switch[5]; details follow in Section V.

IV. Field Study: Locating URLs in Packets

The main challenge in parsing packets to extract URL is that the length and position of the URL are variable; this poses an issue for the P4 parser. However, this issue may not be

[5]Our switch is a simple Netberg Aurora 710, which is available for $ 5000. An ISP switch would be far more capable.

insurmountable. In practice, *almost all users access the Web using one of a small variety of clients* [18]. If these clients generate predictably-structured packets, it is possible the URL is present in a well-defined and predictable location in DNS response, HTTP GET, and TLS client hello packets. In order to test this hypothesis, we carried out a field study, which we present in this section.

A. Model of Protocol Packets.

As shown in figure 2, we model packets as made up of four parts, of length X, Y, Z and R respectively.

- X is the length of the packet up to (and including) the layer-4 header.
- Y the length after TCP/UDP header till the start of URL (*i.e.* the "Host" field in HTTP, "Server Name Indication (SNI)" in HTTPS, "Query Name (qname)" field in DNS).
- Z is the length of the URL itself.
- R is the length of the remaining packet.

Our study attempts to characterize the starting and ending positions of domain names in the packets of different protocols, and whether they vary with browser, OS, *etc.*; in other words, we are concerned only with the variation in Y and Z.[6]

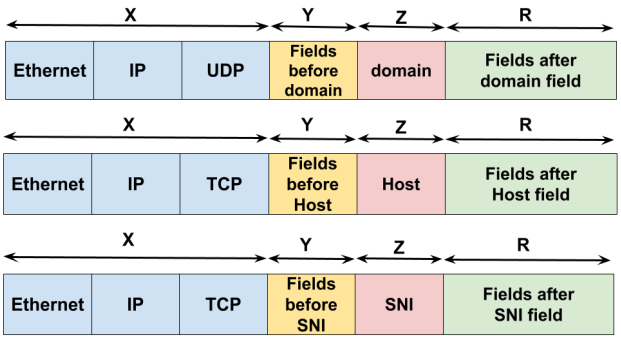

Fig. 2. Packets and their parts: DNS, HTTP, and HTTPS.

B. Field Study and Observations.

In our field study, we generated and captured traffic to the Alexa top-10k websites, using Google Chrome, Mozilla Firefox, and Microsoft Edge on Windows 10 and Ubuntu 18.04 LTS OS. Data was collected and the position of fields was measured using the Python library `scapy`. (Requests to a site often resolved into multiple sub-domain requests – *e.g.* probing `qq.com` also initiates connection to `images.qq.com`. So our analysis actually covered well over 10k domains.)

Our results appear in Table I. For example, when Firefox, Chrome, and Edge browsers on Windows 10 OS access Alexa top-10k websites, the HTTP GET requests always have URL between the 22nd and 45th byte after the TCP header.

We see that for DNS, there is no variation of the minimum starting point and maximum ending point (*i.e.* it remains consistent across OS and browsers for all sites in our study).

[6]X may have some variation caused by optional IP, TCP *etc.* fields, but this is handled using varbit fields in the parser.

	DNS		
	Firefox	**Chrome**	**Edge**
Windows	13 – 49	13 – 49	13 – 49
Linux	13 – 49	13 – 49	13 – 49
	TLS		
	Firefox	**Chrome**	**Edge**
Windows	125 – 198	125 – 161	101 – 198
Linux	125 – 161	127 – 167	127 – 163
	HTTP		
	Firefox	**Chrome**	**Edge**
Windows	22 – 45	22 – 45	22 – 45
Linux	22 – 53	22 – 53	22 – 53

TABLE I
URL POSITION IN PACKETS, AS MIN START – MAX END.

Protocol	Start	End	Parse Acc.	Match Acc.
HTTP	22	53	100	100
TLS	125	157	100	99.9
DNS	13	40	99.6	99.7

TABLE II
PARSING AND PATTERN-MATCHING ACCURACY, ALEXA TOP-10K SITES (USING THE GIVEN START AND END POSITIONS TO EXTRACT URL).

HTTP shows minimal variation. HTTPS shows more, but it is limited enough to cover by case-by-case enumeration.

C. Choosing Start and End Positions.

If Y varies in the range Y_{min} to Y_{max} and Z from Z_{min} to Z_{max}, for a given protocol, the URL is certain to lie in the range between the earliest possible start point *i.e.* Y_{min} and the last possible end point *i.e.* $Y_{max} + Z_{max}$. (Section V explains how we handle any non-URL bytes in the slice.)

However, if we naively parse packets using the ranges in Table I, we find that for some packets with short URL, the *entire packet* ends before $Y_{max} + Z_{max}$. When we ask the parser to fetch a field that *extends past the end of the packet*, it ignores the packet completely.

