5-2023

Network Security With Smart Switches

Sahil Gupta
sg3526@rit.edu

NETWORK SECURITY WITH SMART SWITCHES

by

Sahil Gupta

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
May, 2023

NETWORK SECURITY WITH SMART SWITCHES

by

Sahil Gupta

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Hrishikesh B. Acharya Date
Dissertation Advisor

Dr. Yin Pan Date
Dissertation Committee Member

Dr. Minseok Kwon Date
Dissertation co-advisor

Dr. Sumita Mishra Date
Dissertation Committee Member

Dr. Bruce Hartpence Date
Dissertation Defense Chairperson

Certified by:

Dr. Pengcheng Shi. Date
Ph.D. Program Director, Computing and Information Sciences

NETWORK SECURITY WITH SMART SWITCHES

by

Sahil Gupta

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

The inspection of packet contents, Deep Packet Inspection (DPI), is an important component in network security. However, DPI is provided by complex black-box firewalls which the network administrator has no choice but to trust. This raises the question: Can network administrators build their own DPI-capable filter using a standard programmable switch?

The commonly-accepted answer is that standard switches are not powerful enough; the standard they support (the P4 language) does allow users to specify how to parse packet headers, *but not packet payload fields (e.g. URL)*, as required by DPI. Even though software-defined networks are quite capable of handling various tasks, ranging from firewalling to flow analysis, these are all based on intelligent use of packet headers. DPI tasks, like URL filtering, still require dedicated middleboxes – or, if we insist on SDN solutions, middleboxes *in addition to* SDN. If we insist on developing a solution on the switch itself, we need either custom switch hardware, or heavy support from the SDN controller or an external firewall.

This dissertation challenges this common consensus. For our first contribution, we demonstrate that clients send packets with a predictable structure, so a P4 switch *can* perform some DPI (enough for URL filtering). We then develop and demonstrate a URL-filtering firewall, DiP, completely in the data plane, taking no external help from the SDN controller, firewalls, etc. DiP is a proof-of-concept, but is quite robust, handles multiple protocols (HTTP(S), DNS), and outperforms standard netfilter firewall by orders of magnitude.

However, DiP is not truly a general firewall: it is very specifically a URL filter, and it depends on the strong constraint of predictable URL location in a packet, which may not hold in future. Thus for our final contribution, we present a novel approach that allows *general* 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 DeeP4R, the first general-purpose 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. We can therefore argue with assurance that any platform that supports P4 is powerful enough for Deep Packet Inspection, and in future it may be possible to use programmable switches for this task, rather than dedicated firewalls.

Acknowledgments

The completion of this dissertation and the research behind it would not be possible without the guidance, support, and encouragement from many individuals. I would like to take this opportunity to express my earnest and heartfelt gratitude towards them.

First and foremost, I would like to give special thanks to my advisor, Dr. H.B. Acharya, for his constant support and mentorship throughout my Ph.D. tenure at Rochester Institute of Technology. He has taught me how to become a good and effective researcher. The meaningful discussions that we have shared during my research have been truly inspirational to me.

I would also like to express my sincere appreciation to my Ph.D. dissertation committee members, Dr. Miseok Kwon, Dr. Sumita Mishra and Dr. Yin Pan. Without their critical observations, suggestions and thoughtful feedback it would not have been possible to come up with all the solutions to the research questions answered in this dissertation. I thank them from the bottom of my heart for managing time from their busy schedule for all of my research review meetings. I would also like to thank Dr. Bruce Hartpence for serving as my dissertation defense chair. I would also like to thank Dr. Pengcheng Shi for the support throughout my Ph.D. tenure. A very special thanks to Lorrie Jo Turner and Min-Hong Fu who helped me a lot by taking care of all the administrative task related to my Ph.D. life and let me focus more on my research work. I would like to thank my friends and lab mates Garegin Grigoryan , Gaurav Wagh, Payap Sirinam, and others for their support and encouragement throughout this entire Ph.D. journey. I would like to express warm thanks to Dr. Devashish Gosain, of Max Planck Institute, for his help in every step of the journey. I also thank Dr. Nate Foster of Cornell University, Vladimir Gurevich (from Intel), Andy Fingerhut (from Intel), open source community of p4.org and ICRP community of Intel, for their help in learning about the P4 platform and data plane programming. Finally, I am grateful to have wonderful friends and family who supported me emotionally throughout my journey at RIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis focuses on the use of software-defined network (SDN) switches for deep packet inspection (DPI). We begin with this introductory chapter, describing what these terms mean, why it might be important to perform DPI purely in the data plane, and what our contributions are, ending with a brief summary of the structure of the dissertation.

Software Defined Networks (SDN) are networks with flexible switches. The switches have match-action tables called *flow tables*, which allow them to operate on packets, and these tables can be written to the switch on-the-fly using standard protocols (such as OpenFlow). The ability to configure flow tables, and thus the logic of packet processing, is decided by a separate machine or process called the SDN controller. The main value-add of SDN is that the packet processing logic is flexible, and under the control of the network admin using a standard API (in contrast to closed switches, which can only be configured using manufacturer-specific configuration languages).

Switches and routers – particularly Software-Defined Network (SDN) switches – have been successfully used to implement network-layer firewalls [57], flow analysis [16], 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 [21].

The current state-of-the-art in DPI is still provided by old-school dedicated middleboxes, such

as Cisco Firepower Threat Defense [2], SonicWALL TZ/NSA/SuperMassive Series [11], Fortinet FortiGate [3] , 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 [14], does not violate user privacy, etc). Further, such middleboxes are usually on-path rather than in-path [61], 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 [27]. And finally, such firewalls are not only black boxes taken on trust by the network administrator; they are hard to audit, present a high-value target for attacks, and compromise many users when they leak.

It is interesting that Deep Packet Inspection is *not* implemented using programmable switches, when there already exists a standard language (P4) that allows users to specify packet schemas[1]. Naively, this should imply that a programmable switch can parse packets and extract fields from HTTP, TLS, etc., headers. As soon as a switch can (extract and) filter traffic by, e.g., site URL or file type, it becomes an application layer firewall. What is the reason that such solutions do not replace (or at least compete with) black-box firewalls?

In the early days of SDN, researchers did propose such ideas – for example, Sekar's CoMB architecture [56] built on the Click modular router [42]). But *current SDN platforms are not intended for Deep Packet Inspection*[2]. The $P4_{16}$ standard makes this explicit [19].

- 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.
- The above cases are required to parse headers of important application-layer protocols, such as HTTP. (HTTP has 47 fields, which are mostly optional; important fields for filtering, such as URL, are variable-length).

[1] 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.

[2] Most likely this decision was made to ensure that complex parsers do not slow down packet processing.

Thus while P4-compatible switches have some flexibility, *it is not straightforward to use them for general DPI.* If a network admin wishes to build their own DPI-capable infrastructure, the consensus is that they must *either* use specialized platforms – eg. nVidia DPU [7], 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 [34], and provide enough servers to process traffic at line rate. This is a very different proposition than adding some off-the-shelf programmable switches to the network, and it is hardly surprising that the admins of enterprise networks and ISPs prefer to invest in a standard commercial middlebox.

At this point, we make an important observation. *An application-layer firewall can be valuable even if it only performs a few simple cases of DPI.* More involved DPI, such as content censorship (eg. 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. And the URL is usually present in plaintext in HTTP traffic, in HTTPS traffic (the Server Name Indication field), and in DNS traffic. If it is present *at a predictable location* in network packets, then this common case of DPI can indeed be solved.

We now come to our first contribution.

- In chapter 4, we report on our field study, which shows that even for theoretically "free-form" protocols such as HTTP(S), the header has a predictable structure in actual web traffic. Hence, these protocols *can* reliably be parsed in the data-plane, and the domain name extracted.

  In other words, even simple SDN switches (not designed for DPI) can perform URL filtering in practice, thanks to the predictability of browser clients, and the low rate of adoption of more-secure protocols such as encrypted SNI and DNS-over-TLS.

Our study shows that 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.* This immediately raises the question of how such a (limited) firewall can be implemented, and what its performance would be like. This brings us to our second contribution.

- We develop DPI-in-P4 (DiP), a dataplane firewall capable of simple deep packet inspection, on a real, cheap, easily-available SDN switch (Netberg Aurora 710) [5] [3]. DiP works with

[3] Our implementation runs on the Intel P4-based Tofino ASIC [4], but it can be ported very simply to another

multiple protocols – HTTP, HTTPS, and DNS – and to our knowledge, is the *first* filter to block URLs directly in the data plane.

In terms of *scalability* and *performance*, DiP 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.

DiP 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. It may be considered one important, specific case of limited DPI, like some others built with P4-programmable platforms [41, 60]. However, it may be argued that such a solution is still too partial for general use. If we want DPI in SDN, the choice is still between systems where the switch leaves a portion of the work to an external server [34], systems that require specialized hardware and are thus implemented in eg. NetFPGA (not on a standard switch), or our system, which relies on the incidental fact that real traffic *at present* satisfies some strong conditions [30]. In other words, even though we have built 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)*. This brings us to our final and most important contribution.

- Deep Packet Inspection in P4 using packet recirculation (Deep4R) is a *general system* to perform Deep Packet Inspection using only standard SDN switches and the P4 dataplane programming language. 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, and without strong assumptions about the traffic packets.

  When a packet arrives, we use P4 functions to clone it, then apply the recirculate-and-truncate method of pattern matching [35] 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.

platform such as the Broadcom NPL ASIC [6], as we *only use standard P4 functionality* to parse packets, extract fields, and trigger actions.

There are still Deep Packet Inspection tasks, such as inspecting encrypted traffic, that will defeat DeeP4R; but it is perfectly capable of application-layer firewall tasks such as URL filtering or matching other strings such as keywords.

In addition to the system design and analysis, we implement, demonstrate, and benchmark the *scalability* and *performance* of both our systems DiP and Deep4R. We are happy to report that as dataplane programs, they 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. Details are provided in Chapters 8 and 9.

We also note that our implementation is developed and run on the Netberg 710 switch [5], built around the Intel P4-based Tofino ASIC [4] – a commodity switch with a standard architecture (market cost roughly $5000). Besides the easy accessibility to small network admins, we note that our code can be ported very simply to another platform such as the Broadcom NPL ASIC [6], 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 [8].

This dissertation covers the development of first, a specific (DiP), and later, a general (DeeP4R) solution to the problem of building the first application-layer firewalls in the data plane (without external help or custom hardware). In the following Chapters, we cover the necessary background and related work, our field study of domain name predictability, the DiP and DeeP4R systems and their experimental evaluation, and finally end with a discussion and some concluding remarks.

