Security Applications in P4: Implementation and Lessons Learned

Ali Mazloum, Ali AlSabeh, Elie Kfoury, Jorge Crichigno
College of Engineering and Computing, University of South Carolina, Columbia, U.S.A.
amazloum@email.sc.edu, aalsabeh@email.sc.edu, ekfoury@email.sc.edu, jcrichigno@cec.sc.edu

***Abstract*—The data plane, which used to provide a limited and fixed set of operations in legacy networking devices, is now programmable. The de-facto language to program the data plane is Programming Protocol-independent Packet Processors (P4). After compiling a P4 program, the resulting binary is loaded into the Application Specific Integrated Circuit (ASIC). The ASIC processes the packets based on the logic defined by the P4 program. The flexibility and granularity offered by programmable data plane devices allowed many security applications to be offloaded to the data plane. Thus, P4 and programmable devices allow the security applications to run on the hardware while sustaining the software's flexibility, which enhances their performance. However, developing a P4 program is not straightforward.**

The complexity associated with developing P4 applications has often been an obstacle for researchers. They mainly follow a trial-and-error approach to compile and fit their program into the ASIC. This paper tackles the issue by providing a comprehensive guide to the process of designing P4 security applications. It goes beyond theory and delves into practical implementation by showcasing the creation of several security applications on P4 programmable switches and sharing the P4 source code of these applications. In this paper, the authors will discuss the lessons they learned from implementing multiple security applications on programmable switches using P4, providing the reader with guidelines and insightful considerations.

***Index Terms*—P4, Programmable Data Planes (PDP), Security Applications, DNS DPI, Stateful Packet Filter, DDoS.**

I. Introduction

Traditional networking devices (e.g., routers and switches) implement fixed functionalities hard-coded by the vendors. The lack of flexibility and programmability of these devices has limited the innovation and the immediate response required by the ever-growing networking industry. For instance, in the presence of a zero-day exploit attacking the network infrastructure, traditional networking devices cannot be programmed as a defense strategy, thus, they are deemed ineffective [1]. Such limitations led to the emergence of the Software-Defined Networking (SDN) paradigm [2]. SDN reduces the complexity of the network and enables innovation on the control plane at the speed of software development. However, SDN is constrained by the OpenFlow specifications and the fixed-function data plane [3]. As a result, attack mitigation has either been offloaded to the control plane or to third-party middleboxes, which cannot operate at the speed of the data plane [4, 5]. As the scale of some attacks, such as Denial of Service (DoS), reached a peak of Terabits per second (Tbps), software-based defenses lagged behind [1].

An alternative and widely used approach to keep up with the increasing scale of attacks is the use of traffic scrubbing centers [6]. Scrubbing centers deploy proprietary hardware appliances to achieve high-performing defenses. Despite their effectiveness, scrubbing centers have some limitations. Cost can be a significant factor, as implementing and maintaining scrubbing center infrastructure can be expensive. Scalability can be an issue as well, with large-scale Distributed DoS (DDoS) attacks potentially overwhelming the center's resources, leading to partial or ineffective mitigation [7].

Programmable data plane (PDP) has emerged as a cost-effective solution that combines the flexibility of software-based approaches and the efficiency and performance of hardware-based approaches. PDP is the natural evolution of SDN, which allows network administrators to perform custom packet processing at the data plane level. P4, which stands for Programming Protocol-independent Packet Processors, is the de-facto language for defining the forwarding behavior of PDPs. It allows the processing behavior of the devices to be adjusted by programmers based on their needs. The flexibility offered by PDPs does not entail any processing overhead. PDP devices can apply the logic described in a compiled P4 program at line rate with Tbps processing speed. While programming a PDP, the programmer describes protocols and features in the Application Specific Integrated Circuit (ASIC). However, developing a P4 program is not straightforward. Developers are rarely sure if their program can fit into the ASIC, and consequently, they follow a trial-and-error approach to compile the program [8]. In P4, developers should account for the low-level hardware limitations when writing the programs.

This paper discusses the process of writing security programs in P4 and analyzes how different decisions made by the programmer are reflected in the resource usage of the programmable switches. The paper also describes the lessons learned from implementing the applications, providing the reader with guidelines and insightful considerations. The security applications discussed are stateful packet filters, DDoS defenses, DNS deep packet inspection (DPI), anti-spoofing

This work was supported by the U.S. National Science Foundation, Office of Advanced Cyberinfrastructure, Award #2403360 and Award #2346726.

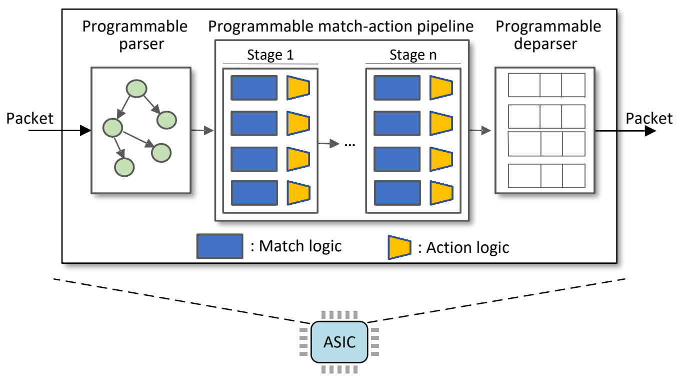


Fig. 1: The Protocol-Independent Switch Architecture (PISA).

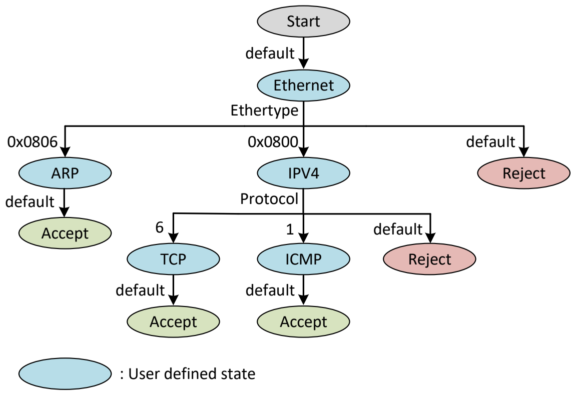


Fig. 2: Example of a parser that accepts packets with Ethertype 0x0806 and 0x0800, packets with Protocol 6 and 1, and rejects all other packets.

defenses, and cryptographic functions.

A. *Contributions*

The literature is rich with P4-related works that either focus on describing the technology and its corresponding work [1, 8–10] or on utilizing the technology to produce innovative systems [11–18]. However, to the best of the authors' knowledge, no previous work focuses on the methodology of writing P4 applications as the complexity of developing P4 applications is one of the main barriers for researchers. To this end, this paper aims to explain the methodology of designing P4 security applications, illustrate the effects of different implementation approaches on the resource utilization of PDP devices, and present the main considerations required to integrate multiple security applications in a single P4 program. The provided implementations are based on Tofino Native Architecture (TNA) and target Tofino programmable switches. All related source codes are released at Github [19].

B. *Paper Organization*

The paper is organized as follows: Section II provides background information on PDPs; Section III discusses implementing stateful packet filters in P4; Section IV discusses implementing DDoS defense in P4; Section V discusses implementing DNS DPI in P4; Section VI discusses implementing spoofing defense in P4; Section VII discusses implementing cryptography in P4; Section VIII summarizes the learned lessons; Section IX discusses future directions, and Section X concludes the paper.