Our challenge is to choose start and end points such that (1) a high percentage of packets are successfully parsed (we call this metric *parse accuracy*), *and also* (2) in a high percentage of parsed packets, the URL lies between our chosen start and end points (we call this *match accuracy*) [7].

Table II shows there is indeed a sweet-spot for the length of field extracted from the packet (32 bytes for HTTPS, 31 bytes for HTTP, and 27 bytes for DNS), such that we successfully parse it from almost or exactly 100% of the target packets (high parse accuracy), and also expect it to contain the URL roughly or exactly 100% of the time (high match accuracy). In almost 100% of cases, the given start and end positions ensure that the user-defined field neither overshoots the end of packet, nor misses the URL in the packet.

[7]These goals are in tension! To increase parse accuracy we want the slice to be as narrow as possible, but to increase match accuracy we want it to be wide.

V. System Design and Implementation

In this section, we explain the system design and the actual setup of the P4Wall system.

A. Deep Packet Inspection with P4

P4Wall is a dataplane program to filter packets, by matching a given string (URL or SNI) at a known position in the packet. We make use of the Ternary Content-Addressable Memory (TCAM) match-action tables, available after the "parser" in the packet pipeline of a P4-compatible switch.

TCAM allows for constant-time retrieval of records using a ternary key value (*i.e.* the key can include don't-care bits). It is therefore a very useful standard component of switches. For example, a rule

$$10.111.*.* \rightarrow 8$$

allows for a partial match on a packet field (here, destination IP) to look up an action (here, route out on interface 8). The wildcard * indicates a don't-care byte.

In our case, the field to match is the URL, and the associated action for a successful match is to drop the packet. If the HTTP GET, HTTPS ClientHello, or DNS response packets cannot get through, this is sufficient to prevent a session with the website, so it is effectively blocked (see Figure 3).

Handling variation in URL length.

Our first challenge is that in match-action tables, the key cannot be a *varbit*, *i.e.* a field of variable length. We must ask the P4 parser to always give us a slice of the same length, though there is variation in the URL length Z. (In Alexa top-100 sites, URL length ranges from 10, for www.qq.com, to 24 for www.thestartmagazine.com.)

We respond to this challenge by asking the parser for a slice of *the optimum expected length*, as explained in the previous section. For all the shorter URLs, we pad the length of the string with don't-care matches. So for example, using "*" to represent a don't-care, we would place a rule matching the pattern
"www.qq.com**********".

Handling variation in URL start position.

The switch can remove all L2 to L4 headers effectively, so the main challenge is the variation in Y as per our model. Suppose Y varies from say 100 to 102, for a given protocol. In order to match "qq_com" starting at position 101, 102, or 103. As the parser will blindly fetch a slice from a constant start to a constant end position, we insert three patterns:
"www.qq.com**********",
"*www.qq.com**********",
"**www.qq.com**********".

In brief, as the URL could be located anywhere in a range, and one rule corresponds to a single position of the URL in the packet, we have a *range* of TCAM rules for a single URL.

B. Experimental Setup: Testbed

Our experimental setup, shown in Figure 4, is designed to test the performance of P4Wall on a real switch (Netberg Aurora 710 [12]) and also to compare it against a standard application-layer firewall (Linux netfilter).

- **Client host** : A Linux machine, set up to generate traffic and craft packets as needed.
- **Server host** : In order to serve requests for all protocols (HTTP, HTTPS and DNS), we run `Nginx` [19], `dnsmasq` [20] and `iperf` [21] in server mode on this host.
- **P4-capable switch** : Netberg Aurora 710 switch based on the Intel Tofino ASIC and natively supporting P4. The switch has a small on-board CPU running Open Network Linux (ONL), and runs the compiled output from the P4Studio IDE (*i.e.* the dataplane P4 program) as a daemon. A local process on the switch serves as SDN controller.
- **Standard Firewall** : A Linux server, set up to forward and filter traffic using the standard `netfilter` firewall, as a baseline to compare with P4Wall. To ensure that the bottleneck is netfilter, and not the network interface, we used the same 10 Gbps Network Interface Cards on the client/server and firewall machines.

C. Testing Workflow

1) Switch Setup: Our first step is to build and deploy P4Wall on the Tofino switch. The dataplane program compiles to a `tofino.bin` file to configures the pipeline, and also two JSON objects *i.e.* the contract between control plane and dataplane. Using this contract, the control plane takes the intermediate BFRT (Barefoot) policies (that declare packets with the target pattern – say **censored.com**** – should be dropped), and install these as match-action rules in the switch.

