DeeP4R: Deep Packet Inspection in P4 using Packet Recirculation

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*—Software-defined networks are useful for multiple tasks, including firewalling, telemetry, and flow analysis. In particular, the P4 language makes it possible to carry out some simple packet processing tasks in the data plane, *i.e.,* on the switch itself (without real-time support from the SDN controller or a server). However, owing to the limitations of packet parsing in P4, these tasks involve only the packet headers. In this paper, we present a novel approach that allows Deep Packet Inspection (DPI) – *i.e.,* inspection of the packet payload – in the data plane, using P4 alone. We make use of the fact that in P4, a switch can clone and recirculate packets. One copy (clone) can be recirculated, slicing off a byte in each round, and using a finite-state machine to check if a target string has yet been seen. If the target string is found, the other copy (original packet) is discarded; if not, it is passed through. Our approach allows us to build the first application-layer firewall (URL filter) in the data plane, and to achieve essentially line-rate performance while filtering thousands of URLs, on a commodity programmable switch. It may in future also be used for other DPI tasks.**

***Index Terms*—Software-Defined Networks (SDN), Programmable Dataplane, Application-Layer Firewall**

I. Introduction

Switches and routers – particularly Software-Defined Network (SDN) switches – have been successfully used to implement network-layer firewalls [1], flow analysis [2], and a wide range of other functions. Part of the reason for this remarkable versatility is that a small number of packet headers (source IP, source port, destination IP, destination port, protocol, *etc.*) are key for a variety of networking tasks. However, more advanced techniques, such as the detection of malicious traffic or malware signatures, require *Deep Packet Inspection* (DPI), *i.e.,* the inspection of packet payloads and not just packet headers. For example, a Network Intrusion Detection System (NIDS) needs DPI to identify if a packet carries the signature of an attack such as Heartbleed [3].

The current state-of-the-art in DPI is still provided by old-school dedicated middleboxes, such as Cisco Firepower Threat Defense [4], SonicWALL TZ/NSA/SuperMassive Series [5], Fortinet FortiGate [6] *etc.* These solutions treat the network administrator as a *consumer* – the admin has no option other than to trust the manufacturer for strong security guarantees (*i.e.,* that the firewall is not itself malicious [7], does not violate user privacy, *etc.*). Further, such middleboxes are usually on-path rather than in-path [8], and may only inspect a sample of traffic so as not to become a bottleneck. A comprehensive line-rate filtering solution is very expensive, and even modest firewalls may be out of the reach of small businesses. Such lack of access was one of the original motivations for developing Software-Defined Networks [9]. It is, therefore, natural to ask why DPI tasks, such as URL filtering, are not performed using programmable switches, which are friendly to the network administrators and usually provide high performance for data-plane tasks.

We find that DPI-in-SDN is challenging because, in general, the payload is large and unpredictable in structure compared to packet headers. For example, one payload item – the HTTP application-layer header – has 47 possible fields, and these fields can occur out-of-order, have variable lengths, or can be entirely missing. Switches are designed for high-performance packet forwarding [10] and not for general computation, so even in the case of P4 – a language that allows users to freely define headers for their own protocols (which the switch then parses as easily as TCP or IP headers) – the authors of the $P4_{16}$ standard explicitly say that P4 is not intended for DPI [11].

However, these challenges are not impossible to overcome. Early attempts such as Sekar et al's CoMb consolidated middlebox [12] built on the Click modular router [13], show that DPI is indeed possible if we are willing to invest in a specialized switch. Indeed, specific cases of limited DPI have been built using P4-programmable platforms [14], [15]. Such solutions are a step in the right direction but are partial *i.e.,* the switch leaves a portion of the work to an external server [16], works only for special cases where the traffic satisfies strong conditions [17], or requires specialized hardware, and are thus implemented in NetFPGA (not on a standard switch). We conclude that while it is possible to build a proof-of-concept DPI system in a software-defined network, *there is still a need*

for a practical DPI-capable firewall that uses only standard functionality (runs on unmodified commodity SDN switches).

In this paper, we present such a system – Deep Packet Inspection in P4 using packet recirculation (*DeeP4R*) – that performs Deep Packet Inspection using only standard SDN switches and the P4 dataplane programming language. Our contributions are as follows.

- *DeeP4R* is the first firewall to achieve "true Deep Packet Inspection in P4" (which we define as, DPI without real-time help from a controller or external firewall), using only standard P4-compatible switches. When a packet arrives, we use P4 functions to clone it, then apply the recirculate-and-truncate method of pattern matching [18] on the cloned packet. (We loop the packet through the switch, consuming one byte from it with each pass. A Deterministic Finite Automaton keeps track if we have seen the target string.)
  If the clone is consumed without us seeing the target string (URL), we let the original packet (which has not been altered) pass through; otherwise, we drop it. Our novel method of combining packet cloning with recirculate-and-truncate allows us to perform flexible parsing in P4 *and* allow non-target traffic to pass through transparently.
  We do not claim that *DeeP4R* can handle *all* Deep Packet Inspection tasks, but it is perfectly capable of application-layer firewall tasks such as URL filtering. To our knowledge, *DeeP4R* is the first filter able to block URLs directly in the data plane (not taking real-time help from SDN controller, firewall, or special hardware). In future we will extend our approach to match other strings such as keywords, and to other protocols such as DNS.
- We implement, demonstrate, and benchmark the *scalability* and *performance* of *DeeP4R*, which as a dataplane program can process traffic very efficiently on a real switch. For instance, with 5000 domain names to filter and 10000 parallel flows, the latency on *DeeP4R* on a commodity SDN switch is under 1 ms while our firewall server (running standard Linux `netfilter` firewall) takes over 5 sec.
  Our implementation is developed and run on the Netberg 710 switch [19], built around the Intel P4-based Tofino ASIC [20]. This is a commodity switch with a standard architecture (market cost roughly $5000); further, our code can be ported very simply to another platform such as the Broadcom NPL ASIC [21], as we *only use standard P4 functionality* to parse packets, extract fields, and match values to actions. Our P4 code (for the Tofino switch) and all related scripts *etc.* are all available for future study or extension, at our repository [22].