Chapter 2

Background: P4 and Programmable Switches

This chapter is a brief overview of the P4 language and programmable switches that support it.

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 [45]. Such a switch is said to have a *programmable dataplane.*

Figure 2.1 shows the existing internal architecture of the Tofino switch ASIC. It is a 4-pipe switch. Each pipe has its Ingress and egress sub-pipeline. Each sub-pipeline has its Match Action Units (MAU). Each MAUs has its own TCAM and SRAM memory blocks used for various purposes.

We explain the necessary components below.

- *Ingress Parser block:* 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 al-

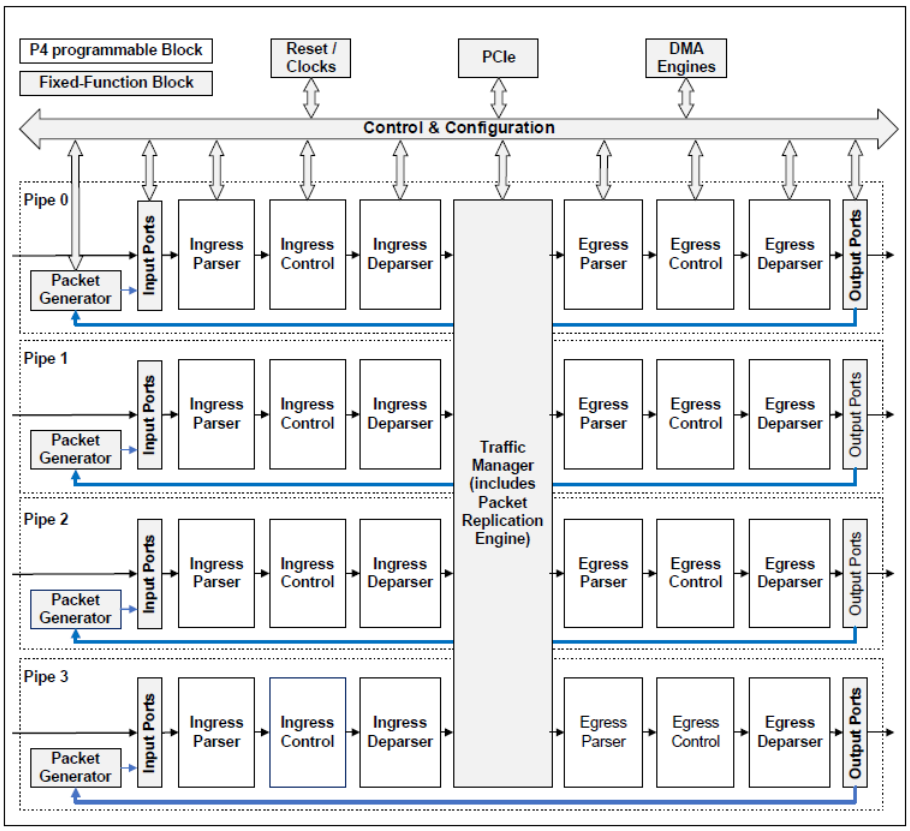

Figure 2.1: 4 Pipe Tofino ASIC [1]

lows 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.

- *Ingress control block:* The control block implements user-defined policies for packet classification. The control block 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. Some ports can be configured as loop back port. Setting egress_port value to such ports recirculate the packet. The other actions can be to drop a packet, to set a target egress port for output, etc These tables are set by the control plane. There are limited storage and computing capabilities provided by the Switch ASIC.

- *Ingress Deparser block:* 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. To mirror the packet a Mirror extern API call is made from this block.

- *Traffic manager:* In between ingress and egress blocks of a switch, there is a traffic manager – which is *not* programmable using the P4 language, and schedules which egress port to forward the packet to.

- *Egress Parser:* Once the ingress pipeline processes the packet, it enters the egress pipeline. The first block is the egress parser. Similar to the ingress parser, we can define how to parse the packet in this block and fill packet headers structures values with it. We can also initialize user-defined metadata values if we want. The parsing logic here can differ from ingress parsing logic per packet processing program requirements. We can also carry forwards some data from the ingress pipeline in egress for further processing.

[1] *Metadata*, in a programmable switch, refers to special data structures where a user program can store information generated during packet processing. It has SRAM and TCAM memory blocks that stores key field values from the control plane. There are three stateful P4 objects, i.e., registers, counters, and meters which can be used to build stateful data plane applications. For our Deep4R work, we used a register to store the results of packet analysis. They also used SRAM to store values in them. Metadata can be *intrinsic* to the switch or specified by the data plane program. Some fields such as timestamp, are read-only, others such as egress_port, can be modified (eg. to control where packet should be output).

- *Egress control block:* Similar to the ingress control block, here also we can define some temporary variables to carry intermediate results and match-action tables. We can also used register statefull feature to store some results in them like the way we had in ingress control block.

- *Ingress Deparser block:* Finally, all modified headers get united and packet moves out from QSFP port of switch.

- *ePCI bus:* This hardware component connects the switch ASIC with onboard small computer. The server program that intract with switch ASIC runs on it. PCIe bus helps to pass control instructions to and fro from switch ASIC to control plane server program. That program helps in control switch ASIC behavior for given data plane program from some remote machine. This flexibily helps to control multiple switches from a single control plane node.

Chapter 3

Related Work

SDN switches with a programmable data plane have been used in a wide range of network functions such as load balancing [23, 38, 52], telemetry [40], and for offloading tasks from servers [46]. More recently, they have also made a substantial impact in network security tasks [15, 26, 29, 39, 43, 44, 48, 49, 51, 53, 63]. In this chapter, we explain how our work 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.

3.1 Programmable Data Plane and the P4 Platform

In the most general case, data plane programming is not limited to P4, and includes other approaches such as the Click modular router [42], and Vector Packet Processors [22]. 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 [18] as the platform for our work.

P4 was originally standardized as the $P4_{14}$ language, but the more current version is $P4_{16}$ [9, 19]. (We note that Budiu et al [19] 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 [32], and the Tofino Native Architecture (TNA). Our own work is implemented on a Tofino-based switch, we avoid architecture-specific

features, so our code should work with other switches also.

3.2 Software-Defined Network Security

In recent years, software-defined networks – and P4-compatible platforms – have made a substantial impact in network security [15, 39, 43, 44, 48, 49, 51, 53, 63]. In particular, we will mention their use 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.*

P4-compatible switches have previously been used to build stateful or stateless firewalls in the data plane [20,37,54,55,59]. In particular, we make note of P4Guard [25], and Gallium [62]. These works build on the traditional approach, using SDN switches [33] and even traditional switches/routers as network-layer firewalls [47], through the examination of link-layer, network layer and transport layer headers. As they do not touch the TCP or UDP payload, they also cannot perform application-layer firewalling or Deep Packet Inspection, and are therefore complementary to our work.

3.3 Deep Packet Inspection with P4

We now go on to consider the most directly-related papers, i.e. those that study the question of Deep Packet Inspection in the data plane.

One early example of DPI in the programmable data plane, Meta4 [41], 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 [60], 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 [34] 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.

3.4 Immediately Related Work

Finally, we come to the direct ancestor of Deep4R: Jepsen et al's "Fast String Matching in PISA" [35], 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. In upcoming chapters, we explain how this limitation is overcome to build the application-layer firewalls in the data plane.

We ourselves began work in this area in collaboration with the authors of the Fast String Matching paper. This earlier joint work [58] involved building a standard SDN firewall, in the control plane. Any new device added to the network sends the SDN controller a config file, carrying the device manufacturer's identity and forwarding path information, and the controller then verifies whether it is secure to route that type of data or not. Based on this analysis, it adds allow or deny rules to SDN switches, thus implementing an Access Control List for IoT devices on the network.

We came to realize, in the course of our work together, that while the problem of "SDN based access control in IoT" was interesting, the side-problem of "deep packet inspection with SDN" was more fundamental and interesting. In the course of the next few years, we performed an in-depth study of this problem, starting with the demo paper [31],where we first learned that actual traffic may have characteristics that make it simple to filter (i.e. predictable URL positioning in the packet), as explained in Chapter 4. In this thesis we build on this work, studying the constraints required for a solution and demonstrating and benchmarking an actual DPI-capable firewall implementation in the data plane.

Chapter 4

Domain Name Field Study (Predictable Internet Clients)

The main challenge in parsing packets to extract URL is that the length and position of the URL are variable. As discussed in Chapter 2, this poses an issue for the P4 parser. However, this issue may not be insurmountable. In practice, *almost all users access the Web using one of a small variety of clients* [12]. 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. This chapter presents our field study, which gives us the confidence to assert that (at least at present) this is indeed the case. It also introduces a new challenge to packet parsing (the tension between *parse accuracy* and *match accuracy*), and how we respond to this difficulty.

4.1 Model of Protocol Packets.

Figures 4.1, 4.2, and 4.3 show how 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.[1]

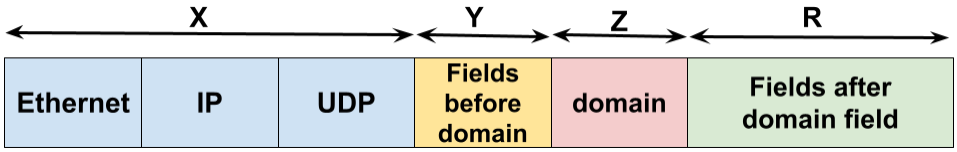

Figure 4.1: Partition of DNS packet.

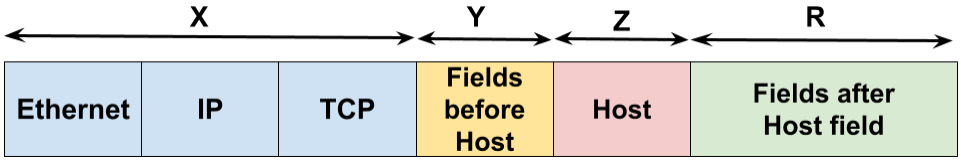

Figure 4.2: Partition of HTTP GET packet.

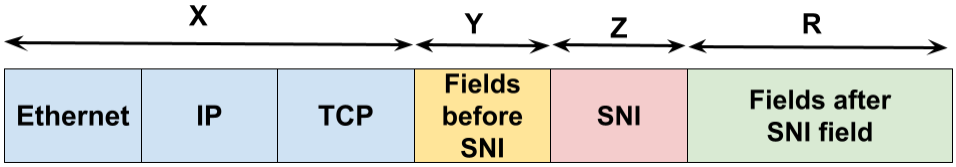

Figure 4.3: Partition of TLS Client Hello Packet.

4.2 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.)

[1] 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 4.1: URL position in packets, as Min Start – Max End.