II. Background

A. *Data Plane Programmability and PISA Architecture*

Data plane programmability allows the user to define the processing behavior of the data plane by implementing custom algorithms. Programmable switches are networking devices with a programmable data plane. Besides enabling the user to deploy the standard forwarding functionalities, programmable switches can provide per-packet visibility, custom packet processing at line rate, stateful memory elements, and APIs to interact with the data plane during the run time, among others.

The Protocol Independent Switch Architecture (PISA) is a widely used architecture for programmable switches. As depicted in Fig. 1, PISA includes three programmable blocks, a programmable parser, a programmable match-action pipeline, and a programmable deparser. The parser defines the headers and parses them. Headers are defined based on custom or standard protocols. The programmable match-action pipeline executes operations on packet headers and intermediate results. The pipeline comprises multiple stages. Each stage contains memory blocks (tables and registers) and Arithmetic Logic Units (ALUs). This arrangement enables parallel lookups and actions, ensuring efficient packet processing. The stages are organized sequentially to manage data dependencies (i.e., when the results of one stage are required as input to another stage). After processing the parsed headers, the deparser assembles and serializes them for transmission.

B. *Programmable Switches*

The flexibility offered by programmable switches allows multiple security applications to be offloaded to the data plane [1]. The five security applications that will be discussed in this paper are stateful packet filters, DDoS mitigation systems, DNS DPI, anti-spoofing defenses, and cryptographic functions. The main components of the programmable switches that allow these applications to be offloaded to the data plane are the programmable parser, the match-action tables, and the registers.

1) Programmable parser: When developing a P4 program, the programmer defines the set of headers to be extracted and processed by the programmable switch. The programmable parser is the component responsible for extracting the headers from the incoming packets. It operates as a finite state machine with an explicit start state, two ending states (Accept and Reject), and the user-defined states in between. Transitioning between states can be either conditional or unconditional. In conditional transitioning, the parser checks if the headers of the packet satisfy a given condition (e.g., check if the Ethertype is 0x0806). If the condition is satisfied, the parser transitions to the state associated with the condition. On the other hand, an

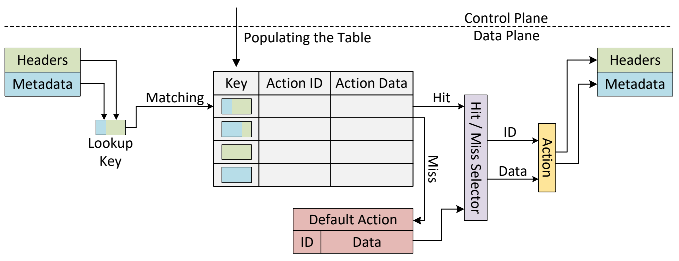


Fig. 3: P4 match-action table. The lookup key is created from headers and metadata. It is matched against keys populated by the control plane. If a hit occurs, the action of the corresponding entry will be applied. If no hit occurs, the default action is applied.

unconditional (also known as default) transition occurs when no conditions are defined or satisfied.

Fig. 2 shows an example of a parser. The first state is the Start state. The only two possible end states are Accept and Reject. The user-defined states are Ethernet, ARP, IP, TCP, and IMCP. In this example, each state represents a different protocol. The parser transitions by default from the Start state to the Ethernet state. At the Ethernet state, the parser applies two conditions on the Ethertype field of the Ethernet header. If the packet is ARP or IP, then one of the two conditions is satisfied, and the parser transitions to the corresponding state. Otherwise, the parser transitions by default to the Reject state. Following the same logic from the ARP and IP states, the packet is either accepted and the defined headers are extracted, or it is rejected.

2) Match-action tables: Match-action tables are the second programmable block in the PISA architecture. As depicted in Fig. 3, each table has multiple entries with a key to match on, an action performed when a packet matches the entry, and action data (which might be empty). The key can be a single field (e.g., source IP address) or multiple fields (e.g., source and destination IP addresses). The components of the key and the matching type should be specified during the implementation of the P4 program and cannot be altered during the run time. The switch builds a lookup key for the incoming packets from their metadata and header fields and then uses the created key to search the match-action tables. There are three main matching types: exact, longest prefix, and ternary. In the exact matching type, the lookup key should match exactly a key in the tables for the corresponding action to be performed. In the longest prefix matching (LPM) type, the entry with the longest matching prefix is selected. In the ternary matching type, the programmer defines what portion of the key should the incoming packets match on. For example, if the switch should forward only the packets with source IPv4 starting with 172 and ending with 172 (i.e., 172.*.*.172), then the ternary matching type should be used. Because a packet might match multiple ternary entries, the control plane should specify the priority of each entry.

3) Registers: Registers are stateful memory elements where each register entry has an index and a value. The number of entries (i.e., register length) for each register and the size of each entry (register width) should be specified in the P4 program during the implementation. The defined registers' length and width cannot be altered during the run time. Hash functions are typically used to create the indexes of the flows. A flow is a group of packets that share some common header fields. The fields used to classify packets into flows (e.g., the 5-tuple: source and destination IP address, source and destination ports, and protocol) are hashed and then used to map all the packets of a flow to the same register entry. To avoid race conditions, P4 does not allow register entries to be accessed by two different stages simultaneously. Furthermore, the value of a register entry cannot be modified more than once in each pipeline pass.

III. Deploying Stateful Packet Filters in P4

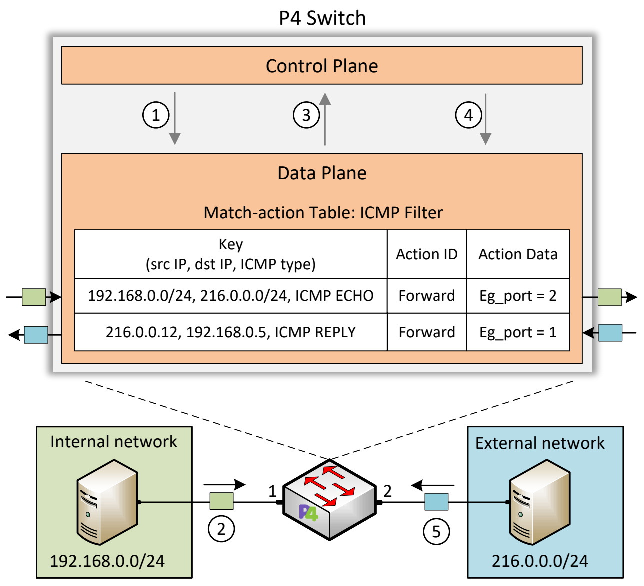
Stateful packet filters divide connections into sessions [20]. Each session has a client-to-server (c2s) flow and a server-to-client (s2c) flow. The endpoint initiating the connection is the client, and the server is the other endpoint. Security rules are defined for c2s flows only [20]. Based on the state of a c2s flow, the corresponding s2c flow is either denied or forwarded.