We now present some background in Section II and our design in Section III, before going on to our evaluation results in Section IV. We then discuss limitations and future work in Section V, mention more related work in Section VI, and end with some concluding remarks.

II. Background

SDN switches like our Netberg Aurora 710 allow the user (network admin) to specify how the switch should process network packets, using a standard language (P4). In brief, a P4 program specifies the schema of packet headers for any desired protocols (the headers can be whatever the user chooses, so long as it is a consistent chunk of bytes at a consistent position in the packet). Once the switch has this schema, it is able to extract these header fields from packets, and use them for routing, load balancing *etc.* Thus it becomes simple for the user to adapt the switch for new protocols, such as MPLS or QUIC [23]. Such a switch is said to have a *programmable dataplane*. We explain the necessary components below.

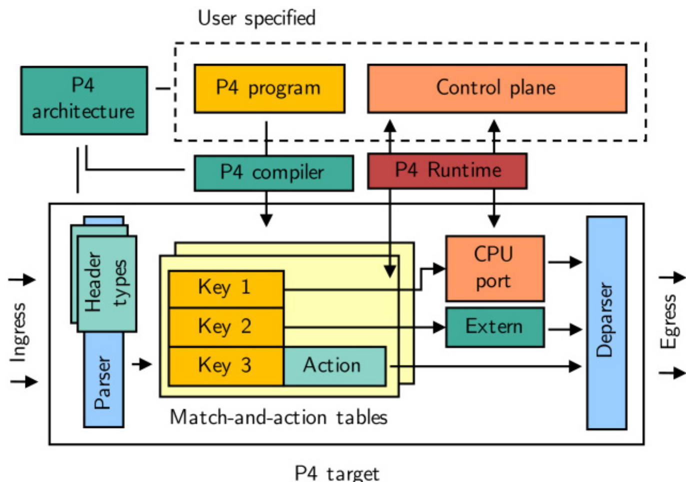

Fig. 1: Pipeline for a switch supporting P4 language, from P4-MACSec [24].

- *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). There may be packets of various protocols in the same traffic; as the parser 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 allow for 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.
- *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*[1] – 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.

[1]*Metadata*, in a programmable switch, refers to special data structures where a user program can store information generated during packet processing. Metadata can be *intrinsic* to the switch or specified by the dataplane program. Some fields such as timestamp, are read-only, others such as egress_port can be modified (*e.g.,* to control where packet should be output).

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.* 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.

Actual programmable switches may provide more facilities.
- *Traffic Manager.* In between ingress and egress blocks of a switch, there may be a traffic manager – which is *not* programmable using the P4 language, and also helps decide which egress port to forward the packet to.
- *Multiple parser-deparser.* In many architectures such as PSA [25] each pipeline (ingress or egress) has its own parser, control block, and deparser. This allows for a second pass over the packet, without needing to loop the packet itself through the entire switch.
- *User defined control blocks.* It is possible to add modular functionality to the data plane, through *user defined* control blocks, which may be incorporated into either the ingress or the egress control block.
  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 Native Architecture (TNA) [26], and so typically will not have such custom logic.

We note that *DeeP4R* performs deep packet inspection using only standard P4. In other words, we do not use any of these advanced features (second parser/deparser, traffic control block, custom logic *e.g.,* 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. System Design

The *DeeP4R* system implements a Deterministic Finite-State Automaton (DFA) in the switch, to match target strings (keywords, URLs) in the packet.[2] For example, Figure 2 shows a DFA to match the target URLs `evil.com` and `bad.com`.

The DFA makes transitions on characters as we traverse the packet extracting them one by one (the method of recirculate-and-truncate [18]). The state of the DFA allows us to determine whether the target string has been seen, so we can match

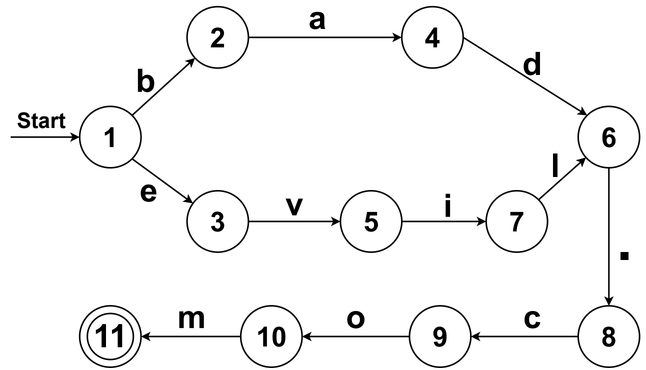

Fig. 2: DFA to match evil.com and bad.com.

URLs (and potentially other strings or keywords) found at any position in the packet. But the method is *destructive*, as it consumes the packet being matched.[3] To use recirculate-and-truncate in a firewall, we *clone* the packet. One copy can be used up for matching, and the other is accepted or discarded depending on whether the keyword (URL) was found. We now present the details of this system design.