Our results appear in Table 4.1. For example, when Firefox, Chrome, and Edge browsers on Windows 10 OS access Alexa top-10k websites, the generated 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). HTTP shows minimal variation, and HTTPS shows more, but it is limited enough to cover by case-by-case enumeration.

4.3 Choosing Start and End Positions.

From the previous section, it is clear that there is a small range of positions where the URL can be found. More precisely (as per our model): 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 5.3 explains how we handle additional non-URL bytes that appear in the slice.)

However, when we naively tried to parse packets using the ranges in Table 4.1, we found an unexpected challenge. For some packets with short URL, the *entire packet* ends before the 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 4.2: Parsing and Pattern-matching Accuracy, Alexa top-10k sites
(using the given start and end positions to extract URL).

position we specified. More precisely: if we try to match a slice that starts at (or before) Y_{min}, and ending at (or after) $Y_{max} + Z_{max}$, for some small packets we are asking the parser to fetch a field that *extends past the end of the packet*. The parser responds by ignoring the packet entirely.

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 at the same time, (2) in a high percentage of parsed packets, the URL lies between our chosen start and end points (we call this *match accuracy*) [2].

Table 4.2 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.

[2]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.

Chapter 5

DiP: Simple In-Switch Deep Packet Inspection

This chapter presents DPI-in-P4 (DiP), our first generic (multi-protocol) application-layer firewall in the data plane. We begin with an overview of the simple idea, then mention the challenges we face and the assumptions we make, and end with a description of how the system works.

5.1 Overview : How DiP Works

Our firewall takes two inputs from the user, i.e. 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.

- 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.

Speaking simply, DiP operates exactly like an IP-level filter, but with different fields of interest (i.e. URL).

In practice, this naive design needs to be modified because of two challenges, which we present below.

5.2 Challenges : DPI in the Data Plane

5.2.1 Challenge 1. Parsing

The "parsers" in P4 are not full-powered general parsers (defined as able to match any context-free grammar), or even 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 very difficult to extract variable-length domain names, especially as the start location in the packet is also variable.

- Our first attempt was to use the P4 `varbit` data type, which the parser can use to extract a field of variable length (eg. 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 the 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 (Chapter 4).

  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 actually use for DiP.

5.2.2 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. This makes programming a real switch different from an emulator, such as the P4 reference implementation BMV2 [10].

- Our system originally used different match-action tables for HTTP, HTTPS and DNS. While this worked, it was very inefficient and the number of URL's filtered by DiP was quite low (< 200). We considered adding the restriction that DiP can work with multiple protocols, but with only one at a time, so a substantial number of rules can be implemented on a typical switch.
- 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 DiP as a single match-action table. Luckly, this key size also comes under maximum allowed key size to be matched in Switch ASIC TCAM Match Action table. With this optimization, we are able to filter over 1000 domains per switch[1]; details follow in Section 5.3.

5.3 System Design and Implementation

The DiP system depends on the assumption that **there are predictable URL starting and ending points** for HTTP GET, TLS client hello, and DNS response packets, as we checked in our study in Chapter 4. We now explain the system design and the actual setup of the DPI-in-P4 (DiP) system.

5.3.1 Deep Packet Inspection with P4

DiP is a data plane 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.

[1]We add in passing that our switch is a simple Netberg Aurora 710, which is available for $ 5000. A high-end switch used in an ISP would be far more capable.

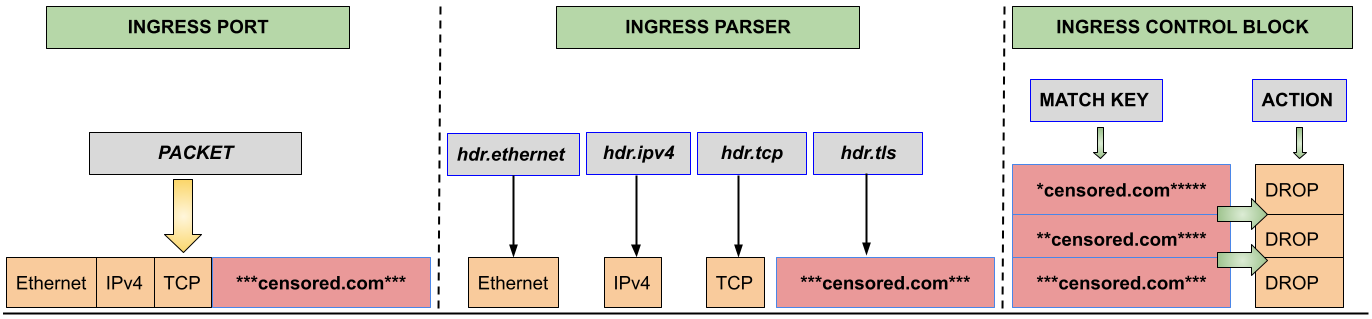

Figure 5.1: DiP: packet parsing and matching in the switch.
First, the packet arrives at the Ingress port. Next, the (ingress) parser separates different headers (eg. TCP), and particularly the user-defined header containing the URL. Finally, the control block matches the user-defined header field to see if there is a rule to drop it. On a successful match, the entire packet is dropped.

As we discuss in Background Section, match-action can be performed both in the ingress and in the egress pipeline. Our platform (Netberg-710, built with the Tofino ASIC packet processor) implements a full pipeline (parser, match-action, deparser) for ingress, a Traffic Manager for the buffer, and another pipeline (parser, match-action, deparser) for egress. However, as the aim of this research is to develop a dataplane firewall that can work on *any* P4-compatible architecture, we use only standard features from the P4 reference behavioral model BMV2 [10]. Our firewall matches and drops packets using TCAM tables in the ingress pipeline, i.e. immediately after the first parse.

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. We note that if the HTTP GET, HTTPS ClientHello, or DNS response packets cannot get through, this is sufficient to prevent a session with the website, and it is effectively blocked (see Figure 5.1).

Handling variation in URL length.

We have the challenge that in match-action tables, the key cannot be a *varbit*, i.e. a field of variable length. We must, therefore, ask the P4 parser to always give us a slice of the same length, though we know 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 greatest expected length*; in other words, when we receive the list of sites to block, we find the longest name (say, length Z_{max}) and always ask for the parser to extract a slice of length Z_{max} after position X + Y.

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, the length-to-URL in the remaining packet. Suppose Y varies from say 100 to 102, for a given protocol. Then for the above example, we need to make adjustments 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 need to insert three patterns:
"www.qq.com**********",
"*www.qq.com**********",
"**www.qq.com**********".

So if Y varies over Y_{min}, Y_{max} and Z over Z_{min}, Z_{max}, we will need to cover the entire range from Y_{min} to $Y_{max} + Z_{max}$, as the URL could be located anywhere in this range. Further, as one rule corresponds to a single position of the URL in the packet, we will have a *range* of TCAM rules for a single URL. We discuss how we optimize the amount of TCAM memory, so a switch can handle a substantial number of URL's, in Chapter 7.

Chapter 6

DeeP4R: Deep Packet Inspection in P4 using Packet Recirculation.

The Deep4R system implements a Deterministic Finite-State Automaton (DFA) in the switch, to match target strings (keywords, URLs) in the packet.[1] For example, Figure 6.1 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 [35]). The state of the DFA allows us to determine whether the target string has been seen, so we can match 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.[2] 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.

6.1.1 Architecture of Deep4R.

Deep4R includes two main methods.

[1] Note that our DFA state is separate from the parser state or the flow state (OpenState [17], 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.

[2] 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.

6.1 System Design

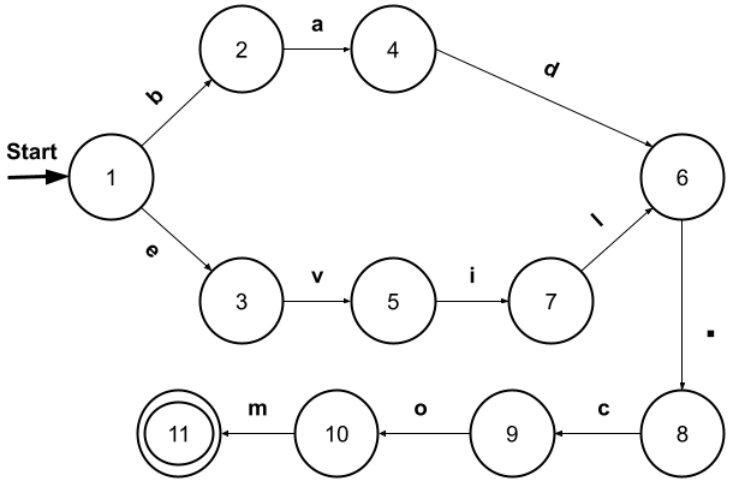

Figure 6.1: DFA to match evil.com and bad.com.