There are two methods for implementing stateful packet filters in P4. The first method is to implement the packet filter using match-action tables. The second method is to use both match-action tables and stateful registers. While both implementations can provide the same functionalities, each has its advantages. The main advantage of the first implementation is that the forwarding rules are populated by the control plane at runtime, allowing the administrators to have full control over their networks. However, this implementation is not recommended if the latency added by the control plane cannot be tolerated. In this case, the second implementation should be considered, as it allows the stateful packet filter to update its rules at the data plane level.

A. Match-action Tables-based Implementation

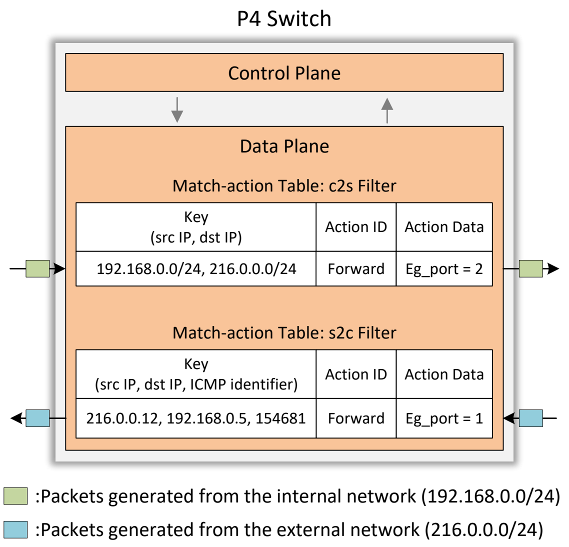
Implementing a stateful packet filter using match-action tables requires the administrator to define security rules for the c2s and the s2c flows. However, the administrator only configure the rules for the c2s flows. The control plane then automatically populates the rules for the s2c flows. One way for the data plane to interact with the control plane is through digests. Digests are packets used by the data plane to report events to the control plane. When the data plane receives the first packet from a c2s flow, it checks the packet against its forwarding rules. If the packet is allowed, the data plane notifies the control plane that a new c2s flow is received. After that, the control plane pushes a new rule for the s2c flow corresponding to the received c2s flow.

Fig. 4 depicts the scenario where only ping packets originating from the internal network are allowed. The data plane defines the ICMP Filter table to play the packet filter role. The administrator populates the rules to accept c2s flows, i.e., accept ICMP ECHO packets originating from the internal network (1). When the switch receives the ECHO packet from 192.168.0.5 (internal network) destined for 216.0.0.12



Key (src IP, dst IP, ICMP type)	Action ID	Action Data
192.168.0.0/24, 216.0.0.0/24, ICMP ECHO	Forward	Eg_port = 2
216.0.0.12, 192.168.0.5, ICMP REPLY	Forward	Eg_port = 1

Fig. 4: P4-based stateful packet filter using a match-action table. The ICMP Filter table in the data plane plays the packet filter role. (1) A security rule is added to accept ICMP ECHO packets originating from the internal network. (2) A device from the internal network (192.168.0.5) sends an ICMP ECHO packet to the external network (216.0.0.12). (3) The data plane forwards the packet through egress port 2 because it matches the rule in the ICMP Filter table with the action Forward. The switch then notifies the control plane that a new flow is active. (4) The control plane pushes a new rule to accept the ICMP REPLY packet from the external network. (5) The switch forwards the ICMP REPLY packet from the external network through egress port 1.


P4 Switch
Control Plane
Data Plane
Match-action Table: c2s Filter
Match-action Table: s2c Filter
🟩 :Packets generated from the internal network (192.168.0.0/24)
🟦 :Packets generated from the external network (216.0.0.0/24)

Key (src IP, dst IP)	Action ID	Action Data
192.168.0.0/24, 216.0.0.0/24	Forward	Eg_port = 2

Key (src IP, dst IP, ICMP identifier)	Action ID	Action Data
216.0.0.12, 192.168.0.5, 154681	Forward	Eg_port = 1

Fig. 5: P4-based stateful packet filter using multiple match-action tables. The c2s Filter table maintains the rules for c2s flows and is typically populated at the beginning of the run time. The s2c Filter table is populated during the run time based on the flows accepted by the c2s Filter table.

(external network) (2), it forwards it through port 2. After that, the data plane of the switch notifies its control plane about the new flow (3). The control plane then adds a new rule to accept the s2c flow, i.e., the REPLY packet from 216.0.0.12 to 192.168.0.5 (4). When the external network sends the REPLY packet, the packet filter forwards it based on the rule added by the control plane (5).

Security policies have different matching criteria depending on the application being deployed. For example, deploying a packet filter that controls TCP traffic might match packets using the 5-tuple. On the other hand, a packet filter that controls ICMP traffic might only use the source and destination IP addresses and the ICMP type. A PDP cannot deploy a stateful packet filter encompassing various policies and protocols using a single match-action table. The number of tables the PDP requires depends on the different deployed matching keys (e.g., having one match-action table for ICMP traffic and another table for TCP traffic or even multiple tables for each).

Defining multiple tables provides more flexibility and might enhance the packet filter's security and performance. For instance, a more secure approach than the one presented in Fig. 4 is to use the ICMP identifier field of the ICMP protocol to map REPLY packets to their corresponding ECHO packets. The ICMP header has an identifier field, which is a randomly generated number that uniquely identifies the related ICMP ECHO and REPLY packets. As shown in Fig. 5, this approach requires defining a separate match-action table for the s2c flows and using the ICMP identifier as an additional matching field beside the source and destination IP. An incoming ECHO packet is matched against the c2s Filter table; if it hits an entry, its identifier field will be sent to the control plane, which adds a new rule for the s2c Filter table. An incoming REPLY packet is matched against the s2c Filter table; if it hits an entry, the packet will be forwarded, and the corresponding entry will be deleted (if no other packet with the same identifier is expected to arrive).

Listing 1 shows the TNA-based P4 code that implements a stateful packet filter. The packet filter operates on ICMP and TCP traffic. For ICMP traffic, the table *c2s_Filter_ICMP* matches the c2s flows on their source and destination IP addresses. The matching types used are LPM and ternary. The reason for not using two LPMs is that TNA does not support having two LPM matching types for a single key. The action *notify_control_plane_ICMP_c2s* sends a notification (i.e., digest) message to the control plane. The notification includes the source IP, destination IP, and the identifier of the ICMP packet. After receiving the notification, the control plane adds a rule to the *s2c_Filter_ICMP* table. The key of the *s2c_Filter_ICMP* table matches on the exact values of the source IP, destination IP, and ICMP identifier. If a match occurs, the data plane notifies the control plane which removes the matched rule if no other ICMP REPLY packets with the same identifier are expected.

The same logic of populating the s2c rule applies to TCP traffic but with different keys. Note that the *s2c_Filter_TCP* table has no action to notify the control plane when a match occurs. Unlike ICMP, TCP s2c flows can have an arbitrary number of packets, and consequently, an s2c rule cannot be removed after a specific number of hits (i.e., cannot detect the termination of the s2c flow by monitoring the number of hits).