A. *Architecture of* DeeP4R.

DeeP4R includes two main methods.
1) *Recirculation-and-truncation* [18] involves looping the packet from egress to ingress (or, in case of a real switch, from the ingress-section deparser back to the ingress-section parser), so it passes through the pipeline again – effectively forming a loop. With every pass-through, the packet is edited, removing the first byte and checking its value to make transitions according to the DFA.
2) *Duplication* is our method to keep the original packet intact through the matching process. The packet is cloned, and the clone is consumed byte-by-byte in the DFA matching process, while the original remains intact until it is either dropped or allowed to pass.

For these functions, we both need to implement a DFA, and to keep track of the correspondence between original and cloned packets. (We also need to clean up the state after the packet processing is over.) Accordingly, we build the *DeeP4R* system with the following components.
- `label` *header.* This is a small custom header which we insert into both original and clone packets just after the TCP header. It is for use as a scratch pad, and if the packet is forwarded by the switch, this header is deleted.
  The *DeeP4R* Finite State machine tracks state using a field of the "label" header in the clone packet. We also keep track of which clone packet (and which `decision`) corresponds to which original packet, by sharing the same `unique_ID` in the "label" header.
- `decision` *register.* The other component of state tracked in *DeeP4R* is the decision for a packet. This is held

[2]Note that our DFA state is separate from the parser state or the flow state (OpenState [27], for example, uses a DFA that makes one transition per packet to track flow state) – our state machine attempts to match a string in the packet, by making one transition per byte.

[3]The method was originally meant for applications where the programmable switch is the final handler of the packet – for example, in a data center fabric that directly processes a query carried in the packet.

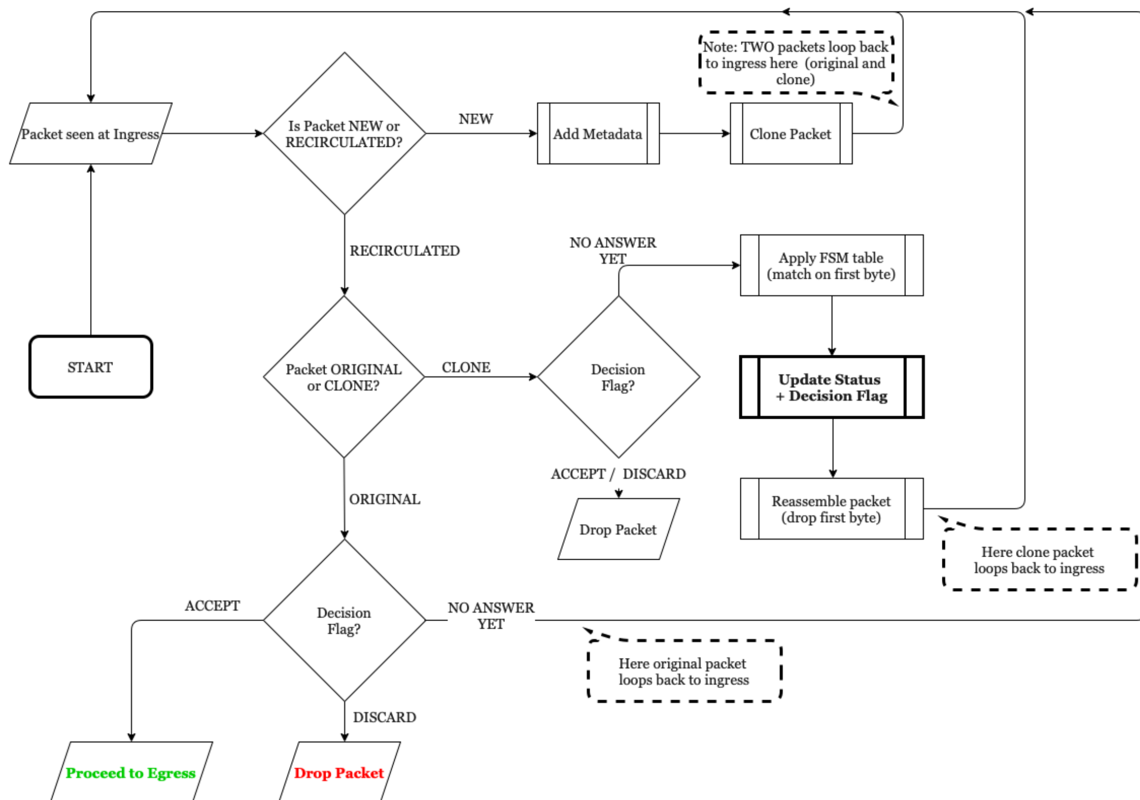

Fig. 3: Life cycle of a packet processed in *DeeP4R*. The *supervisor* table actions correspond to the decision blocks, and the *DFA* table corresponds to the bolded block "update status".

in a "register" (which in P4 language, simply means an associative array *i.e.,* a hash table in the switch SRAM memory), named `decision`.
`decision` stores the decisions for packets currently being processed in the switch. If a packet with `unique_identifier = X` is to be discarded, then `decision[X]` is 1. If it is to be accepted, then `decision[X]` is -1. And if no decision is available, then `decision[X]` is 0.