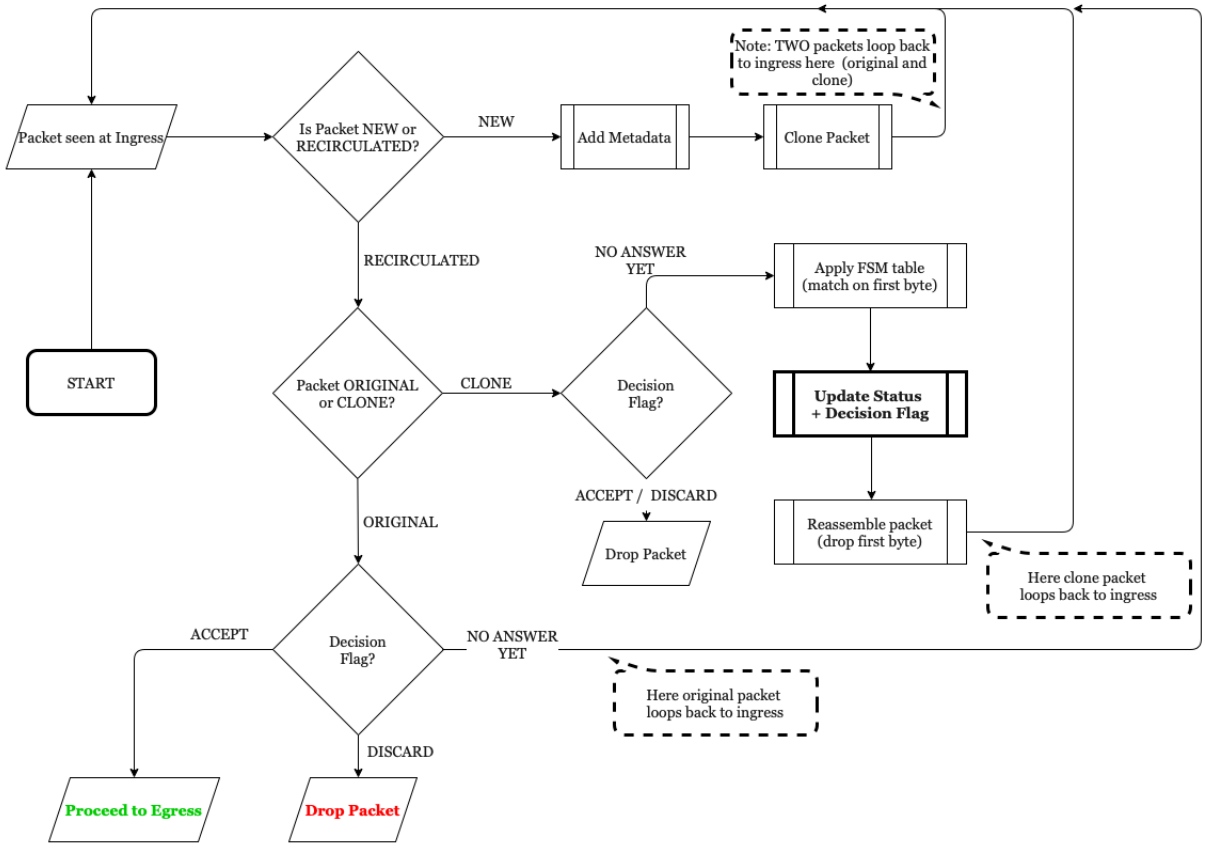

Figure 6.2: Life cycle of a packet processed in Deep4R. The *supervisor* table actions correspond to the decision blocks, and the *DFA* table corresponds to the bolded block "update status".

1. *Recirculation-and-truncation* [35] 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 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 6.1 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 6.2.

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 6.1: 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.

6.1.2 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).

- `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) eg.in Table 6.1, 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. eg. if 'e' appears after state 4 in Table 6.1, 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.

To wrap up this chapter, we once again recapitulate the design goals that inform our system and how these are met.

- **Preserving the original packet.** Our primary idea, to use recirculate-and-truncate to identify strings in a packet, has the weakness that the original packet is consumed. The goal of our design is to keep an additional copy of the packet, which can be forwarded intact (or dropped if the target string, i.e. URL is found).

We solve this issue by cloning the packet with the *mirroring* feature of standard P4. The cloned packet is consumed by the analysis, but the original remains (it is recirculated without truncation, while the clone is being analyzed).

- **Associating the clone and original packet.** The above design has the problem that the target string is found in one packet (the clone used for recirculate-and-truncate) while another packet, the original, is dropped or forwarded. The question follows how the decision made on one packet can be associated with another.

  This issue is addressed using the switch ASIC's storage space (called registers), which use SRAM memory to store information. Different packets are processed by different threads, but any thread has access to any register, so communication is simple. We compute an index (using the switch's inbuilt hash function), using packet header fields unique to the packet (and its copy). Information, such as whether a decision was reached (and what decision it was), is communicated by the register at the chosen index.

Chapter 7

Experimental Setup.

Our system designs, as explained in previous chapters, raise some important questions which need to be answered by experimental evaluation.

- Programmable switches are usually restricted in their memory, computational power, etc. Do the match-action rules (for DiP) or the finite-state machine (for DeeP4R) that we would need, fit in the memory available for flow tables in a commodity switch? Assuming a reasonable-sized firewall (say 1000 - 2000 URLs [13]), is there enough room for other match-action tables (eg. 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 do DiP and DeeP4R compare to the baseline performance of the switch, and to a traditional firewall? How well can they handle multiple flows and network congestion?

This chapter explains the experimental setup we use to test, and check the performance of, both our systems for Deep Packet Inspection on P4 (i.e. DiP and Deep4R). It will be followed by the chapter on experimental results.

7.1 Testbed Layout

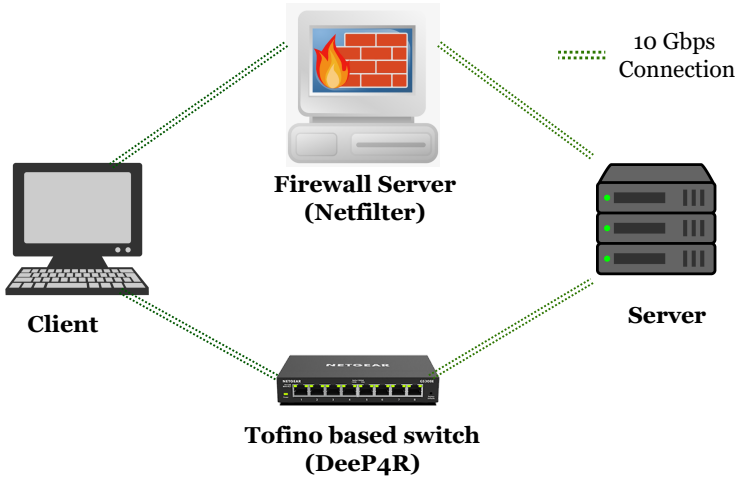

Figure 7.1: Experimental setup: Client machine fetches HTTP or HTTPS traffic (web pages, including large files) from Server. In separate runs, we pass identical traffic through our Tofino-based switch (running Dip / Deep4R), and through a server running Netfilter firewall, for a fair comparison.

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 (ie. the list of URLs to block) into the format required. In case of DiP, this is a simple match-action table the SDN controller uploads into the switch. In case of DeeP4R, this is a DFA, expressed 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 7.1, 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 used for filtering, or to implement a DFA on the switch), as well as the control-plane barefoot runtime python scripts (used to install the match-action rules, and also for other tasks like checking the data plane 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.

7.2 Testing Workflow

7.2.1 Switch Setup: DiP

Our first step is to build and deploy DiP 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 Chapter 4, 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.)

7.2.2 Switch Setup: DeeP4R

In deploying DeeP4R on the Tofino switch, the workflow is essentially similar, but instead of installing drop rules in the TCAM match-action tables we install the DFA transition table and supervisor table. The system design, our main challenges (the cloning and recirculation of packets, and tracking the relationship i.e. which cloned packet corresponds to which original packet), and how we address these concerns, are explained in detail in the system design section of Chapter 6.

7.2.3 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.

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.

7.2.4 Routing

Traffic is routed through two separate network interfaces to pass through our Linux firewall or through our switch running DiP/DeeP4R , 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.

7.2.5 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 HTTP / HTTPS / DNS as needed for the test).

7.2.6 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.

Chapter 8

DPI-in-P4 (DiP): Experimental Results.

8.1 Evaluation

This section covers our experimental evaluation of DiP (or more precisely, of our implementation of DiP 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 DiP and Nffw. Our experiments use the setup described in Section 7.

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 8.1 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 processing time

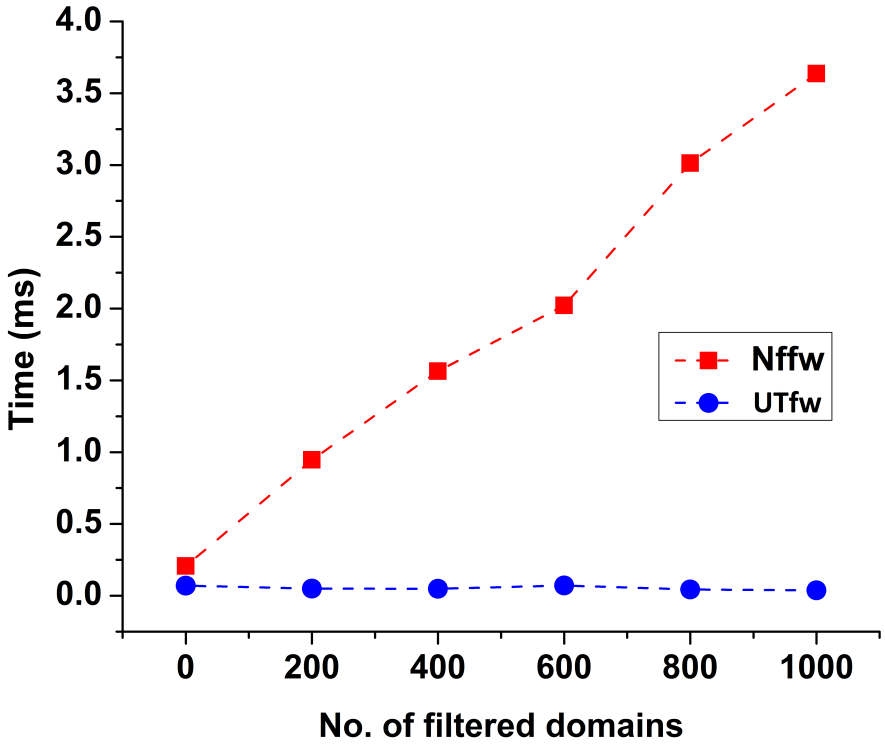

Figure 8.1: Packet processing time: Avg time a packet spends within firewall.

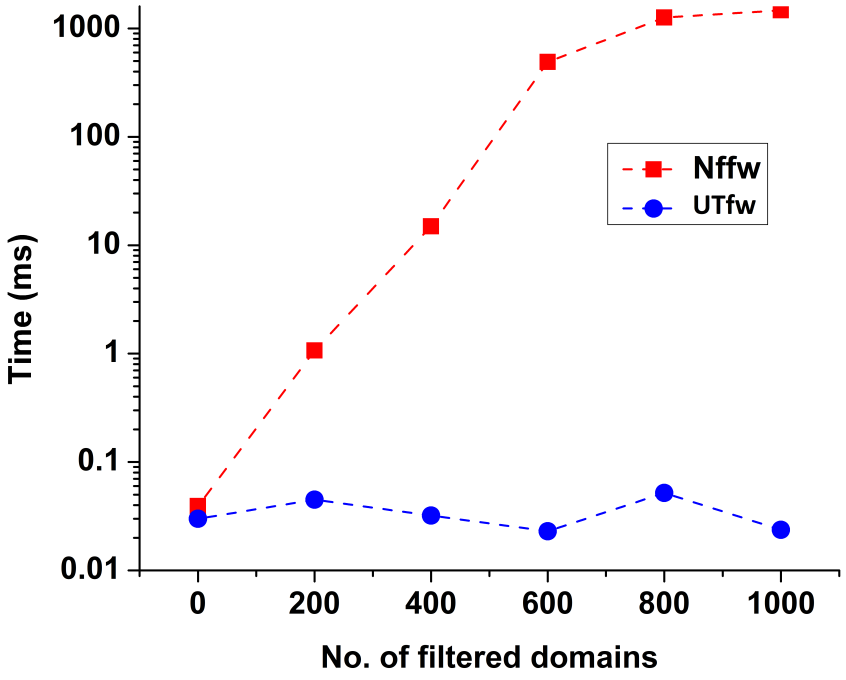

Figure 8.2: Packet processing time (log scale): With 10k flows through firewall.