Listing 1: TNA-based P4 code to implement stateful filters using the tables-based approach.

```
/************************ICMP*************************/
table c2s_Filter_ICMP {
    key = {
        hdr.ipv4.src_addr: lpm;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {
        notify_control_plane_ICMP_c2s;
        drop;
    }
    size = 10000;
}

table s2c_Filter_ICMP {
    key = {
        hdr.ipv4.src_addr: exact;
        hdr.ipv4.dst_addr: exact;
        hdr.icmp.identifier:exact;
    }
    actions = {
        notify_control_plane_ICMP_s2c;
        drop;
    }
    size = 250000;
}

/**********************End ICMP**********************/

/************************TCP*************************/
table c2s_Filter_TCP {
    key = {
        hdr.ipv4.src_addr: lpm;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {
        notify_control_plane_TCP_c2s;
        drop;
    }
    size = 10000;
}

table s2c_Filter_TCP {
    key = {
        hdr.ipv4.src_addr: exact;
        hdr.ipv4.dst_addr: exact;
        hdr.tcp.src_port: exact;
        hdr.tcp.dst_port: exact;
        hdr.ipv4.protocol: exact;
    }
    actions = {
        tcp_s2c;
        drop;
    }
    size = 250000;
    idle_timeout = true;
}

/**********************End TCP**********************/
```

To detect the termination of a TCP flow, the switch can monitor FIN and RST packets. Either the client or the server sends a FIN packet to the other communicating end to terminate a TCP session. After receiving the FIN packet, a handshake occurs between the communicating entities to terminate the connection. On the other hand, by sending an RST packet, a TCP endpoint indicates that it will not accept more data. In an ideal scenario, monitoring FIN and RST packets allows the switch to accurately monitor the termination of the flows. However, in a practical setting, hosts might fail to send a termination signal (e.g., FIN packet), preventing the programmable switch from detecting the termination of the flow [21].

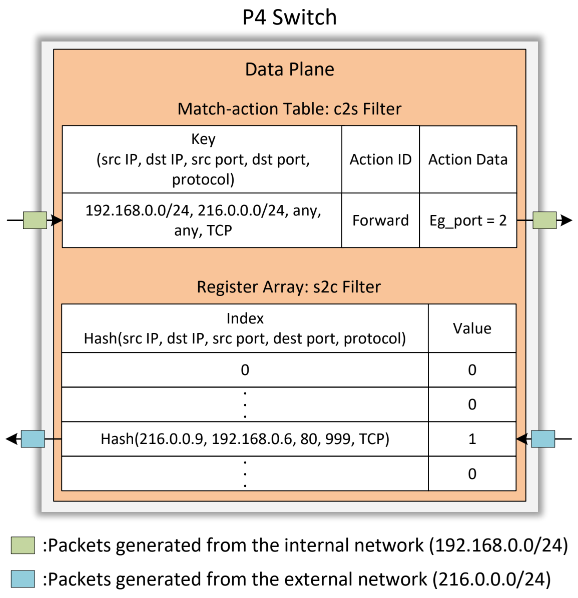

P4 Switch
Data Plane
Match-action Table: c2s Filter
Key (src IP, dst IP, src port, dst port, protocol)
Action ID
Action Data
192.168.0.0/24, 216.0.0.0/24, any, any, TCP
Forward
Eg_port = 2
Register Array: s2c Filter
Index Hash(src IP, dst IP, src port, dest port, protocol)
Value
0
0
0
Hash(216.0.0.9, 192.168.0.6, 80, 999, TCP)
1
0
🟩 :Packets generated from the internal network (192.168.0.0/24)
🟦 :Packets generated from the external network (216.0.0.0/24)

Fig. 6: P4-based stateful packet filter using register array. A match-action table is needed to apply the rules on c2s flows. The state of the accepted flows is then maintained by the s2c Filter register.

An alternative approach to detect the termination of the sessions is through setting idle timeouts [21]. The idle timeout is a feature supported by programmable switches where the data plane notifies the control plane when no hits occur on a table entry for a specific duration. The timeout duration and the action to be performed on idle entries are configured from the control plane. The provided code in Listing 1 utilizes the idle timeout method to remove s2c rules related to terminated s2c flows.

B. Stateful Registers-based Implementation

By utilizing registers, the programmable switch can maintain the state of the flows without the intervention of the control plane at run time. The c2s flows are controlled using match-action tables, where the rules are populated once based on the policy. The c2s rules are only modified if the policy changes. On the other hand, the s2c flows (the flows that required the control plane to update the match-action tables at run time in the previous implementation) are controlled using registers. When a new c2s flow is received, the switch checks if the flow should be accepted based on the policy (i.e., using an entry in the c2s table). If the flow is accepted, the switch calculates the hash of the packet fields that identify the flow and uses them to create an entry for the s2c flow. When a packet belonging to an s2c flow arrives at the switch, the switch calculates the hash of the packet fields. If the register entry index by the hash value corresponds to an active s2c flow, the packet is forwarded. Otherwise, the packet is dropped.

The workflow of a packet filter that filters TCP traffic using registers is depicted in Fig. 6. There are two networks:

TABLE I: Comparison between tables-based and registers-based implementations of stateful packet filters in P4.

Criteria	Tables-based	Registers-based	Notes
Resource usage	High	Low	It is more practical and feasible to match on hashes in registers-based implementations than on the exact value
Visibility	High	Low	Tables-based implementation notifies the control plane on each table hit, providing detailed logs of the traffic
Scalability	Low	High	As the number of flows increases, the number of notifications received by the control plane from the tables-based implementation increases
Inference latency	Around 600 *ns*	Around 600 *ns*	If a P4 application compiles, it is guaranteed to have line-rate packet processing speed [8]
Latency of adding/updating c2s rules	Around 1 *ms*	Around 1 *ms*	The control plane is responsible for managing the c2s rules for both implementations
Latency of adding s2c rules	Around 1 *ms*	Nanosecond scale	Tables-based implementation requires the intervention of the control plane to add s2c rules
Latency of removing s2c rules	Nanosecond scale	Nanosecond scale	The data plane is responsible for removing s2c rules for both implementations

an internal network and an external network. The deployed policy only allows the TCP connections originated from the internal network and destined to the external network. The control plane populates the data plane by inserting a rule at the c2s Filter table. When the internal network initiates a TCP connection, the programmable switch forwards the corresponding SYN packet because it is accepted by the policy. The switch then creates an entry in the register s2c Filter that will accept the corresponding TCP packets from the external network. To create the entry, the hash of the 5-tuple is calculated and used as an index of the register. The value of the entry at the calculated hash is set to the destination port of the flow. Using the destination port reduces the number of false positives. A false positive occurs when a new flow has the same hash as an active flow (i.e., hash collision). In utilizing the destination port of the flows, a false positive occurs only if two flows have the same hash and the same destination port.

Besides creating new entries for the s2c flows, the data plane is responsible for removing the entries when flows terminate. To detect flow termination without the intervention of the control plane, the data plane should maintain a timestamp for each register entry (i.e., s2c rule) representing the time when the last hit occurred. When an incoming packet maps to an s2c rule, the switch calculates the time difference between the arrival of the packet and the last time the rule was hit. If the difference is larger than a predefined idle timeout, the new packet is dropped, and the s2c rule is deleted.