- *DFA transition table.* DFA State is updated using a standard match-action table in the ingress block. The current DFA state is stored in the label header of the clone packet; we use this state, and the first byte of the packet payload (`slice`), as the lookup key for the match-action table. The action looked up in the table, sets the new DFA state, and can also write to `decision` when a string is successfully matched. Table I is an example of a simple DFA transition table.
- *Supervisor table.* The supervisor match-action table is responsible for packet classification – *i.e.,* treating packets differently based on whether they are new or recirculated, original or cloned packets, and whether the `decision` is set or not. It is very small compared to the DFA table, but it may be considered the "main() function" of the system. Its actions form the high-level processing flow of Figure 3.

state	slice	action
1	b	update_state(2)
2	a	update_state(4)
4	d	update_state(6)
1	e	update_state(3)
3	v	update_state(5)
5	i	update_state(7)
7	l	update_state(6)
6	.	update_state(8)
8	c	update_state(9)
9	o	update_state(10)
10	m	update_state(11, **1**)

TABLE I: Our example DFA, expressed in match-action table rules. 1 is the start state. 11 is the only accept state *i.e.,* it indicates that the URL was seen. Note that the last transition from 10 to 11 not only updates the state in the label header, it also writes to `decision` – hence the 1 in bold.

B. Processing of a Packet.

1) When a new packet enters the ingress block the first time, a new header (`label`) is inserted between the TCP and application-layer header; next, the whole packet is cloned. The label header contains the following information: (a) whether the packet is original or cloned; (b) the current DFA state information (only present in cloned packet, absent in original); (c) a `unique_id` that uniquely identifies a packet pair (an original packet and its clone).

After the packet is cloned, both original and cloned packets are forwarded to the recirculation port at egress. From this point, the packets are treated differently,

2) The clone packet, when it arrives at ingress, is sent to the DFA match-action table. The combination of the current state and `slice` (first byte of payload) – say, (4, d) – is used as a key to look up the appropriate action in the match-action table. (`slice` will later be dropped by the deparser, just before the packet is recirculated.)
   The transition table either calls the action `update_state` (if the table has an action matching the lookup key) or `reset_state` (if it does not).w
   - `update_state` updates the state variable to the new state, and if it reaches an accept state, sets the flag `decision[unique_id]` to 1 (indicating the original packet should be dropped) *e.g.,* in Table I, this happens when the state reaches 11.
     On the next recirculation, if the supervisor table sees that the flag `decision[unique_id]` is set, it simply drops the clone packet (its job is over).
   - `reset_state` is the default action, and restarts the evaluation after a false start, *i.e.,* if an unexpected character appears. *e.g.,* if 'e' appears after state 4 in Table I, *DeeP4R* starts again with the next byte, from state 1.)
3) What if the packet terminates, and no blocked URL has been matched in the entire TCP payload? P4 catches this case (we fail when trying to extract a slice, and the inbuilt construct *validity bit* returns 0). Instead of going to the DFA match-action table, we fall through to the supervisor table, which carries out the actions (a) set `decision[unique_id]` to -1 (indicating the original packet should be passed), and (b) drop the current (clone) packet, its job is over.
   Otherwise, the supervisor table recirculates the packet, so the cycle continues with the next byte.
4) When an "original" flagged packet appears at ingress, the supervisor table simply checks the flag corresponding to its id, `decision[unique_id]`. If it is still 0, it is recirculated to wait until a decision becomes available.
   When the decision is made (packet dropped or forwarded), the supervisor carries it out on the packet and clears the `decision[unique_id]` to 0 to avoid possible unique_id collision with future packets.

IV. Evaluation

Our *DeeP4R* system design, from the previous section, raises some important questions which need to be answered by experimental evaluation.

- Does the finite-state machine required to filter a substantial number of URLs (say 1000 - 2000 URLs for a real firewall [28]), fit in the memory available for flow tables in a commodity switch? Is there enough room for other match-action tables (*e.g.,* for routing functions)?
- *DeeP4R* matches patterns by recirculating packets. This inevitably introduces some delay in packet processing, *even for packets that are finally accepted*. How large is this delay, and does it cause an unacceptable penalty in performance? For instance, do TCP packets time out because of the latency introduced by *DeeP4R*?
- Besides latency, network performance also depends on throughput. How does *DeeP4R* compare to the baseline performance of the switch, and to a traditional firewall? How well can it handle multiple flows and network congestion?

We now present the experimental setup for our evaluation, followed by our tests and observed results.

A. Experimental Setup.

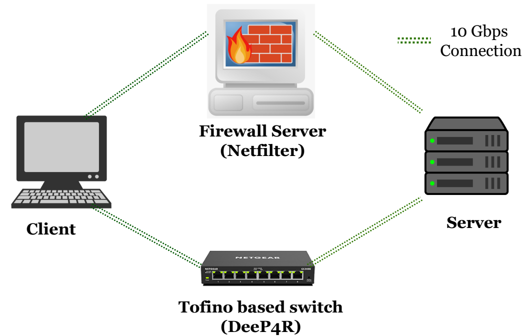

Fig. 4: Experimental setup: Client machine fetches HTTP or HTTPS traffic (web pages, including large files) from Server. In separate runs, we pass identical traffic through our Tofino-based switch (running *DeeP4R*), and through a server running Netfilter firewall, for a fair comparison.

Our test setup includes the following components.

- **Client host** : An Ubuntu Linux (20.04 LTS) system, that generates requests for traffic. This can include high or low volume traffic from `wget` or `iperf` as well as tailored TCP packets (to control the packet length).
- **Server host** : An Ubuntu Linux (20.04 LTS) system set up to respond to requests from client. The server runs `nginx` and `iperf` in server mode.
  Both client and server have 10 Gbps Ethernet connections (limited by NIC capacity).
- **Management host** : The management host converts high-level filtering policies (*i.e.,* the list of URLs to block) into a DFA, which it outputs in intermediate (barefoot runtime) Python API commands. The SDN controller runs these commands to set up the switch match-action tables (DFA and supervisor).
  This host is not shown in Figure 4, as it is not an operational part of *DeeP4R*. It is only used for a one-time setup of the switch. In theory the management functions could be run in the linux computer embedded in the switch, but we used a separate desktop machine for better performance and to ensure we have enough memory for DFA construction.
- **P4-compatible SDN switch** : We use a commodity smart switch, the Netberg Aurora 710, built around a Tofino ASIC. Our switch runs the Open Network Linux (ONL) Operating System. The ASIC's ethernet ports (dev ports) are directly connected to the physical switch QSFP connectors.