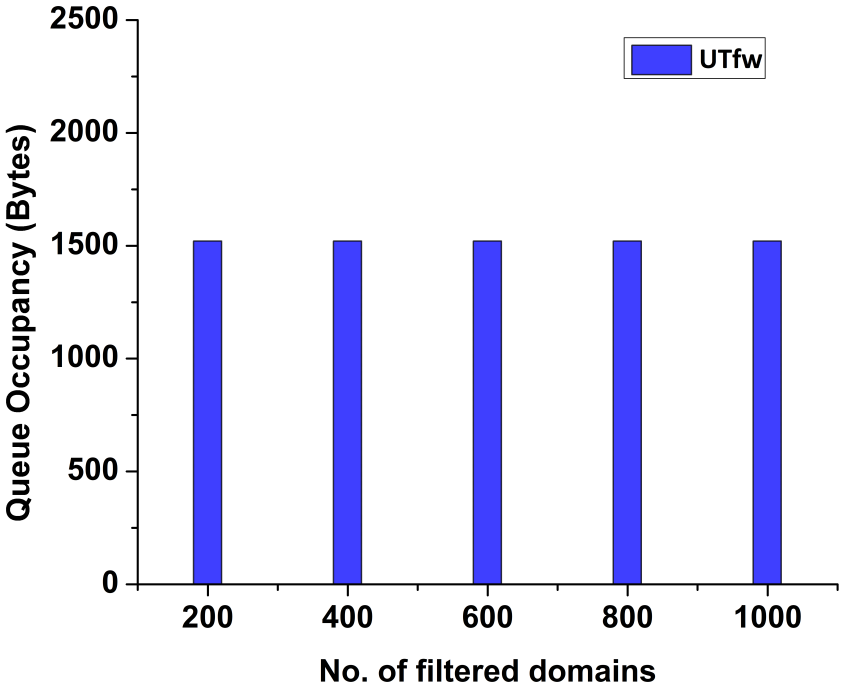

Figure 8.3: Queue occupancy: Under heavy cross-traffic (i.e. 10k flows), increasing firewall rules do not impact queue occupancy.

of Nffw increases steadily, but DiP 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 8.2 shows, even with 1000 filtering rules DiP 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 DiP (i.e. a speedup of 7×10^4 times). We conclude that with a large number of firewall rules, Nffw performs poorly compared to DiP . 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 8.3 shows, there is no such backlog of packets accumulating within the switch, even with 10k parallel flows, and even with large filter lists (1000 URLs).

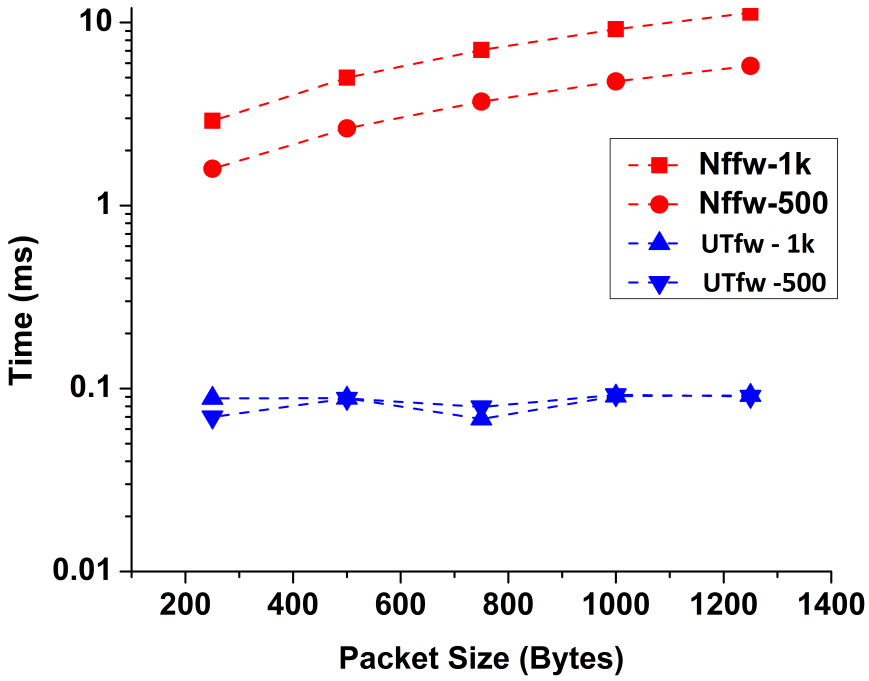

Figure 8.4: Impact of packet size on packet processing time.

Impact of packet size

Different applications generate packets of different sizes. Could applications generating larger packets cause congestion at the P4 switch running DiP? 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 DiP) configured to filter 500 domains, and then again for 1000 domains.

Figure 8.4 illustrates that for both 500 and 1000 domains, DiP not only consistently outperforms Nffw, but also does not lose performance with increasing packet size.

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

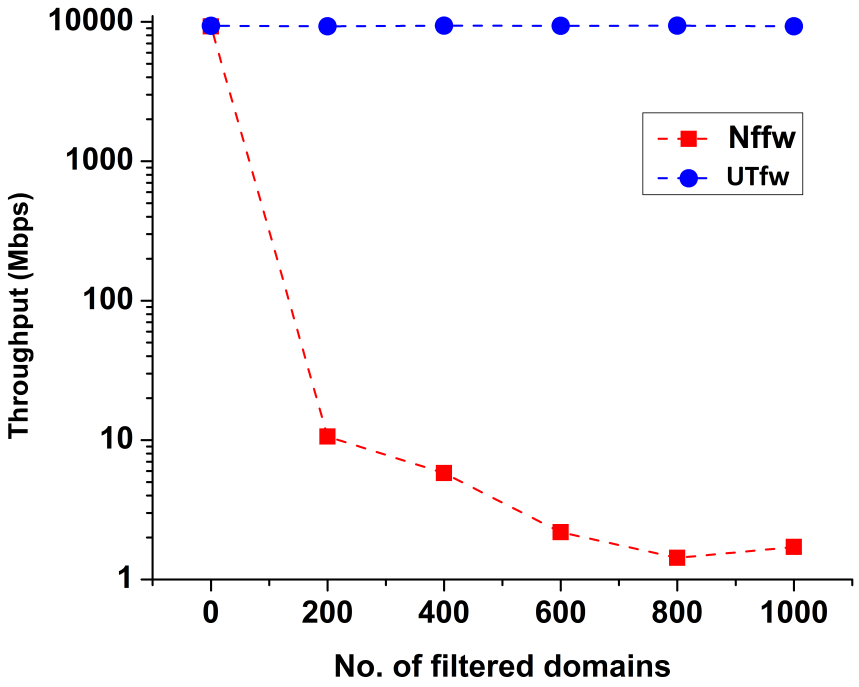

Figure 8.5: Throughput (log scale): Impact of increasing firewall rules.

we increase the number of filtered domains (and thus firewall rules).

As Figure 8.5 shows, adding firewall rules adversely impacts Nffw, but DiP shows no measurable impact. With no filtering rules, `iperf` reports nearly 10 Gbps throughput for both Nffw and DiP. However, with rules for 100 domains, Nffw reduces the throughput by 100×, while DiP shows no decrease at all.

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

Chapter 9

Deep4R: Experimental Results.

9.1 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. eg. when testing for 5 URLs we filter for `google.com`, `youtube.com`, `facebook.com`, `baidu.com` and `wikipedia.org`.)

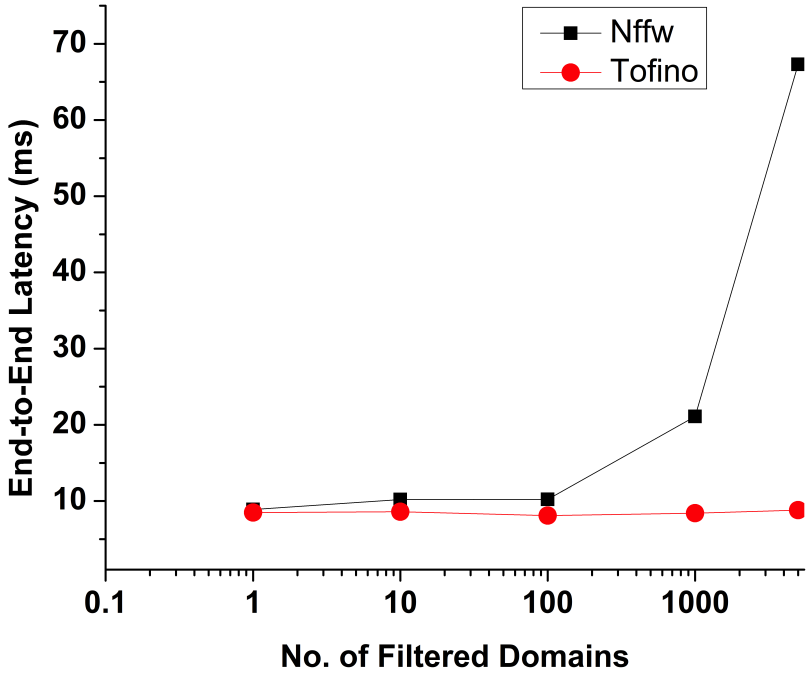

Figure 9.1: E2E Delay vs Filtered Domains.

As seen in Figure 9.1, 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 9.2 shows our results. Deep4R consistently introduces a delay of 0.2 ms, while we vary the

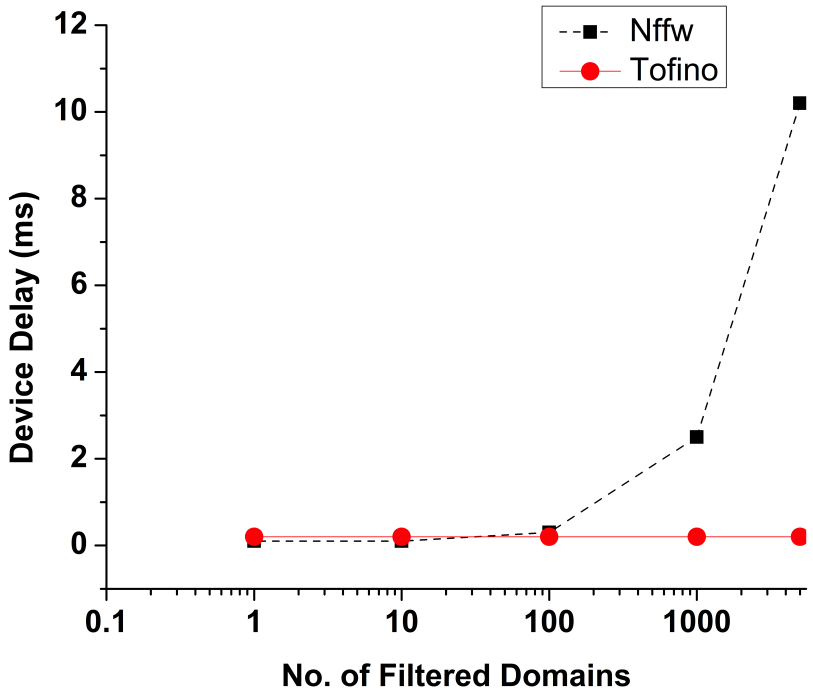

Figure 9.2: Device Delay vs Number of Filtered Domains.