Our rules for different protocols (HTTP, HTTPS, and DNS) are *all placed in a single match-action table*, so we can fit as many domain names as possible in the switch blocklist.

This setup raises a practical issue: as per Section IV-C, the values of X, Y and Z, which determine the length of string to match, are different for different protocols. But at the same time, the key string for a single match-action table must be a uniform length. Our solution is to select a length of 32 bytes for the key, as the range for HTTPS is 32 bytes, for HTTP is 31 bytes, and for DNS 27 bytes. For DNS and HTTP, the URL is padded with additional bytes to create a 32-byte key.

The string is converted to ASCII hex representation to create a key for the match-action table. Finally, a script converts these into filtering rules as per the Tofino Barefoot API (the key triggers the action DROP, so a match causes the packet to be dropped), and the control plane installs them as match-action table rules in our switch. (Our SDN controller is a simple python process which installs our rules in the switch TCAM table, and also extracts statistics for evaluation.)

2) Traffic source and sink: Our analysis uses mixed traffic, with live client-server connections for HTTP, HTTPS and DNS. The client machine creates multiple parallel HTTP, HTTPS and DNS connections (for example, using bash scripts to run `curl` and `dig` respectively, or using `scapy` to generate TCP packets of a specific length). `iperf` is used to generate additional requests, *i.e.* cross traffic.

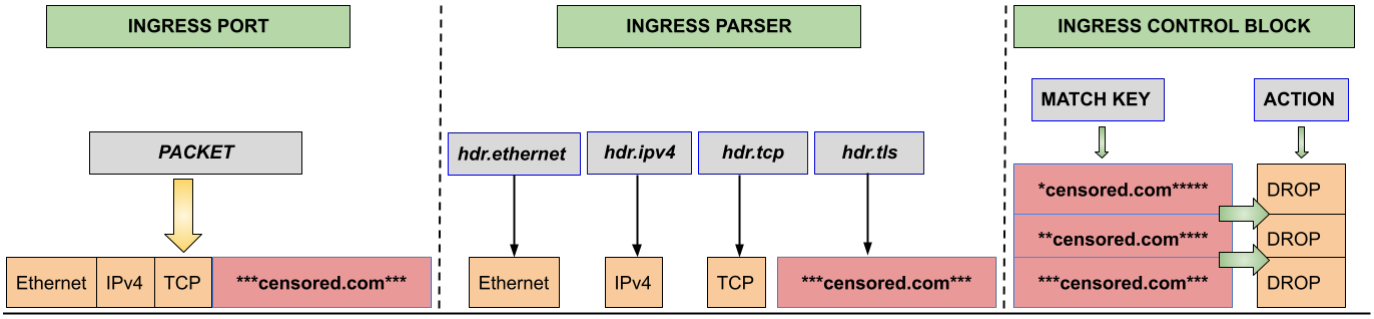

Fig. 3. P4Wall: packet parsing and matching in switch.
First, the packet arrives at the Ingress port. Next, the (ingress) parser separates different headers (*e.g.* TCP), and particularly the user-defined header containing the URL. Finally, the control block matches the user-defined header field to see if there is a rule to drop it. On a successful match, the entire packet is dropped.

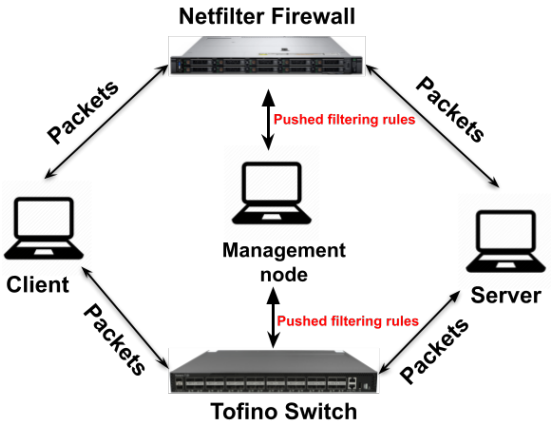

Fig. 4. Experimental setup: Client machine fetches web pages or DNS responses from Server machine. The traffic is passed through our Tofino switch (running P4Wall) and Netfilter firewall in separate runs for performance comparison.

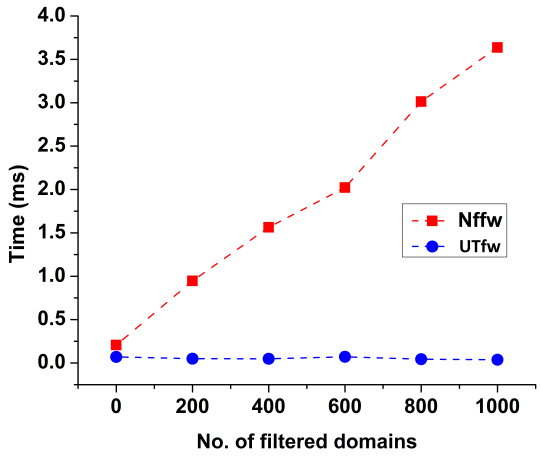

Fig. 5. Packet processing time: Avg time a packet spends within firewall.