P4 studio is used to develop both the data plane P4 program (*i.e.,* the match-action rules to implement a DFA on the switch), as well as the control-plane barefoot runtime python scripts (used to install the DFA, and also for other tasks like checking the dataplane traffic statistics, as reported below).

- **Controller** : In our experiments, the SDN controller is physically located on the Netberg switch itself.
  We note that our build process (with P4 studio) outputs the connectors to allow for the interaction of control plane with the daemons running on the switch. We can at any time move to a physically separate controller, or have one controller in charge of multiple switches, as is common in SDN. In our case, we find a local setup is adequate for our experiments, so the controller is logically separate but physically run on the same box *i.e.,* the Linux computer embedded in the switch.
- **`netfilter` firewall server** : An Ubuntu 20.04 LTS server, set up to forward packets, using separate NIC and separate physical ports for ingress and egress. We use the standard Linux `netfilter` firewall (kernel process, configured using `iptables` – our filtering rules are installed in the `filter` table and FORWARD chain).

In our experiments, while the switch has QSFP ports (100 GBPS capacity), our client and server NICs are only 10 Gbps, so this is the limiting factor w.r.t. throughput. To make sure the physical connections are not a bottleneck, we used QSFP+ breakout cables for all connections.

B. Experimental Results.

Our first observation is that even the largest DFA we test (5000 URLs), uses a very small fraction of the available switch memory (of the order of 0.02%). Our experiments are limited by the time and memory required for the offline pre-computation of the DFA; once computed, it took little space. A natural follow-up question is, what is the *maximum size* of policy that the system can support? – this depends on the switch used (and its memory capacity), and also on the specific URLs used (hence the size of the DFA). While exact details of the memory layout in Tofino or Tofino 2 are confidential, our current best estimate is that even in the worst case (combinatorial explosion), assuming URLs of length 10, we can support a policy of approximately 70 000 URLs. Our focus in these experiments is to check performance with a policy of reasonable size, but we intend to stress-test the system with maximum-size policies in future work.

We now discuss the performance of *DeeP4R*, as measured in terms of latency for single and multiple flows (experiment 1 and 2), as well as throughput and packets dropped (experiment 3).

Experiment 1. Latency for a Single Flow.

Our first experiment was to measure the average end-to-end latency experienced by network applications, with a single flow passing through *DeeP4R*, and to compare it with the latency of our firewall server (running `netfilter`).

We define *end-to-end application-layer latency* as the time difference between sending the request packet, and receiving the corresponding response packet. It is measured from the timestamps seen in packet captures at the client. We measure how this latency varies as we increase the number of domains filtered by *DeeP4R* as well as `netfilter`. (The domain names used for our test are the top URLs from Alexa. *e.g.,* when testing for 5 URLs we filter for `google.com`, `youtube.com`, `facebook.com`, `baidu.com` and `wikipedia.org`.)

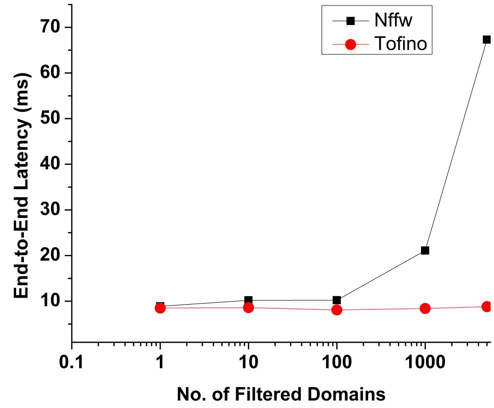

Fig. 5: E2E Delay vs Filtered Domains.

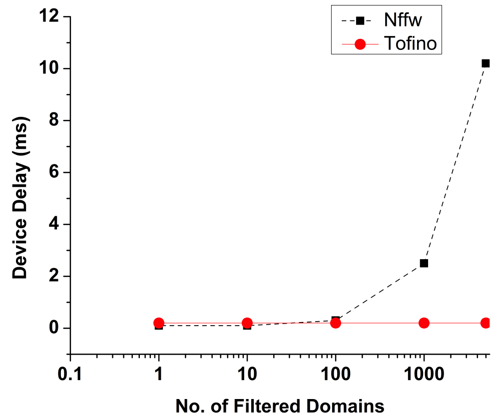

Fig. 6: Device Delay vs Number of Filtered Domains.

As seen in Figure 5, we find that the performance of *DeeP4R* was consistent for all our tested policies (varying the number of filtered URL's from 1 to 5000). The delay was roughly 8.5 ms, which is the same as the delay seen with our firewall server for very small policies; for larger policies the server performance degrades steadily (67.3 ms for 5000 rules). We note that the baseline delay of 8.5 ms includes client, server *etc.* delays, so rather than the absolute values we focus on the fact that *even with thousands of rules,* DeeP4R *adds no more latency than a single-rule server firewall.*

We now have the question of how much of the end-to-end delay was directly caused by *DeeP4R*, or by the `netfilter` firewall. Accordingly, we measured the average *device delay* – the time taken by a packet of interest (*i.e.,*, a packet carrying

TLS ClientHello or HTTP GET request) to pass from ingress port to egress port in the switch or the firewall.