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.

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 9.3 and 9.4, 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.

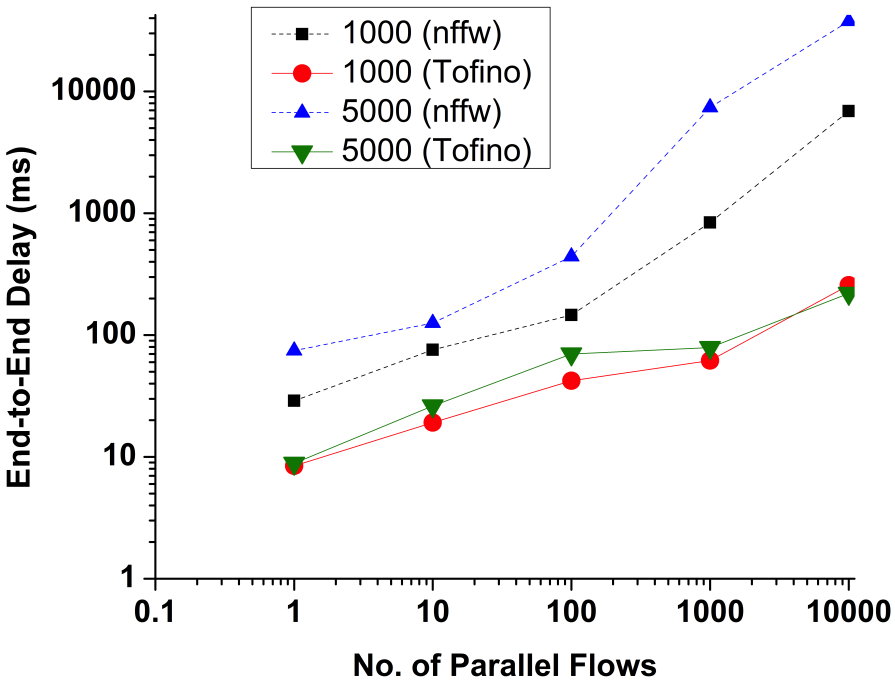

Figure 9.3: E2E Delay vs Parallel Flows.

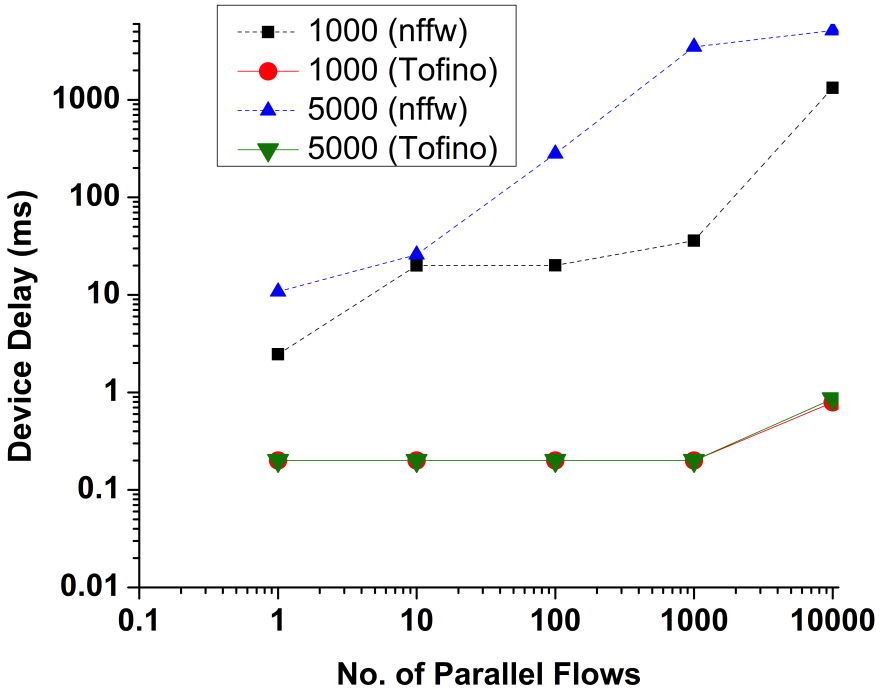

Figure 9.4: Device Delay vs Parallel Flows.

Of this, in Deep4R the device delay is only 0.2 ms for one flow and rises to 0.8 ms for 10k

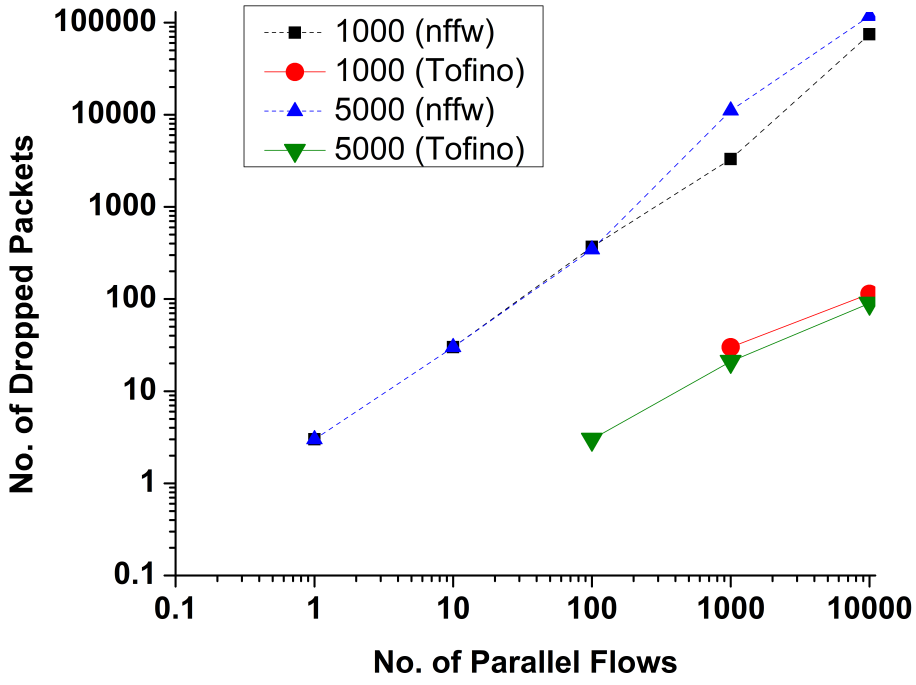

Figure 9.5: Dropped Packets vs Parallel Flows.

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.

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 9.5 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).

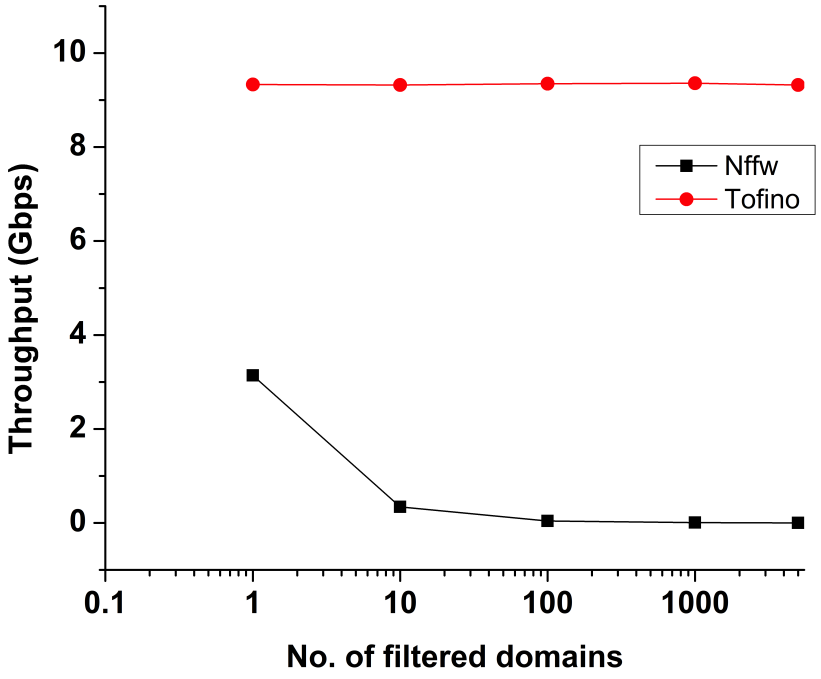

Figure 9.6: Impact of increasing firewall rules on Throughput.

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 9.6 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.

Chapter 10

Discussion: DiP and DeeP4R.

This chapter first presents the separate analysis and limitations for our two systems, DiP and DeeP4R. We then add a short discussion on their separate merits and how they may be brought together in a comprehensive solution. Finally, we consider our entire contribution in the context of existing systems as well as community needs, and conclude with a small discussion of future work.

10.1 DiP : Analysis and Limitations

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

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

DiP is a proof-of-concept system, and its primary purpose is to show that DPI is *possible* using P4-programmable 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 8.1 demonstrate that a switch can easily filter substantial traffic without any increase in latency or packet loss. The limiting factor is TCAM memory: it is indeed a concern when firewall rules fill up the memory and leave it unable to perform other functions such as packet routing. This concern is compounded by the fact that a single domain can translate to multiple

firewall rules; for instance, we require 21 rules to filter `censored.com` (see Section 5.3).

Our experiments demonstrate that we were able to create filtering rules for Alexa top-1k domains, using a standard cheap switch (Netberg 710). In fact, as we utilized $\approx 80\%$ of the switch's TCAM memory, even this low-end switch had the capacity for additional functions (routing, etc,) Hence while URL filtering is of the order of 100x less efficient than simple IP blocking (our switch could handle $\approx 147.5k$ IPv4 addresses using the same TCAM memory we used to block $1k$ URLs), a small standard switch is capable of implementing DiP with a decent number of URLs. A proper ISP-scale switch would of course have much higher capacity.