The server machine runs `nginx` to respond to both HTTP and HTTPS requests from the client, and `dnsmasq` to handle DNS requests. It also runs `iperf` in server mode, to respond to cross-traffic requests from the client.

3) Routing: Traffic is routed through two separate network interfaces to pass through our Linux firewall or through our switch running P4Wall, as required. For all test websites (Alexa top-1k), we insert records in the client `/etc/hosts` file to redirect them to the appropriate server IP for the test (*i.e.* 10.0.0.2 for the NIC connected to the switch, and 10.0.3.1 for the NIC connected to the firewall). We also ensure the routing tables on client, server, firewall, and switch are all set up for correct forwarding of packets.

4) Firewall Setup: The netfilter firewall is deployed on a Ubuntu 20.04 LTS server, set up to filter the traffic it forwards from ingress to egress port. We ensure no other rules are installed besides our URL filters (blocking the Alexa top-1k websites, for all three target protocols).

5) Measurement Collection: To benchmark our switch and firewall, we collect packet captures (pcap) from the ingress and egress ports – *i.e.*, the client-side and server-side network interfaces, respectively – on the switch and on the firewall server, and note the difference in timestamp.

VI. Evaluation

This section covers our experimental evaluation of P4Wall (or more precisely, of our implementation of P4Wall on the P4-compatible `Netberg Aurora 710` switch, with the Intel Tofino ASIC). We assess its performance w.r.t several metrics: packet processing time and queue occupancy, throughput, and impact of packet size. To provide a baseline we use the standard Netfilter firewall (Nffw), using the same filtering rules in P4Wall and Nffw. It is shown as UTfw in the legend in the graphs. Our experiments use the setup described in Section V.

1) Packet processing time: Our first metric of interest is *packet processing time*, which we define as the average-case difference between the egress time and the ingress time for a packet passing through the switch.

We generate test traffic using standard clients (browsers and `curl`) to access web pages hosted on our web server/DNS server machine, using both HTTP and HTTPS protocols. DNS traffic is generated using the client `dig`.

Figure 5 shows the average response time over 10 webpage accesses, for (1) DNS queries (2) HTTP GET requests and (3) TLS client hello packets. As we add more rules, the packet

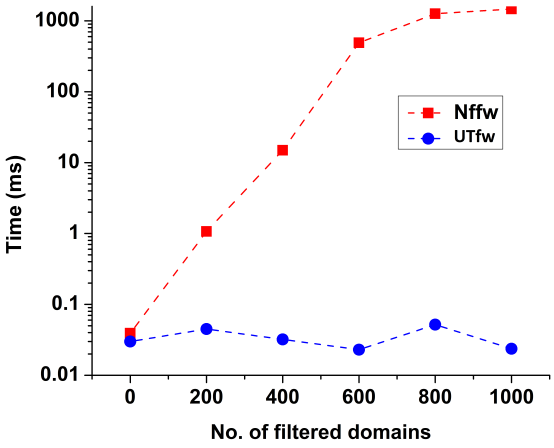

Fig. 6. Packet processing time (log scale): With 10k flows through firewall.

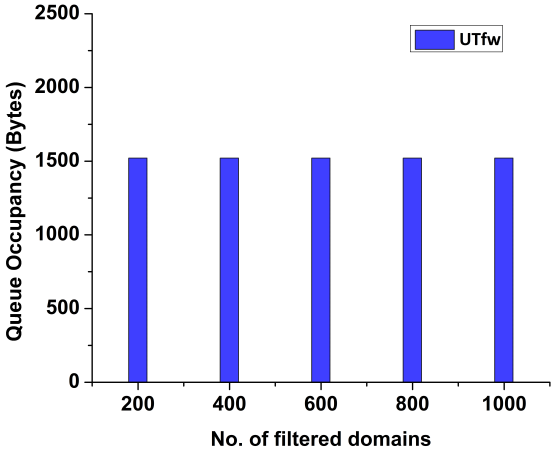

Fig. 7. Queue occupancy: Under heavy cross-traffic (*i.e.* 10k flows), increasing firewall rules do not impact queue occupancy.

processing time of Nffw increases steadily, but P4Wall has nearly constant processing time ($.02 - .04$ ms) as expected from a TCAM implementation.