Figure 6 shows our results. *DeeP4R* consistently introduces a delay of 0.2 ms, while we vary the number of domains from 1 to 5000. The server firewall starts with almost the same delay (0.1 ms for 1 or 10 rules), but increases to 2.5 ms at 1000 and 10.2 ms at 5000 rules. This is consistent with our position that, within the limits of noise in measurement, *DeeP4R* with up to thousands of rules adds no more delay than a server firewall with 1–10 rules.

Experiment 2. Latency with Parallel Flows.

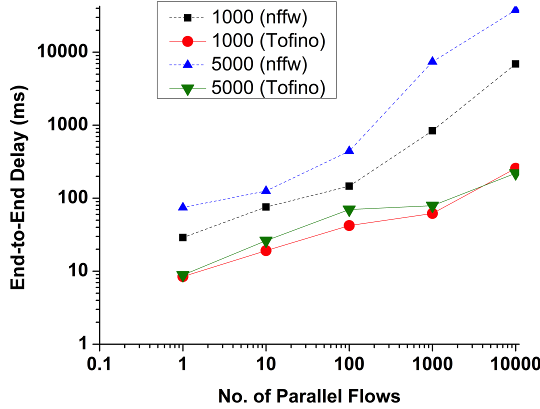

Fig. 7: E2E Delay vs Parallel Flows.

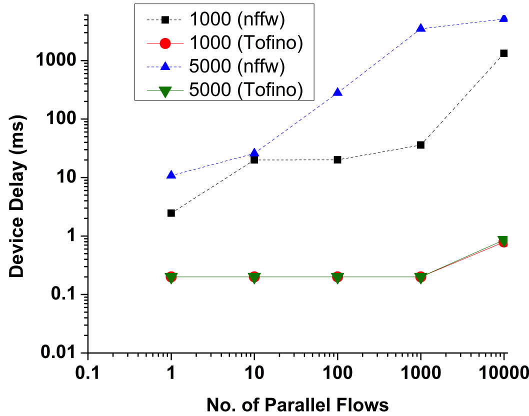

Fig. 8: Device Delay vs Parallel Flows.

In our second experiment, we essentially repeat the measurement of end-to-end application-layer delay and device delay, but at the same time introduce parallel connections to evaluate the impact of cross traffic. The number of parallel connections was varied as 1, 10, 100, 1000 and 10000.

As seen in Figure 7 and 8, end-to-end delay and device delay are both consistently lower for *DeeP4R*.

- With 1000 domain names in the filter, *DeeP4R* end-to-end delay starts at 8.5 ms for one flow (as seen in Experiment 1) and gradually increases to 257 ms for 10k flows. The firewall server starts at 28.9 ms for one flow – already worse than in Experiment 1 – and increases to 6893 ms for 10k flows.
  Of this, in *DeeP4R* the device delay is only 0.2 ms for one flow and rises to 0.8 ms for 10k flows. The firewall server starts at 2.5 ms device delay for one flow and rises to 1329 ms for 10k flows.
- With 5000 domain names in filter, *DeeP4R* shows almost the same performance: 8.8 ms end-to-end delay for one flow, rising to 220 ms for 10k flows. The `netfilter` server degrades sharply, from 74.7 ms for a single flow rising to 37795 ms for 10k flows (almost 40 seconds).
  Of this, in *DeeP4R* the device delay is still 0.2 ms for one flow rising to 0.86 ms for 10k flows. The `netfilter` server starts at 10.8 ms for one flow and rises to 5139 ms for 10k flows – 6000 X slower than *DeeP4R*.

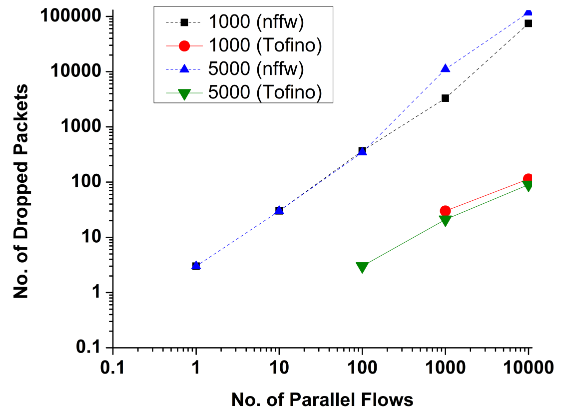

Fig. 9: Dropped Packets vs Parallel Flows.

As an additional experiment, we also observed how many packets were dropped owing to congestion as we increased the load (number of parallel flows). Figure 9 shows that *DeeP4R*, running on a switch which is not worked at full capacity, drops no packets at all until 100 flows and only 114 packets for 10k parallel flows. The firewall server started out with 3 dropped packets for a single flow, and at 10k parallel flows dropped 117733 packets over the duration of the test (38 sec).

Experiment 3. Throughput

For our final experiment, we consider that network performance depends not only on packet latency but also on throughput. Accordingly, we measured the connection's throughput using the standard tool `iperf`, setting it to send traffic on the ports of interest (80, 443).

As Figure 10 shows, *DeeP4R* achieves excellent throughput (about 9.3 Gbps, close to the theoretical value of 10 Gbps) and this does not degrade for our tests with up to 5000 URLs. The `netfilter` server performance degrades much more rapidly.

V. Discussion

In this section, we discuss a few points regarding limitations, further work, and general comments about *DeeP4R*.

Can *DeeP4R* be used with other protocols such as DNS?