Listing 2: TNA-based P4 code to implement stateful filters using the registers-based approach.

```
table c2s_Filter_TCP {
    key = {
        hdr.ipv4.src_addr: lpm;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {
        NoAction;
    }
    size = 10000;
}

Register<bit<16>, _>(250000) s2c_Filter;
RegisterAction<bit<16>, _, bit<16>>(s2c_Filter)
     update_tcp_dst_port = {
    void apply(inout bit<16> register_data,out bit<16>
         result) {
        if(register_data == 0){
        register_data = hdr.tcp.dst_port;
        result=1;
        }
        else{
            result = 0;
        }

    }
};
action exec_update_tcp_dst_port(){
    meta.already_occupied_tcp = update_tcp_dst_port.execute(
         meta.flow_id);
}

RegisterAction<bit<16>, _, bit<16>>(s2c_Filter)
     check_tcp_dst_port = {
    void apply(inout bit<16> register_data, out bit<16>
         result) {
        if(register_data == hdr.tcp.src_port){
            result =1;
        }
        else{
            result = 0;
        }
    }
};
action exec_check_tcp_port(){
    meta.allow_REPLY = (bit<16>)check_tcp_dst_port.execute(
         meta.rev_flow_id);
}
```

Listing. 2 shows the TNA-based P4 code of the scenario represented in Fig. 6. TCP traffic has a table for c2s rules and a register for s2c rules. The tables can store and apply 10,000 different c2s rules (i.e., policies). The register can maintain the state of 250,000 active flows simultaneously. Each register can have one or more register actions (denoted by *RegisterAction* in the P4 code). Each register action applies a set of operations on the corresponding register at a specific index. The index is a required argument when executing the register action. A good practice is to define an action that executes the register action. For instance, the action *exec_update_tcp_dst_port* executes the register action *update_tcp_dst_port* at the index *meta.flow_id*. This index should be calculated by a hash function prior to executing the register action (refer to [19] for the full P4 program). *update_tcp_dst_port* is associated with the register *s2c_Filter*. This means that all the operations defined by *update_tcp_dst_port* are performed on *s2c_Filter*.

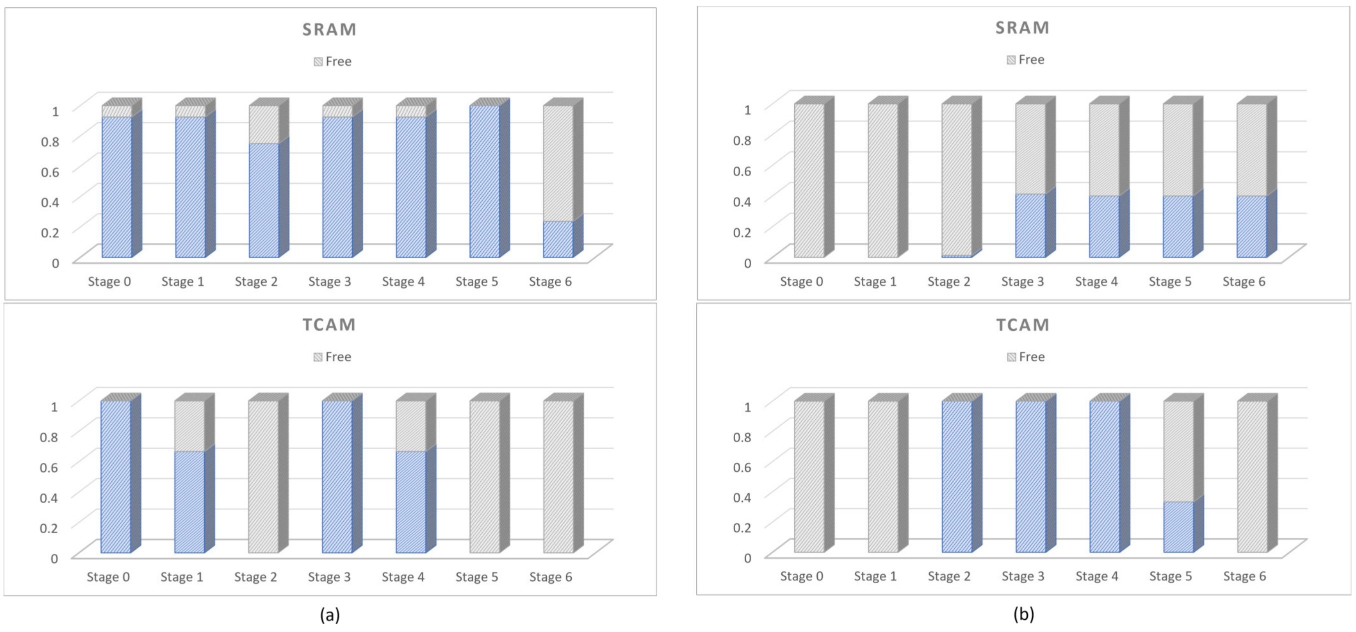


Fig. 7: SRAM and TCAM resources used by tables-based (a) and registers-based (b) stateful packet filters monitoring ICMP and TCP protocols on a P4 programmable switch.

C. Comparing between Tables-based and Registers-based Implementations

Table I compares between the tables-based and registers-based implementations of stateful packet filter in P4. The first major difference between the two implementation methods is the overhead added by the control plane. In the tables-based implementation, the control plane is responsible for maintaining the state of the flows (i.e., adding and removing table entries for the s2c flows). Experiments show that the control plane requires around one millisecond (ms) to add a new rule to the data plane. All s2c packets received before the control plane pushes the new rule are dropped by the data plane. On the other hand, the registers-based implementation does not require the intervention of the control plane, allowing it to add s2c rules at line rate.

Besides, another difference is in determining the time to remove the s2c rules. The control plane sets an *idle-timeout* threshold on the table entries. An entry is removed if no packet hits it for the pre-defined duration. On the other hand, to implement a self-expiry register, a timestamp is attached to each register entry. When a new packet maps to an entry, the time difference between the stored timestamp and the packet's timestamp is calculated and compared to a timeout threshold. If the time difference is larger than the timeout threshold, the entry is deleted, and the packet is dropped. Otherwise, the packet is forwarded, and the timestamp at that register entry is updated to the packet's timestamp. However, using the packet's timestamp (which is 48 bits long) is memory intensive. For this, only a portion of the packet's timestamp is used to implement the self-expiry functionality. Which portion to use is application-dependent and is generally defined by two factors. The first factor is the granularity of the timestamp (e.g., nanosecond, microsecond, etc.). The second factor is the maximum value the timestamp covers (e.g., 1 s, 200 ms, etc.). For instance, using the first 32 bits of the packet's timestamp provides nanosecond granularity and a maximum value of around 4.3 seconds. If only microsecond granularity is needed, then the first 10 bits are excluded (i.e., only 22 bits are used).

Furthermore, the other difference between the two implementations is in resource usage. Fig. 7 compares the resources used by the two implementation methods when deployed on a hardware switch. The switch supports 12 stages, but the two implemented packet filters utilize resources from the first seven stages only (stage 0 to stage 6), and consequently, only the resource usage of the first seven stages is reported. Each stage has dedicated resources of SRAM and TCAM. The compiler distributes the resources required by the P4 program on the stages. The two implementations support monitoring 250,000 flows. For the tables-based implementation, more than 80% of the SRAM resources of the first six stages and 30% of the SRAM of the last stage are utilized. The tables-based implementation also requires the TCAM of the first and fourth stages (stages 0 and 3), as well as around 70% of the TCAM resources of the second and fifth stages (stages 1 and 4).