If actually used as a firewall, *we do not recommend the switch running DiP be used for other tasks.* An actual ISP or even enterprise admin would certainly use other switches for other required SDN applications, such as load balancing. If we DiP is run on a single smart switch – say, as the "gateway" providing Network Address Translation (NAT), IP-based Access Control (ACL), and URL filtering at the same time, eg. in a campus network – in such a case the blacklist would also be small, and there would be enough capacity to carry out these functions.

DiP 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, DiP *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 DiP is limited to simple DPI tasks that do not involve decryption.

However, we contend that even with this limited power DiP 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 5.3, 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. DiP cannot handle encrypted DNS packets, as in the DNS-over-TLS (DoT) or DNS-over-HTTPS (DoH) protocols [28]. But the percentage of DNS packets making

use of such encryption is quite negligible: Kim et al [41] report a figure of 0.09%. It will take considerable time for DNS-over-HTTPS to become popular enough to threaten firewalls.

Hybrid solutions (i.e. firewalls where DiP is paired with a middlebox, specifically to handle encrypted traffic) can be an interesting area of future work.

How robust is DiP to adversarial traffic? Can it handle short or fragmented packets?

While DiP 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 DiP inspection.

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 generally not 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.

Is DiP resistant to attacks that defeat current state-of-the-art middleboxes, eg. HTTP GET request with multiple URLs?

DiP is a proof-of-concept; it does not aim to resist attacks that can cause leaks in current firewalls. In practice, it *does* resist some of these attacks, such as sending multiple URLs in HTTP GET request separated by '$\backslash r \backslash n$' (because the active URL is the first one in the string, and will still be in the position matched by the TCAM table). It will not resist other attacks like mixed-case URL (GooGle.Com) unless we are willing to create a large number of rules to handle every possibility for a given URL.

With DiP, our aim is to demonstrate that a simple SDN switch can act as a DPI-capable firewall, and blacklist URLs in the data plane (and as a bonus, our tests show it can do this with excellent

performance). But if it is actually used as a firewall, there do exist known firewall exploits that would defeat DiP. Whether we can make a system resistant to these exploits using simple off-the-shelf switches and similar components is a promising open question for future work.

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

DiP is currently a proof of concept. Our 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 [9].*

We compare our implementation of DiP against netfilter as it is a standard (software) firewall [36, 50]. It is true that DiP wins the comparison; however, before DiP can actually be deployed as a solution at Enterprise or Internet scale, additional work remains to be done. In particular, while DiP can perform URL (and IP) filtering, it does not support advanced features like Keyword filtering, DoS or scan detection, etc. (which full-scale hardware firewalls do support). Future work could explore the development of a more comprehensive firewall solution, able to support such advanced features, and competent to be considered a full-featured firewall.

10.2 DeeP4R: Analysis and Limitations

In this section, we discuss the limitations, further work, and general comments about our more general Deep Packet Inspection system, DeeP4R.

DeeP4R is more general than DiP, and does not make strong assumptions regarding the location of URL in packet. But DiP works with DNS – can DeeP4R do the same?

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 [60], 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 other factors affect performance, and how does it do?

Though this is a factor that shows limited variation, *payload size* in the packet can also affect performance. 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.)

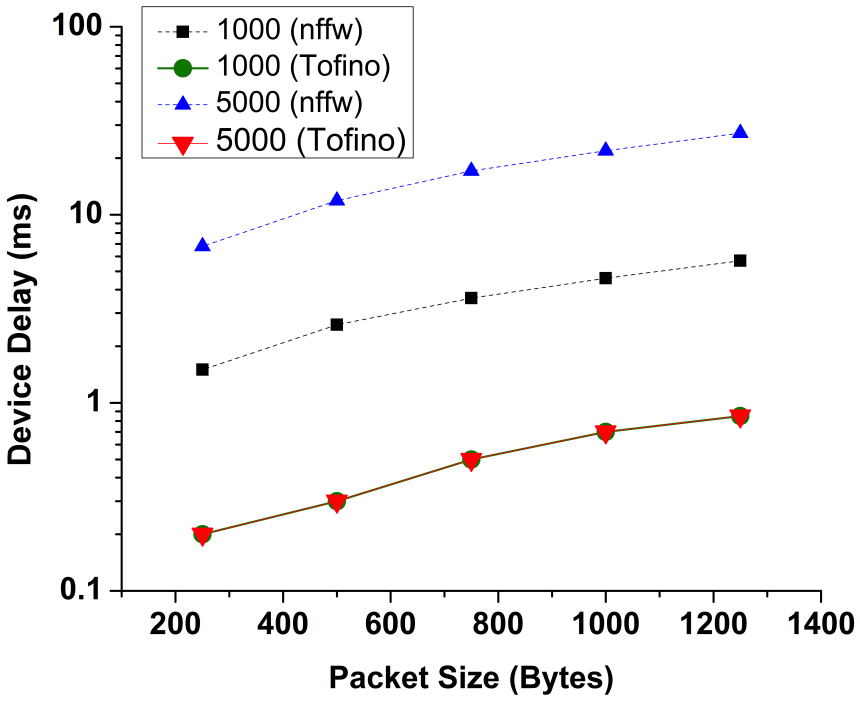

Figure 10.1: Device Delay vs Packet Size.

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 questions remain open, and are good candidate problems for 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 [24]. 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.

10.3 DiP and DeeP4R: working together?

We wrap up our discussion considering how DiP and DeeP4R relate to each other. Both are solutions to the problem of Deep Packet Inspection in the data plane; also, DeeP4R does not require the strong assumption of predictable location of URL string. How do our two systems relate to each other? Is DeeP4R clearly a better solution "in all ways"?

The answer is somewhat more nuanced. DiP is fast, as it extracts all possible bytes which could potentially contain the target string (domain name), and matches these extracted bytes against the list of domain names using a single match-action table lookup. In other words, it uses the switch primitives as they were meant to be used, and hardly causes any traffic slowdown at all. On the other hand, it has significant limitations: first, in the amount of bytes that can be extracted (to be matched as TCAM key), and second, in whether the key is long enough to contain even long domain names but short enough not to cause packet parsing to fail for short ones, as detailed in Chapter 4.

DeeP4R is a relatively slow system. It requires packets to be circulated through the standard P4 mechanism of recirculation; this induces latency, and as the target pattern has to be matched in all packets, it affects *all* traffic rather than simply the packets which are filtered. The positive of this method is that the entire payload is inspected, and further, it is not restricted to a given application-layer header such as domain name. This approach allows for the general detection of *any* pattern, and at *any* position in the packet.

A possible important direction of future work would be to build a "telescoping" system, where DiP is used to give a quick guess if the target pattern is present (maybe with a high rate of false positives), and then the specific packets it flags are passed to DeeP4R for more thorough inspection. Such a solution can be thorough without incurring a substantial slowdown. However, this system would still be reliant on the predictable structure of application-layer headers, to detect that packets are of interest in the first place.

10.4 Our Work In Context

10.4.1 P4-based Network Security, DiP and DeeP4R.

The main limitation of SDN-based security solutions, is that they focus on smart analysis of packet headers and flows. While there have been several interesting systems to detect, say, port scans [53] and Denial-of-Service attacks [15, 44, 63], very few systems have explored Deep Packet Inspection; indeed, the P4 standard itself states that it is not suitable for such uses [19]. However, in fact some Deep Packet Inspection *can* be performed in the data plane, and the work in this dissertation advances the boundary of this new area.

One major direction of research in dataplane programming focuses on special cases, mainly involving

the DNS protocol (which has a relatively predictable structure, URL always starts after 13 bytes, etc.). For instance, in Rexford et al's tool Meta4 [41], authors parse DNS headers to collect packet stats per domain name. However, even here the model of parsing is highly restricted, as it depends on parser state (the URL can have only up to four labels, each being up to 15 letters long; nationalgeographic.com or studiotwentyeight.com cannot be parsed). Similar restrictions apply for systems like P4DNS [60], which serve DNS responses from on-chip memory: there is only limited support for DNS, and none at all for more flexible protocols such as HTTP. Our work is much more general, and while we focus on URL detection, can be used to match *any* pattern in any traffic, so long as the pattern is present in plaintext in a packet.

The other approach to Deep Packet Inspection involves the use of specialized hardware, such as network processing units (NPU). A state-of-the-art system in this regard is DeepMatch [34], which not only performs DPI but is able to match patterns across multiple packets. However, the limitation of this approach is that it requires special hardware, which is not only expensive but is limited in the rate of processing. Specifically DeepMatch achieved 40 Gbps for simple matching, dropping to 10 Gbps when some packets were reordered, using a Netronome N6000. Our work is complementary as it uses standard switches (making it possible to do filtering or intrusion detection in a network and not just the edge), and may also scale to high line rates.

Finally, we consider the PPS approach [35] which first introduces the idea of matching patterns through recirculating packets. This algorithm is a direct predecessor to our own work, and may be considered part of the same family. Our contributions were, firstly, to pioneer the method of packet cloning so packets were not consumed during processing, and secondly, to use this approach to build a simple application-layer firewall (currently focused on URL filtering). We also made use of the unexpected regularity of real traffic to implement a simpler system for direct filtering (DiP).

10.4.2 DiP and DeeP4R: Impact.

The existence of our systems, DiP and DeeP4R, immediately makes it clear that it is indeed possible to perform Deep Packet Inspection in the data plane. Further, our results indicate that while the algorithm is likely quite inefficient (involving the recirculation of the entire packet, to create a loop), the overall system is quite performant. Indeed, we note that under the constraints of our testbed, we found almost no detectable delay or loss of throughput, even with large filter policies.

We note that this is a limitation of our testing. In reality, while DiP actually performs as expected (a simple TCAM match is expected to take only constant time, and the limitation is the small size

of policy it can accommodate), DeeP4R *does* suffer from delays and costs which are not apparent from our evaluation. First, as packets take longer to process they continue to take up space in the switch buffers, which is further increased by the need to make a clone copy for each packet. But more importantly, *recirculation comes at a cost.* Each recirculation of the packet takes some time, and impacts the bandwidth of the pipeline; total bandwidth reduction depends on the number of times a packet must be recirculated (and this affects *all packets* passing through the switch).

The reason why in spite of these costs, DeeP4R apparently introduces *no* delay or throughput loss, is the very high performance of our switch – it is able to handle 100 Gbps at each port, so the maximum of 10 Gbps we could test with from our client-server setup did not stress the system. Our tests were sufficient for our goals (proving that the system works), but we have not tested its limits. For the present we conclude that the high performance of the switch easily compensates for the inefficiency of the algorithm, and it handily outperforms a server-based solution; but stress testing the system remains a problem for future research.