To further assess the impact of cross-traffic, we generated 10k parallel web connections through the P4 switch and Nffw separately. As Figure 6 shows, even with 1000 filtering rules P4Wall does not have an appreciable impact on the packet-processing time of the switch. This graph is semi-logarithmic: with traffic filtering rules for 1000 domains, the average packet processing time is ≈ 1.4 sec for Nffw, and ≈ 0.02 ms with P4Wall (*i.e.* a speedup of 7×10^4 times). We conclude that with a large number of firewall rules, Nffw performs poorly compared to P4Wall. Further, the difference increases under heavy cross-traffic.

We note that our experiment focuses on the average-case performance. Is it possible that a few worst-case packets get arbitrarily delayed, or perhaps a queue in the switch is slowly filling up (so performance would degrade after a few hours or days)? To answer this question, we check the queue occupancy inside the switch. As Figure 7 shows, there is no such backlog

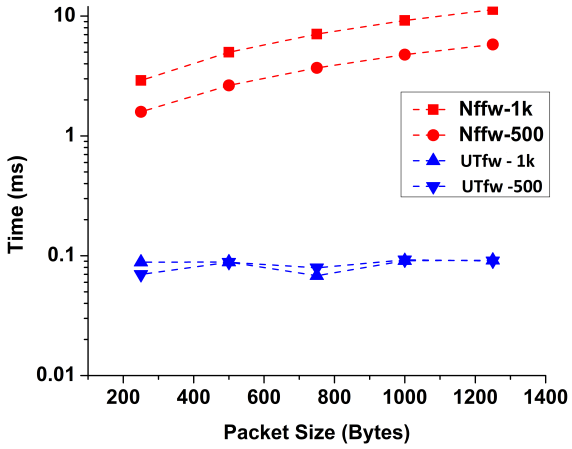

Fig. 8. Impact of packet size on packet processing time.

of packets accumulating within the switch, even with 10k parallel flows, and even with large filter lists (1000 URLs).

2) Impact of packet size: Different applications generate packets of different sizes. Could applications generating larger packets cause congestion at the P4 switch running P4Wall? To answer this concern, we generated traffic with varying packet sizes (with random bytes inserted), and recorded the packet processing time with the firewalls (Nffw and P4Wall) configured to filter 500 domains, and again for 1000 domains.

Figure 8 illustrates that for both 500 and 1000 domains, P4Wall not only consistently outperforms Nffw, it also does not lose performance with increasing packet size.

3) Throughput: Our final metric of interest is *throughput*, which determines the rate at which a user can access bulk content (streaming, downloads). We use `iperf` to measure throughput and how it varies as we increase the number of filtered domains (and thus firewall rules).

As Figure 9 shows, adding firewall rules adversely impacts Nffw, but P4Wall shows no measurable impact. With no filtering rules, `iperf` reports nearly 10 Gbps throughput for both Nffw and P4Wall; with rules for 100 domains, Nffw reduces the throughput by $100\times$ while P4Wall shows no decrease at all.

Overall, our experimental evaluation confirms that P4Wall not only outperforms a netfilter firewall by orders of magnitude, we also see the difference steadily increases as we increase the test load.

VII. Discussion and Future Work

In this section, we discuss a few points of interest regarding P4Wall including analysis, possible significance, and some issues (that may be addressed in future work).

A. P4Wall runs on a switch with very limited computational power. Will it work at Enterprise or Internet scale?

P4Wall is a proof-of-concept system, and its primary purpose is to show that DPI is *possible* using P4-programmable

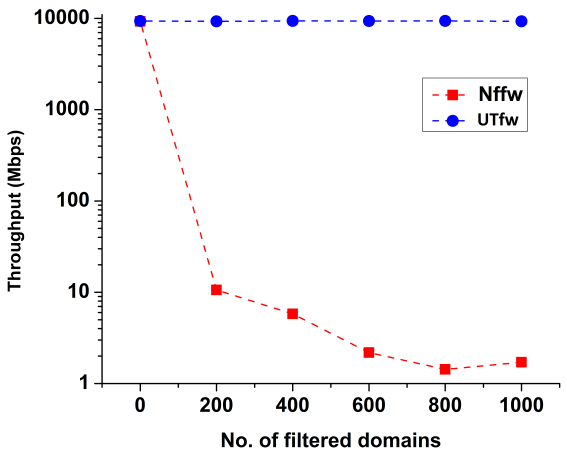

Fig. 9. Throughput (log scale): Impact of increasing firewall rules.

switches. At the same time, we note that even as a proof of concept, it clearly works at a non-trivial scale.