For the registers-based implementation, only 40% of the SRAM resources in four different stages were utilized. Although both implementations can monitor the same number of flows (250,000), the tables-based implementation used 3.5 times more SRAM than the registers-based implementation. This difference in the utilized SRAM is mainly caused by the difference in the matching key sizes. In tables-based implementation, the 5-tuple is used to define flows, and consequently, the size of the matching key is the summation of all the fields (i.e., 96 bits). On the other hand, in registers-

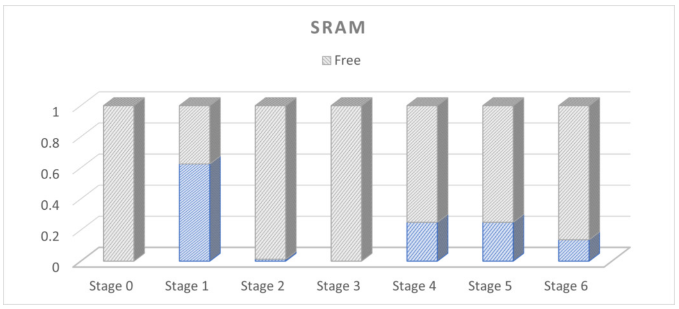


Fig. 8: SRAM resources used by the tables-based implementation with hashed matching keys.

based implementation, the hash of the fields is used to index the register entries. As shown by Fig. 8, the SRAM resources utilized by the tables-based implementation with hashed keys are equivalent to the amount of SRAM resources utilized by the registers-based implementation.

Hash size can vary depending on the number of flows to be monitored. Because the packet filter should maintain the state of 250,000 flows, the length of the hash is set to 18 bits. It is important to note that this significant saving in SRAM is at the cost of losing the state of some flows due to collisions. A hash collision often happens due to the limited output space of hash functions, which generate a fixed-size output. If minimizing the number of collisions is a priority, then the size of the hash can be adjusted according to the requirements. Increasing the hash size does not eliminate the probability of having hash collisions. To eliminate hash collisions, the exact values of the matching fields should be used.

Regarding the TCAM resources, both implementations use the same amount (the only difference is how the TCAM resources are distributed over the stages). Both implementations utilize match-action tables to set the rules for c2s flows, where the matching keys use LPM and ternary matching types. Because LPM and ternary matching types are used only for defining the c2s rules and the two implementations have the same number of rules, they utilize exactly the same amount of TCAM.

IV. Deploying DDoS Defense In P4

DoS attacks aim at exhausting a targeted machine, preventing it from replying to legitimate requests [1]. A DDoS attack is a large-scale DoS attack where multiple compromised machines (botnets) are utilized to flood the victim. While all DDoS attacks aim at preventing the victim's machine from serving legitimate requests, they cannot be mitigated using a single approach [7].

Different DDoS attacks utilize different protocols and require different defenses. To mitigate a specific attack, the P4 security application should monitor the protocols utilized by the attack (e.g., track the number of SYN packets and limit their rate to mitigate SYN flood). The programmer should dedicate memory resources for each type of defense. The more memory resources required per defense type, the lower the number of different attacks that can be mitigated by a single P4 application.

A main consideration while developing DDoS defense in P4 is whether the application should detect the attacker or just protect the victim [1]. If detecting the IP address or the subnet of the attacker is a priority, the P4 application should monitor the traffic coming from the external network based on the source IP. Because a DDoS attack can utilize a wide range of IP addresses using botnets, a significant portion of the memory resources will be allocated to track the statistics of the traffic. Consequently, a limited number of attack mitigation defenses can be implemented in a P4 application. On the other hand, by only focusing on protecting the victim machine, the P4 application can utilize all the resources to monitor the traffic by their destination IP address instead of their source IP address. Thus, the application can defend against a wide range of DDoS attacks [7].

Deploying a P4 application on a switch's ASIC provides a significant performance over a software-based DDoS defense [1]. However, limited on-chip resources and restrictive computational models are two main challenges in implementing DDoS defense in P4 programmable switches. Complex detection algorithms (e.g., puzzle for HTTP Flood) cannot be deployed in the data plane of the switches because of the computational models restrictions [7]. To overcome these challenges, a programmable switch can steer suspicious traffic to a scrub center where complex algorithms can be implemented.

The remainder of this section illustrates the P4 defense implementation of some widespread DDoS attacks:

SYN flood attack: In this attack, the victim is flooded with fabricated SYN packets from fake source IP addresses. The victim machine replies with a SYN-ACK packet for each received SYN packet but never receives back the ACK packet (i.e., the last stage in the TCP 3-way handshake). This leads to a significant difference between the number of SYN packets and the number of ACK packets received by the victim. The P4 switch can monitor this difference to detect the attack. There are two ways to count the number of SYN and ACK packets. The first way is per source IP address, which requires a number of register entries equivalent to the number of fabricated IP addresses, draining the memory resources of the switch if the attacker uses a massive number of different IP addresses. The other way is to count the number of SYN and ACK packets per destination IP address, which requires two register entries per a protected destination IP.

Another approach to detecting SYN flood attacks is monitoring the number of SYN packets only. In this approach, the network administrator defines an activate threshold that limits the number of SYN packets to be allowed per unit of time. If the threshold is exceeded, the firewall starts probabilistically dropping packets. As the gap between the activate threshold and the number of received SYN packets increases, the probability of dropping packets increases. To implement this method using P4, the data plane should periodically report the count of SYN packets. The control plane compares the count to the activate threshold and configures the data plane with the probability of dropping packets (which is 0% if the number is smaller than the threshold). Listing 3 shows the TNA-based P4 implementation of the registers used by this approach. The *syn_counts* register maintains the count of SYN packets for 65535 flows simultaneously. The *last_period_timestamp* register checks if a new report should be sent based on the difference between the timestamp of the current packet and the timestamp of the last report.

Listing 3: TNA-based P4 definition of the registers used by SYN flood defense.

```
Register<bit<32>, bit<16>>(65535) syn_counts;
RegisterAction<bit<32>, bit<16>, bit<32>>(syn_counts)
     update_syn_counts = {
    void apply(inout bit<32> register_data, out bit<32>
         result) {
        if (time_period_expired == 0) {
            register_data = register_data + 1;
        } else {
            result = register_data + 1;
            register_data = 0;
        }
    }
};

action apply_update_syn_counts() {
    meta.syn_count = update_syn_counts.execute(meta.
         flow_hash);
}

Register<bit<16>, bit<16>>(65535) last_period_timestamp;
RegisterAction<bit<16>, bit<16>, bit<1>>(
     last_period_timestamp) update_last_report_timestamp = {
    void apply(inout bit<16> register_data, out bit<1>
         result) {
        if (register_data == 0) {
            register_data = ig_intr_md.ingress_mac_tstamp
                 [44:29];
        } else {
            bit<16> tmp;
            tmp = ig_intr_md.ingress_mac_tstamp[44:29] -
                 register_data;
            if (tmp > 1) {
                register_data = ig_intr_md.
                     ingress_mac_tstamp[44:29];
                result = 1;
            } else {
                result = 0;
            }
        }
    }
};

action apply_update_last_report_timestamp() {
    time_period_expired = update_last_report_timestamp.
         execute(meta.flow_hash);
}
```

ICMP flood attack: In this attack, a target system or network is flooded with a high volume of ICMP packets, particularly ICMP Echo packets, often with forged source IP addresses. By doing so, the attacker overwhelms the target's available network bandwidth, CPU resources, and memory, disrupting legitimate network traffic and rendering the target unresponsive. To defend against this attack, the P4 switch should track the count of ICMP ECHO packets. The switch drops ICMP packets when their number exceeds a predefined threshold. As discussed before, the switch can track the count either by source or destination IP address of the packets. The drawback of using the source IP is the memory resources overhead. On the other hand, using the destination IP address to track the number of ICMP packets prevents the switch from detecting the source of the attack. Thus, the switch will not be able to distinguish between benign and fabricated ICMP ECHO packets when the threshold is exceeded. Upon detecting an ICMP flood, the switch will drop all ICMP packets, including the benign ones.