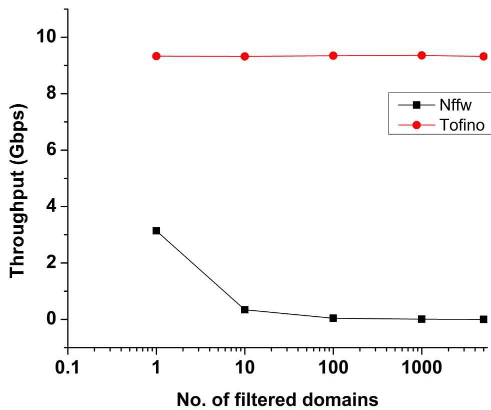

Fig. 10: Impact of increasing firewall rules on Throughput.

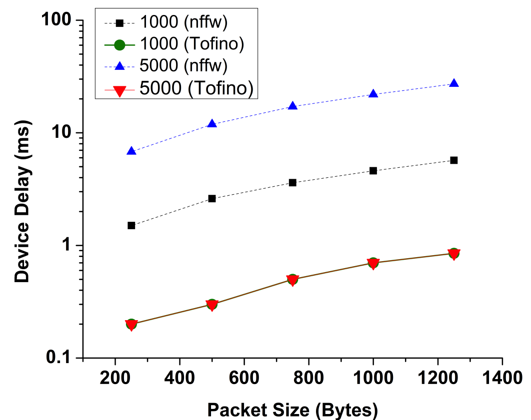

Fig. 11: Device Delay vs Packet Size.

There is no reason the same *approach* would not work with DNS traffic, but one complication is that URL's in DNS are expressed differently – the dot separator between labels is replaced with the length of the label ("www.google.com" becomes "3www6google3com"). We will therefore need multiple DFA's to handle packets of different protocols. We intend to compare this approach to existing DNS-in-P4 solutions such as P4DNS [14], in future work.

We note in passing that while domain names are usually made of ASCII-coded characters, which neatly map into one character per byte, IDN registrations *can* have non-ASCII characters. This is not a problem for *DeeP4R*: it simply handles such cases as requiring two transitions instead of one to match a single character. But this does leave the possibility of *collateral damage*, where (say) a character that is encoded as θ uses two bytes *with the exact same bits* as the ASCII representation of "ab" – so blocking θ.com ends up blocking ab.com as well. This is also an issue we are currently studying.

***DeeP4R* scales well with increasing number of flows and number of filtered domains. What about other factors?**

While it does not commonly vary much, *payload size* in the packet can also affect performance, as a larger number of recirculations will cause the delay to increase.

In practice, for normal sized packets, this effect is small. Studying packets of size varying from 250 to 1250 bytes, we note the *DeeP4R* performance was the exact same for 1000 or for 5000 domains filtered (device delay slowly rising from 0.2 ms to 0.85 ms as packet size increases). Interestingly, netfilter also slowed down – from 1.5 ms to 5.7 ms for 1000 rules, and from 6.8 ms to 27.2 ms for 5000 rules. (The slowdown was slightly less than linear with packet length, for both *DeeP4R* and for netfilter.)

A comprehensive study of the effect of packet size, DFA size, *etc.* involves deep knowledge of the architecture of our switch – its memory capacity, a method to count the number of recirculations, and so on. We also intend to check if it is possible to further improve performance by taking "larger steps" – taking slices of multiple bytes in length, rather than a single byte, every time a packet passes through the switch. These tasks are the direction of our immediate future work.

Deep Packet Inspection is not limited to URL matching. Can *DeeP4R* be used to match other patterns, such as Snort signatures?

The mechanism used in *DeeP4R* can match *any* string, or indeed any regular expression; it is not limited to URLs. *DeeP4R* can therefore be used to match known keywords and other patterns (so long as they are contained in a single packet). However, there are two constraints we must mention.

The first constraint is that the target string must be available in plaintext in a single traffic packet. Owing to the popularity of HTTPS, most Web traffic is now encrypted. We note that over the past decade, firewalls and Network Intrusion Detection Systems (Snort, Zeek) have become more constrained in their DPI capabilities because of the lack of plaintext traffic; this issue affects *DeeP4R* as well. As a partial solution, some firewalls man-in-the-middle TLS connections to be able to inspect their traffic. At present, this "bump-in-the-wire" approach is not a design goal of *DeeP4R*, so we suggest it is best used for DPI with unencrypted traffic (HTTP) or for strings that are available in plain text even in HTTPS (server name indication in the ClientHello, *etc.*).

The second constraint is more subtle. Snort – and UNIX tools in general – offer a syntax called "Perl compatible regular expressions" (PCRE), rather than true regular expressions that correspond to DFA. PCRE extend regular expressions with features such as backlinks, that make them strictly *more powerful*; as a result they cannot always be matched by DFA and require a top-down parser [29]. As Snort allows PCRE expressions, we cannot state that *DeeP4R* can be extended to match *all* patterns matched by Snort, but only those that are (formally) regular expressions.

VI. Related Work

SDN switches with a programmable data plane have been used in a wide range of network functions such as load balancing [30]–[32], telemetry [33], and offloading tasks from servers [34]. Recently, they have also made a substantial impact in network security [35]–[43], in particular, in detecting

and protecting against attacks such as port scans and distributed denial-of-service attacks. However, these contributions are focused on clever manipulation of flow-level information from packet headers – for instance, a scan would be indicated by many flows in quick succession with the same source IP but different destination, while for a DDoS attack it is the opposite. These contributions show the importance of programmable data plane, but *do not use Deep Packet Inspection*. In this section, we explain how our work with *DeeP4R* fits into the overall research area of programmable data planes, and especially network security using such programmable switches, with special attention to the systems that inspired *DeeP4R* and systems offering a complementary approach.