Our results in Section VI demonstrate that a switch can filter substantial traffic without any increase in latency or packet loss. The limiting factor is TCAM memory: firewall rules could fill up the memory and leave the switch unable to perform other functions such as packet routing. However, we note that in practice, we were able to create filtering rules for Alexa top-1k domains using a standard cheap switch (Netberg 710), and even then had $\approx 20\%$ of memory left over for additional functions. Hence, while URL filtering is of the order of 100x less efficient than simple IP blocking (we could block $\approx 147.5k$ IPv4 addresses using the same TCAM memory as $1k$ URLs), it is clearly possible to implement P4Wall with a decent-sized blacklist even on modest hardware.

An actual ISP or enterprise admin would certainly use multiple switches to cover required SDN applications, such as load balancing, and would likely spread the blacklist across multiple switches as well. When P4Wall is run on a single smart switch by a small user – say, as the "gateway" in a campus network, providing Network Address Translation (NAT), IP-based Access Control (ACL), and URL filtering – most likely the blacklist would also be small, and there would be enough capacity to carry out these functions.

B. P4Wall can only perform specific cases of DPI. In future, will it handle encrypted traffic, as proposed by secure modern protocols?

Like most firewalls and IDS, P4Wall *cannot* by itself filter encrypted traffic. This power is only enjoyed by bump-in-the-wire firewalls, which (i) require all users of the network to install a new Certificate Authority so their TLS sessions can be compromised, and (ii) can thus perform Man-in-the-Middle attacks on encrypted connections to inspect traffic. A standard programmable switch is not able to perform encryption/decryption without additional logic, so P4Wall is limited to simple DPI tasks that do not involve decryption.

However, even with this limited power P4Wall is quite effective. Besides providing data security for any websites that use HTTP without TLS, in case of HTTPS, the URL is revealed by the Server Name Indication (SNI) in a TLS client Hello message. TLS 1.3 can also encrypt the Client Hello message, including an encrypted SNI; however, in our study in Section V, we did not see a single use of this feature, and indeed found that modern browsers still use TLS 1.2 (at least for our sample, *i.e.* Alexa Top 10k websites).

The case of DNS is similar. P4Wall cannot handle encrypted DNS packets, as in the DNS-over-TLS (DoT) or DNS-over-HTTPS (DoH) protocols [22]. But the percentage of DNS packets making use of such encryption is quite negligible: Kim et al [23] report a figure of 0.09%. It will take considerable time for DNS-over-HTTPS to become popular enough to threaten firewalls.

We intend to explore hybrid solutions (*i.e.* firewalls where P4Wall is paired with a middlebox, specifically to handle encrypted traffic), in future work.

C. How robust is P4Wall to adversarial traffic? Can it handle short or fragmented packets?

While P4Wall is a proof-of-concept, it is surprisingly robust to adversarial traffic.

First, we mention small packets. Legitimate small packets were never an issue in our tests with HTTP(S), but some DNS queries were indeed too short (and were skipped by the parser); we addressed this problem by focusing on DNS responses rather than queries. In future we may divide the packets into groups by length, and send them to different switches (with different filter settings) for P4Wall inspection.

Fragmentation is a more challenging problem. When a URL is split across multiple packets, it does not show up as a single string in any one packet and the firewall can be bypassed. We currently defeat this attack simply by dropping such fragmented packets; honest HTTP GET, TLS ClientHello, or DNS responses are rarely large enough to be spread over multiple packets. A more nuanced approach would be to delay them, using the P4 technique of *recirculation* (looping from egress back to ingress) until all fragments are received, then reassemble the packet and *then* match URL. This is a direction of planned future work.

D. Why is P4Wall tested against netfilter, rather than a Next-Generation Firewall device (Fortigate etc.*)?*

P4Wall is currently a proof of concept. Our current contribution is not focused on its performance – it is that we demonstrate how (limited) DPI *can* be performed in the data plane, using only standard P4, *even though P4 was not designed for such use [24].*

In our tests, we compare our first implementation of P4Wall against netfilter as it is a standard (software) firewall [25], [26]. P4Wall wins the comparison, and demonstrates that it is a competitive option within the scope of our problem. However,

it does not yet support advanced features like Keyword filtering, DoS or scan detection, *etc.* (which enterprise firewalls do support). In future work, we will explore how P4Wall can be extended to support such features, and provide a meaningful performance comparison against true Internet-scale firewalls.

VIII. Related Work

SDN switches with a programmable data plane have been used in a wide range of network functions such as load balancing [27]–[29], telemetry [30], and for offloading tasks from servers [31]. More recently, they have also made a substantial impact in network security tasks [3], [32]–[41], in particular detecting and protecting against attacks such as port scans and distributed denial-of-service attacks. However, these contributions focus on manipulating flow-level information from packet headers – *i.e.*, no Deep Packet Inspection[8].