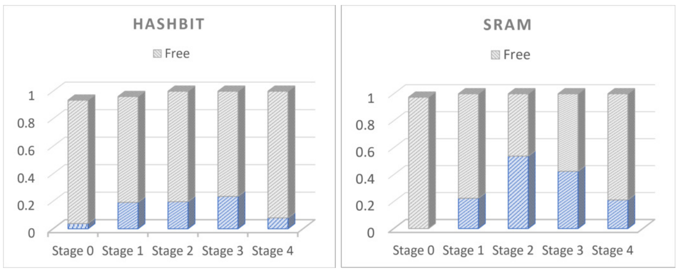


Fig. 9: Resources used by a DDoS application running on a P4 programmable switch.

DNS amplification attack: In this attack, the victim is flooded with DNS replies from multiple public DNS servers. To perform this attack, the attacker sends DNS queries with the IP of the victim to public DNS servers. The servers then flood the victim's machine by directing the replies to it. To mitigate this attack, the P4 switch should drop all DNS replies that do not belong to DNS queries requested by the victim. The switch should store the IP addresses of the machines that issued DNS queries and allow only the corresponding DNS replies. The number of register entries required to implement DNS amplification defense is equivalent to the number of machines to be protected. Each entry will be indexed by the hash of the source IP address (if the packet is coming from the internal network) or by the destination IP address (if the packet is coming from the external network). The value stored by each register entry represents the number of DNS queries requested by the corresponding IP address and yet to be resolved. The switch will drop any DNS response that maps to an entry with a value of 0.

SlowLoris attack: Unlike conventional DoS attacks that involve flooding the target with massive amounts of data, SlowLoris operates by opening multiple connections to the target server and maintaining them for an extended period. It does this by sending HTTP requests in a slow, fragmented manner, using partial request headers, and deliberately prolonging the completion of these requests. The typical defense solution is to limit the number of TCP connections per IP source that send a few bytes. This requires the switch to count the number of bytes each connection sends. Although this implementation can limit the number of connections that send a few bytes, it might exhaust the resources of the switch. Another approach is to get the average number of bytes per connection for each IP address and block IP addresses that send a few bytes per connection. This implementation requires two register entries. The first entry to store the number of flows for a given IP address and the other entry to get the number of bytes received from the same IP address. The second implementation reduces

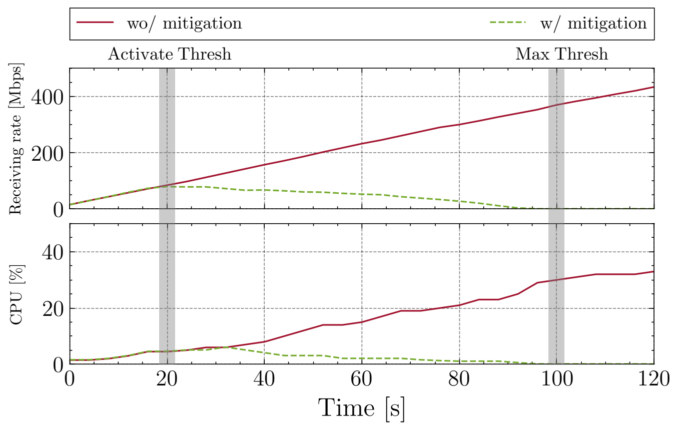


Fig. 10: DDoS detection and mitigation using P4 programmable switch.

the number of register entries from being equal to the number of connections to being equal to the number of IP addresses. This reduction might be significant as a single IP address can have multiple connections.

Fig. 9 presents the resources used by deploying SYN flood, ICMP flood, and DNS amplification defenses on a programmable switch. The P4 program can simultaneously monitor the count of SYN packets, ICMP packets, and DNS packets for 65,536 different IP sources. A fixed threshold is used on the count of packets per millisecond. If the threshold is exceeded, the data plane notifies the control plane. The control plane then constructs a configuration rule that probabilistically drops packets based on the attack volume. This technique is known as the Random Early Discard (RED) and is used by modern firewalls [22]. This program mainly utilized SRAM and Hashbit resources because it applies different hashing functions based on the packet protocol (e.g., TCP, ICMP, and UDP), and it utilizes registers to store the count of packets per source IP address.

To evaluate the performance of the application, a SYN flood attack was generated alongside background traffic from CAIDA captured from high-speed monitors on a commercial backbone link [23]. Fig. 10 shows the receiving rate and the CPU percentage of the SYN flood victim under attack. It can be seen that the RED method was successfully implemented as the receiving rate continuously decreases when the Activate threshold was surpassed.

V. Deploying DNS DPI in P4

DNS DPI is a network security and monitoring technique that inspects the contents of DNS packets. DPI goes beyond traditional DNS processing by analyzing the payload of DNS packets to identify potentially malicious or unauthorized activities. This technique is crucial for detecting and mitigating various cybersecurity threats like DNS tunneling, data exfiltration, and malware communication. DPI solutions often use pattern matching, heuristics, and anomaly detection to identify suspicious DNS traffic patterns. However, it also raises privacy concerns as it can be used to monitor user activities, emphasizing the need for responsible and transparent use of DPI technologies.

P4 switches provide a new vantage point for analyzing DNS traffic without the need to relay it to an external server. Thus reaping the benefits of its fast processing speed and privacy-preserving manner (domains are analyzed within the data plane) [11, 24, 25]. However, parsing the domain name in P4 switches has several challenges stemming from their hardware and software restrictions. This section discusses the challenges of parsing and processing domain names in the parser and match-action pipeline of P4 switches. Furthermore, it presents different solutions to overcome these challenges.

A. *Parser*

To understand the challenges of parsing domain names in a P4 switch, it is essential to understand how the variable-length domain names are encoded in DNS packets. A domain name is separated into labels, each delineated by a period. Each label is preceded by a single octet indicating the length of the label that follows, and the last label is followed by the *0x00* octet, indicating the end of the domain name. For instance, the domain name “google.com” is encoded as *(0x06)google(0x03)com(0x00)*. The length of a label can go up to 63 characters, and the maximum length of the entire domain name is 253 characters [3].

To preserve the line-rate processing speed of P4 programmable switches, parsers are not versatile enough to parse variable-length header fields. Although the latest P4$_{16}$ language provides a type *varbit* that can be used for header fields with variable length (e.g., IP options header field), the number of operations permitted on such a field is very limited. For instance, TNA does not allow the *varbit* field as a table key, thus making it unsuitable for domain name analysis.