In addition to the immediate usefulness of our algorithms (DiP, DeeP4R) as in-path middleboxes that can perform URL filtering in multiple protocols, we note that they greatly lower the barrier to entry for network security under the control of network administrators themselves. Rather than relying on the black-box functionality of a large enterprise firewall, the admin can verify the packet processing functionality from our source code and modify it as needed. Thus, we hope that in addition to the intellectual merit of our work, we make a contribution to more transparent and accessible network security for enterprise, data center, and small networks.

Chapter 11

Concluding Remarks.

In this dissertation, we have designed and demonstrated DiP and DeeP4R, two new systems that show how Deep Packet Inspection can be performed in the programmable data plane. Both our systems succeed in their target task – filtering network traffic to blacklisted URLs – with excellent performance: in our tests, they perform DPI at line rate, without appreciably impacting the performance of other (non-blocked) traffic. In this concluding chapter, we mention the importance of each system, and how, moving forward, they can provide the foundation for future work.

Our first system, DPI-in-P4 (DiP), is a simple URL-filter in the dataplane. The main advantage of DiP is that it makes use of only standard P4 (as supported by the PISA platform), so our test implementation on a standard switch (Netberg Aurora 710) can be trivially ported to any P4-compliant packet processor. Thus, DiP proved for the first time that standard P4-compliant switches can perform limited DPI, without extern hardware (implemented using NetFPGA etc.), and without external help from firewalls. Indeed, DiP performs simple Deep Packet Inspection (i.e. URL detection) across multiple application-layer protocols, and scales to a substantial number of domains (1000) while outperforming the standard Linux firewall, Netfilter, by three to four orders of magnitude.

As it stands, our implementation of DiP is adequate for small network admins to blacklist e.g. malware domains, so it could be directly useful as a firewall for small business or campus networks. However, DiP is limited in its power, as it relies heavily on the observation (from Chapter 4 that practical traffic has packets with the URL field at a consistent location. It is possible that as DiP runs on programmable switches, it can be updated with extensions and patches as necessary (i.e. if browsers decide to change the position of URL in a packet, the code could be easily updated to

adapt the firewall to its new requirements), but this would lead to a constant game of cat-and-mouse between the user and the network admin. Accordingly, we go on to build a more robust system, DeeP4R.

Our second system, DeeP4R, shows a general method how DPI can be performed in the programmable data plane – making it possible to build an *application-layer* firewall on SDN switches. As expected for a data plane program, it is able to filter thousands of URLs at line rate, without loss of performance for non-blocked traffic. Further, DeeP4R is also built using only standard P4, and is easily portable to any standard switch. However, as DeeP4R works by recirculating packets, we can predict that at a large enough scale it may run into performance issues. Our results show it works much better than standard netfilter firewall on a server, but it may still have problems with line-rate filtering in an ISP. Perhaps in future, it may be possible to build a hybrid approach (using DiP to rapidly screen packets, then send only packets of interest to be deeply analyzed using DeeP4R).

In addition to proving our hypothesis – that Deep Packet Inspection can indeed be performed in the data plane – we have also made our implementation code publicly available at [8]. We expect that our work, as presented in this dissertation, will draw attention to the fact that P4 can provide quite sophisticated filtering and is not limited to screening packets by their header fields. It also raises several new open problems, such as building a DiP-DeeP4R hybrid, scaling to very large Access Control Lists (eg. the Great Firewall of China has 350 k blocked sites), and on-path rather than in-path filtering, all of which will lay the foundation for our group's future work. In future, such a data-plane solution may become a very useful option for data center or enterprise network security.

Bibliography

[1] 4 Pipe Tofino ASIC. https://github.com/barefootnetworks/Open-Tofino.

[2] 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.

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

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

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

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

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

[8] Open Source. https://zenodo.org/badge/latestdoi/574774753.

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

[10] P4 behavioral model. https://github.com/p4lang/behavioral-model. Accessed: 2022-05-25.

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

[12] Statcounter. https://gs.statcounter.com/.

[13] Hrishikesh B Acharya and Mohamed G Gouda. Firewall verification and redundancy checking are equivalent. In *2011 Proceedings IEEE INFOCOM*, pages 2123–2128. IEEE, 2011.

[14] Dirk Achenbach, Jörn Müller-Quade, and Jochen Rill. Universally composable firewall architectures using trusted hardware. In *International Conference on Cryptography and Information Security in the Balkans*, pages 57–74. Springer, 2014.

[15] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Network anti-spoofing with sdn data plane. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[16] Deepa Balagopal and X Agnise Kala Rani. Netwatch: Empowering software-defined network switches for packet filtering. In *2015 International conference on applied and theoretical computing and communication technology (iCATccT)*, pages 837–840. IEEE, 2015.

[17] Andrea Bianco, Paolo Giaccone, Seyedaidin Kelki, Nicolas Mejia Campos, Stefano Traverso, and Tianzhu Zhang. On-the-fly traffic classification and control with a stateful sdn approach. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[18] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. Pisa—a platform and programming language independent interface for search algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 494–508. Springer, 2003.

[19] Mihai Budiu and Chris Dodd. The p416 programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, 2017.

[20] Jiamin Cao, Jun Bi, Yu Zhou, and Cheng Zhang. Cofilter: A high-performance switch-assisted stateful packet filter. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 9–11, 2018.

[21] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. Heartbleed 101. *IEEE security & privacy*, 12(4):63–67, 2014.

[22] Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. Pvpp: A programmable vector packet processor. In *Proceedings of the Symposium on SDN Research*, pages 197–198, 2017.

[23] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. Appswitch: Application-layer load balancing within a software switch. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 64–70, 2017.

[24] Russ Cox. Regular expression matching can be simple and fast, 2007.

[25] Rakesh Datta, Sean Choi, Anurag Chowdhary, and Younghee Park. P4guard: Designing p4 based firewall. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2018.

[26] Damu Ding, Marco Savi, Federico Pederzolli, Mauro Campanella, and Domenico Siracusa. In-network volumetric ddos victim identification using programmable commodity switches. *IEEE Transactions on Network and Service Management*, 18(2):1191–1202, 2021.

[27] Colin Dixon, Arvind Krishnamurthy, and Thomas E Anderson. An end to the middle. In *HotOS*, volume 9, pages 2–2, 2009.

[28] Trinh Viet Doan, Irina Tsareva, and Vaibhav Bajpai. Measuring dns over tls from the edge: Adoption, reliability, and response times. In *International Conference on Passive and Active Network Measurement*, pages 192–209. Springer, 2021.

[29] Garegin Grigoryan and Yaoqing Liu. Lamp: Prompt layer 7 attack mitigation with programmable data planes. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018.

[30] Sahil Gupta, Devashish Gosain, Garegin Grigoryan, Minseok Kwon, and HB Acharya. Simple deep packet inspection with p4. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2021.

[31] Sahil Gupta, Devashish Gosain, Garegin Grigoryan, Minseok Kwon, and HB Acharya. Simple deep packet inspection with p4. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–2, 2021.

[32] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *arXiv preprint arXiv:2101.10632*, 2021.

[33] Hongxin Hu, Gail-Joon Ahn, Wonkyu Han, and Ziming Zhao. Towards a reliable {SDN} firewall. In *Open Networking Summit 2014 (ONS 2014)*, 2014.

[34] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan 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*, pages 336–350, 2020.

[35] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 21–28, 2019.

[36] Avinash Kak. Computer and network security lecture notes. `https://engineering.purdue.edu/kak/compsec/`.

[37] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable {In-Network} security for context-aware {BYOD} policies. In *29th USENIX Security Symposium*, pages 595–612, 2020.

[38] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[39] Xin Zhe Khooi, Levente Csikor, Dinil Mon Divakaran, and Min Suk Kang. Dida: Distributed in-network defense architecture against amplified reflection ddos attacks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 277–281. IEEE, 2020.

[40] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, 2015.

[41] Jason Kim, Hyojoon Kim, and Jennifer Rexford. Analyzing traffic by domain name in the data plane. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 1–12, 2021.

[42] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[43] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4cep: Towards in-network complex event processing. In *2018 Morning Workshop on In-Network Computing*, pages 33–38, 2018.

[44] Mário Kuka, Kamil Vojanec, Jan Kučera, and Pavel Benáček. Accelerated ddos attacks mitigation using programmable data plane. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–3. IEEE, 2019.

[45] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan 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*, pages 183–196, 2017.

[46] Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. Offloading real-time ddos attack detection to programmable data planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27. IEEE, 2019.

[47] Alex X Liu. *Firewall design and analysis*. World Scientific, 2010.

[48] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. {NetHide}: Secure and practical network topology obfuscation. In *27th USENIX Security Symposium*, pages 693–709, 2018.

[49] Yu Mi and An Wang. Ml-pushback: Machine learning based pushback defense against ddos. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, pages 80–81, 2019.

[50] MG Mihalos, SI Nalmpantis, and K Ovaliadis. Design and implementation of firewall security policies using linux iptables. *Journal of Engineering Science & Technology Review*, 12(1), 2019.

[51] Francesco Musumeci, Valentina Ionata, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. Machine-learning-assisted ddos attack detection with p4 language. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2020.

[52] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

[53] Francesco Paolucci, Filippo Cugini, and Piero Castoldi. P4-based multi-layer traffic engineering encompassing cyber security. In *Optical Fiber Communication Conference*, pages M4A–5. Optical Society of America, 2018.

[54] Ruben Ricart-Sanchez, Pedro Malagon, Jose M Alcaraz-Calero, and Qi Wang. Hardware-accelerated firewall for 5g mobile networks. In *26th IEEE International Conference on Network Protocols (ICNP)*, pages 446–447. IEEE, 2018.

[55] Ruben Ricart-Sanchez, Pedro Malagon, Jose M Alcaraz-Calero, and Qi Wang. Netfpga-based firewall solution for 5g multi-tenant architectures. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 132–136. IEEE, 2019.

[56] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[57] Gao Shang, Peng Zhe, Xiao Bin, Hu Aiqun, and Ren Kui. Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[58] Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. Proof-carrying network code. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1115–1129, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Péter Vörös and Attila Kiss. Security middleware programming using p4. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 277–287. Springer, 2016.

[60] Jackson Woodruff, Murali Ramanujam, and Noa Zilberman. P4dns: In-network dns. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6, 2019.

[61] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. Where the light gets in: Analyzing web censorship mechanisms in india. In *Proceedings of the Internet Measurement Conference 2018*, pages 252–264, 2018.

[62] Kaiyuan Zhang, Danyang Zhuo, and Arvind 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*, pages 283–295, 2020.

[63] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.