This is even more true for the body of work that makes use of P4-compatible switches as stateful or stateless firewalls in the data plane [42]–[46], such as in particular, P4Guard [47], and Gallium [48]. These works build on the tradition of using SDN switches [49] and even plain switches/routers as network-layer firewalls [50], and utilize header information from Layers 2, 3 and 4. They do not touch the TCP or UDP payload, and therefore, cannot perform application-layer firewalling or Deep Packet Inspection. We consider these approaches to be complementary to our work.

We now go on to consider the most directly-related papers, studying Deep Packet Inspection in the data plane.

Meta4 [23], one early example of DPI in the programmable data plane, captures packets stats per domain name. It has a very limited domain-parsing ability (four domain name labels), works only for DNS packets, and makes use of packet re-cirulation to update statistics in registers. Even so, this approach may be useful for specific use cases such as IoT device fingerprinting, DNS tunnel detection, and DNS based denial-of-service attacks.

A more directly-related approach comes from Jepsen et al. [51], who recirculate packets to parse out keywords in the payload. While we found their use of Deterministic Finite Automata (DFA) on a P4-capable switch to be very interesting, and are ourselves investigating this approach, the fact that *the packet is consumed while searching for strings* makes the approach useless for a firewall. We are actively trying to see if this shortcoming can be overcome and used for Deep Packet Inspection for network security.

DeepMatch [11] is perhaps the closest match to our own work: it successfully performs Deep Packet Inspection (DPI) on packet payloads. However, DeepMatch is developed in Micro-C, and targets the Netronome NFP-6000 SmartNIC – *i.e.*, custom logic integrated in the switch. In comparison, we target a standard platform.

The other closely-related work we are aware of, P4DNS [52], extracts the domain name from a DNS query packet and builds a DNS response packet using the match-action table as a lookup table. Their solution was DNS-specific only parsed very limited-length domain names, but it remains a potential approach we may use for DNS-specific security.

Finally, in our own demo paper [redacted for review], we first mention that actual traffic may have predictable URL positioning in the packet. These early observations laid the foundation of our systematic field study (in Section IV). In this paper, we build on this work, studying the constraints required for a solution and demonstrating an actual implementation of a DPI-capable firewall in the data plane.

In this paper, we demonstrate how the practical predictability of network traffic makes it possible to perform DPI in the programmable data plane. Our system P4Wall can handle multiple protocols, with excellent performance for its (limited) DPI task (filtering of blocklisted URL). We trust our implementation will draw attention to the fact that P4 can provide more than just header-field filtering. It also opens the question of how we might build and scale a proper application-layer firewall in the data plane, which we will explore in future work.

IX. Conclusion

In this paper, we demonstrate P4Wall, a URL-filter in the dataplane. Our primary contribution is to show that simple P4 switches are powerful enough to perform limited DPI, without custom logic (`extern` hardware implemented using NetFPGA *etc.*), or external help from firewalls. Secondly, P4Wall performs simple Deep Packet Inspection (*i.e.* URL detection) for multiple application-layer protocols, scales to a substantial number of domains (1000), and outperforms a standard software firewall by three to four orders of magnitude. So even though P4Wall is a proof-of-concept, it could be immediately deployed *e.g.* by small network admins to blacklist malware domains. Further, as our implementation is compatible with any standard P4-compatible SDN switch, it is easily within reach for *e.g.* campus networks who cannot afford enterprise solutions. Finally, as P4Wall uses a simple algorithm, we anticipate it can easily be updated in response to change (for example, if browsers change the structure of their HTTP requests). We intend to study methods to scale-up the power of P4Wall so it can handle larger lists and more robust methods, in our future work.

[8] A scan is indicated by many flows in quick succession with the same source IP but different destination IP. For a DDoS attack, there are many source IPs and one destination IP.

References

[1] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE security & privacy*, vol. 12, no. 4, pp. 63–67, 2014.

[2] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, "P4 edge node enabling stateful traffic engineering and cyber security," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 11, no. 1, pp. A84–A95, 2019.

[3] D. Ding, M. Savi, F. Pederzolli, M. Campanella, and D. Siracusa, "In-network volumetric ddos victim identification using programmable commodity switches," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1191–1202, 2021.

[4] "Cisco firepower threat defense," https://www.cisco.com/c/en/us/support/\\docs/security/asa-5500-x-series-firewalls/212420-configure-firepower-threat-defense-ftd.html.

[5] "Sonicwall supermassive series," https://www.sonicwall.com/\\medialibrary/en/datasheet/datasheet-sonicwall-supermassive-series.pdf.