A. Network Security with Programmable Data Plane

In the most general case, data plane programming is not limited to P4, and includes other approaches such as the Click modular router [44], and Vector Packet Processors [45]. These approaches apply a directed-graph of transformations, such as header rewrites. P4 however, has become a standard platform supported by various manufacturers as well as the research community, so we use its standard PISA programming model [46] as the platform for our work.

P4 was originally standardized as the $P4_{14}$ language, but the more current version is $P4_{16}$ [11], [47]. (We note that Budiu et al [11] is the source of our assertion that the community does not consider P4 to be capable of DPI.) P4 is highly versatile and can run on various target architectures, such as the basic v1model, PSA and its simplified version SimpleSume [48], and the Tofino Native Architecture (TNA). We note that while our implementation is on a Tofino-based switch, we avoid architecture-specific features, so our code should work with other switches also.

As mentioned above, P4-compatible switches have previously been used to build stateful or stateless firewalls in the data plane [49]–[51]; in particular, we make note of P4Guard [52], and Gallium [53]. These works build on the traditional approach, using SDN switches [54] and even plain switches/routers as network-layer firewalls [55], through the examination of link-layer, network layer and transport layer headers. As they do not touch the TCP or UDP payload, they cannot perform application-layer firewalling or Deep Packet Inspection, and are therefore complementary to our work.

B. Deep Packet Inspection attempts with P4

One early example of DPI in the programmable data plane, Meta4 [15], 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.

The other closely-related work we are aware of, P4DNS [14], 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 only parses domain names of limited length, but is a potential complementary approach to *DeeP4R*, which works with HTTP(S) traffic.

DeepMatch [16] is perhaps the closest match to our own work: it successfully performs Deep Packet Inspection (DPI) on packet payloads. The main difference with our work is that DeepMatch is developed using Micro-C, and targets the Netronome NFP-6000 SmartNIC; in other words, it requires custom logic to be integrated in the switch and will not run on a standard P4-compatible platform.

Finally, we come to the direct ancestor of *DeeP4R*: Jepsen et al's "Fast String Matching in PISA" [18], that first introduced the recirculation approach to find keywords in the payload. Their system is suitable for a "smart fabric" that consumes a packet (carrying a query) and returns the response directly, but not for a network switch or middlebox. We overcome this limitation in our work as explained in Section III, to build the first application-layer firewall in the data plane.

VII. Conclusion

In this paper, we design and demonstrate a new system, *DeeP4R*, that shows how DPI can be performed in the programmable data plane – thus making it possible to build an *application-layer* firewall on SDN switches. As expected for a data plane program, it shows excellent performance for its target DPI task (filtering of blocklisted URL), and is able to filter thousands of URLs at line rate, without loss of performance for non-blocked traffic.

We trust our implementation, which we make publicly available at [22], will draw attention to the fact that P4 can provide security more sophisticated than the traditional header field filtering. It also raises several new open problems, which we intend to explore in future work. In future, such a data-plane solution may become a very useful option for data center or enterprise network security.

References

[1] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[2] D. Balagopal and X. A. K. Rani, "Netwatch: Empowering software-defined network switches for packet filtering," in *2015 International conference on applied and theoretical computing and communication technology (iCATccT)*. IEEE, 2015, pp. 837–840.

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

[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, Accessed: 2022-05-25.

[5] "Sonicwall supermassive series," https://www.sonicwall.com/medialibrary/en/datasheet/datasheet-sonicwall-supermassive-series.pdf, Accessed: 2022-05-25.

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

[7] D. Achenbach, J. Müller-Quade, and J. Rill, "Universally composable firewall architectures using trusted hardware," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2014, pp. 57–74.

[8] T. K. Yadav, A. Sinha, D. Gosain, P. K. Sharma, and S. Chakravarty, "Where the light gets in: Analyzing web censorship mechanisms in india," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 252–264.

[9] C. Dixon, A. Krishnamurthy, and T. E. Anderson, "An end to the middle." in *HotOS*, vol. 9, 2009, pp. 2–2.

[10] "Cisco series 8000 data sheets," https://www.cisco.com/c/en/us/products/collateral/routers/8000-series-routers/datasheet-c78-742571.html.

[11] M. Budiu and C. Dodd, "The p416 programming language," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 5–14, 2017.

[12] 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)*, 2012, pp. 323–336.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] "Open Source," https://zenodo.org/badge/latestdoi/574774753.

[23] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[24] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 58 845–58 858, 2020.

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

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

[27] A. Bianco, P. Giaccone, S. Kelki, N. M. Campos, S. Traverso, and T. Zhang, "On-the-fly traffic classification and control with a stateful sdn approach," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[28] H. B. Acharya and M. G. Gouda, "Firewall verification and redundancy checking are equivalent," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 2123–2128.

[29] R. Cox, "Regular expression matching can be simple and fast," 2007. [Online]. Available: https://swtch.com/ rsc/regexp/regexp1.html

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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 *27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

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

[44] 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.

[45] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, and C. Kim, "Pvpp: A programmable vector packet processor," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 197–198.

[46] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "Pisa—a platform and programming language independent interface for search algorithms," in *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 2003, pp. 494–508.

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

[48] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *arXiv preprint arXiv:2101.10632*, 2021.

[49] 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.

[50] 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*, 2020, pp. 595–612.

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

[52] 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.

[53] 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.

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

[55] A. X. Liu, *Firewall design and analysis*. World Scientific, 2010.