[6] "Fortinet fortigate series," https://www.fortinet.com/products/next-generation-firewall/mid-range, Accessed: 2022-05-25.

[7] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 323–336. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[9] M. Budiu and C. Dodd, "The p416 programming language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 5–14, sep 2017. [Online]. Available: https://doi.org/10.1145/3139645.3139648

[10] "Nvidia bluefield data processing units," https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[11] 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*, 2020, pp. 336–350.

[12] "Netberg Aurora 710 Intel Tofino Switch," https://netbergtw.com/products/aurora-710/.

[13] "Intel® Tofino™," https://www.intel.com/content/www/us/en/products/\\network-io/programmable-ethernet-switch/tofino-series.html, Accessed: 2022-05-25.

[14] "Network programming language," https://nplang.org/npl/explore/, Accessed: 2022-05-25.

[15] P. W. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. E. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, pp. 87–95, 2014.

[16] "Portable switch architecture (working draft)," https://p4.org/p4-spec/docs/PSA.pdf.

[17] "Open Tofino," https://github.com/barefootnetworks/Open-Tofino, Accessed: 2022-05-25.

[18] "Statcounter," https://gs.statcounter.com/.

[19] "Nginx web server," https://www.nginx.com/.

[20] "dnsmasq: Dns server," https://thekelleys.org.uk/dnsmasq/doc.html.

[21] "iperf: Network measurement tool," https://iperf.fr/.

[22] T. V. Doan, I. Tsareva, and V. Bajpai, "Measuring dns over tls from the edge: Adoption, reliability, and response times," in *International Conference on Passive and Active Network Measurement*. Springer, 2021, pp. 192–209.

[23] J. Kim, H. Kim, and J. Rexford, "Analyzing traffic by domain name in the data plane," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2021, pp. 1–12.

[24] "The p4-16 language specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[25] M. Mihalos, S. Nalmpantis, and K. Ovaliadis, "Design and implementation of firewall security policies using linux iptables." *Journal of Engineering Science & Technology Review*, vol. 12, no. 1, 2019.

[26] A. Kak, "Computer and network security lecture notes," https://engineering.purdue.edu/kak/compsec/.

[27] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2016, pp. 1–12.

[28] E. Cidon, S. Choi, S. Katti, and N. McKeown, "Appswitch: Application-layer load balancing within a software switch," in *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017, pp. 64–70.

[29] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 125–139.

[30] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, vol. 15, 2015.

[31] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 19–27.

[32] G. Grigoryan and Y. Liu, "Lamp: Prompt layer 7 attack mitigation with programmable data planes," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–4.

[33] M. Kuka, K. Vojanec, J. Kučera, and P. Benáček, "Accelerated ddos attacks mitigation using programmable data plane," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–3.

[34] F. Paolucci, F. Cugini, and P. Castoldi, "P4-based multi-layer traffic engineering encompassing cyber security," in *Optical Fiber Communication Conference*. Optical Society of America, 2018, pp. M4A–5.

[35] Y. Mi and A. Wang, "Ml-pushback: Machine learning based pushback defense against ddos," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 80–81.

[36] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with sdn data plane," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[37] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore, "Machine-learning-assisted ddos attack detection with p4 language," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–6.

[38] X. Z. Khooi, L. Csikor, D. M. Divakaran, and M. S. Kang, "Dida: Distributed in-network defense architecture against amplified reflection ddos attacks," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2020, pp. 277–281.

[39] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev, "{NetHide}: Secure and practical network topology obfuscation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 693–709.

[40] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[41] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 33–38.

[42] J. Cao, J. Bi, Y. Zhou, and C. Zhang, "Cofilter: A high-performance switch-assisted stateful packet filter," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 2018, pp. 9–11.

[43] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable {In-Network} security for context-aware {BYOD} policies," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 595–612.

[44] R. Ricart-Sanchez, P. Malagon, J. M. Alcaraz-Calero, and Q. Wang, "Hardware-accelerated firewall for 5g mobile networks," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 446–447.

[45] ——, "Netfpga-based firewall solution for 5g multi-tenant architectures," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 132–136.

[46] P. Vörös and A. Kiss, "Security middleware programming using p4," in *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 2016, pp. 277–287.

[47] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4guard: Designing p4 based firewall," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–6.

[48] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 283–295.

[49] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao, "Towards a reliable {SDN} firewall," in *Open Networking Summit 2014 (ONS 2014)*, 2014.

[50] A. X. Liu, *Firewall design and analysis*. World Scientific, 2010, vol. 4.

[51] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé, "Fast string searching on pisa," in *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019, pp. 21–28.

[52] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4dns: In-network dns," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6.