One approach to parse the variable-length labels in P4 is depicted in Fig. 11. The parser defines a set of states for each possible label length and makes different parsing decisions by switching to different parser states based on the length of the label. Despite the fact that this method is effective for domain name parsing, the TCAM of the parser is limited and cannot cover all possible combinations in today’s hardware. A straightforward solution in P4 is to define a header for each possible length, as depicted in Listing 4.

Listing 4: Header definitions in P4 to cover all possible lengths for one DNS label.

```
header Label1_1 {
        bit<8> one_character;}
header Label1_2 {
        bit<16> two characters;}
header Label1_3 {
        bit<24> three_characters;}
header Label1_4 {
        bit<32> four_characters;}
        ...
header Label1_63 {
        bit<504> sixty_three_characters;}
```

The problem with the straightforward solution is that it would be inefficient and a waste of resources to extract the

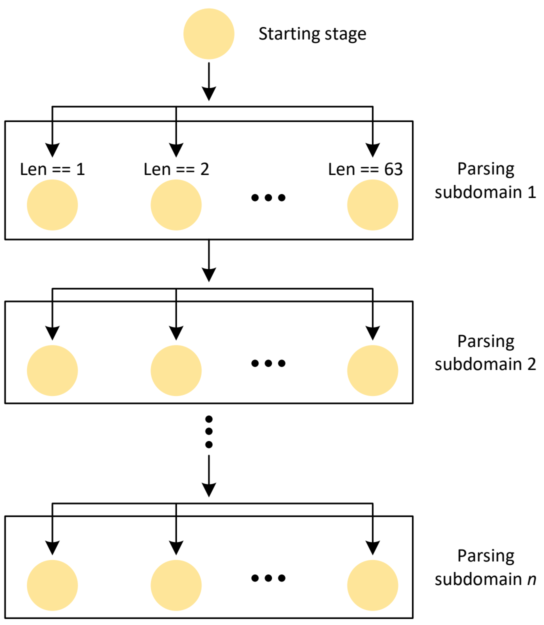


Fig. 11: Parsing DNS header. A state should be created for each possible length of a subdomain (1 .. 63), where the length of the domain is limited to 255.

whole label at once as this requires a header for each possible length for each label. The required resources per label are the summation of all the definitions, which is $\sum_{i=1}^{63} 8i = 16,128$ bits. An alternative approach that is adopted by P4 developers implementing DNS solutions [24, 25] is to store domain name labels in header fields with sizes in bytes that are exact power of 2 (e.g., 1, 2, 4, 8, 16 bytes). Listing 5 shows the P4 code required to define headers for a variable length label under the optimized approach. Only six definitions are needed to cover all possible lengths per domain. The required bits per label can be calculated as the sum of a geometric series following the formula:

$$S = \frac{a \cdot (r^n - 1)}{r - 1}$$

Where S is the sum of the series; a is the first term of the series; r is the common ratio; and n is the number of terms in the series. The first term is eight (which represents the number of bits to parse one character). The common ratio is two, and the number of terms is six. By plugging the values in the formula, the resulting number of bits is 504.

Listing 5: Optimized header definitions in P4.

```
header Label1_1 {
       bit<8> one_character;}
header Label1_2 {
       bit<16> two characters;}
header Label1_4 {
       bit<32> four_characters;}
header Label1_8 {
       bit<64> eight_characters;}
header Label1_16 {
       bit<128> sixteen_characters;}
header Label1_32 {
       bit<256> thirty_two_characters;}
```

In general, the number of bits saved by this technique is

$$S = 8 \cdot l \cdot (\sum_{i=1}^{n} i - (2^{\log_2(\lceil \frac{n}{2} \rceil)+1} - 1)$$

Where l is the number of labels, and n is the number of characters. For example, the number of bits saved by defining headers to parse one label with 63 characters is $S = 8 \cdot 1 \cdot (\sum_{i=1}^{63} i - (2^{\log_2(\lceil \frac{63}{2} \rceil)+1} - 1) = 15,624$ bits.

B. Match-action Pipeline

The parsed characters should be assigned as keys to tables in the match-action pipeline so that domain names can be identified. Fig. 12 shows a simplified example of how match-action tables identify DNS domains. The tables in this example identify at most four labels. Each label can be at most 15 characters. The first table matches on the four headers defined for label one (L1). L1 is extracted into the four headers L1_1, L1_2, L1_3, and L1_4. The remaining tables match on the remaining labels sequentially, i.e., label two (L2), label three (L3), and label four (L4). Note that the data plane can only track domain names that are populated by the control plane. For instance, any domain other than *www.google.come* and *www.another.domain.com* will not be identified by the tables in Fig. 12.

This poses a challenge for identifying domain names in the data plane since the processing pipeline has a limited number of stages and memory (i.e., a limited number of tables can be applied). For instance, Meta4 [24] can parse up to 4 labels with 15 characters each; Kaplan et al. [25] developed a solution that can parse up to 6 labels with 31 characters each.

To overcome the limited number of stages and memory in the hardware, which restricts the number of characters parsed and analyzed in a domain, packet recirculation is utilized (i.e., reiterating the packet multiple times) as done in P4DDPI and P4DGA [11, 13]. In a single packet pipeline pass, a limited number of characters can be matched. At the end of each pipeline pass, the labels that were matched are removed from the packet so that new labels appear in the next recirculation. Packet recirculation can be applied multiple times so that multiple labels can be matched. However, each recirculation adds more latency to the parsed packet. Also, note that packets are damaged after recirculation due to the removed header. Thus, for inline deployments (e.g., the switch is placed as a forwarding device), header manipulation should be performed on a cloned packet rather than the original one that will be forwarded to the next hop.

The resource usage of a P4 program that deploys the DNS DPI application [25] is depicted in Fig. 13. The figure shows that most of the exact matching resources of the programmable switch are occupied. This is because the parser contains many states to parse headers of variable length. The P4 program also supports monitoring up to 250 different domain names. The SRAM and TCAM resources are not reported because the DNS DPI application does not consume a noticeable amount (e.g., TCAM resource usage of the programmable match-action pipeline is 0).

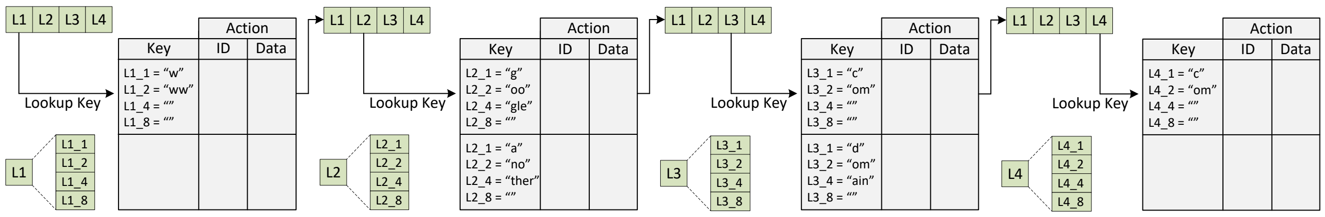


Fig. 12: Identifying the labels of DNS domains using match-action tables. Note that the number of tables can be more or less than the number of labels depending on the number of entries and the size of each entry.

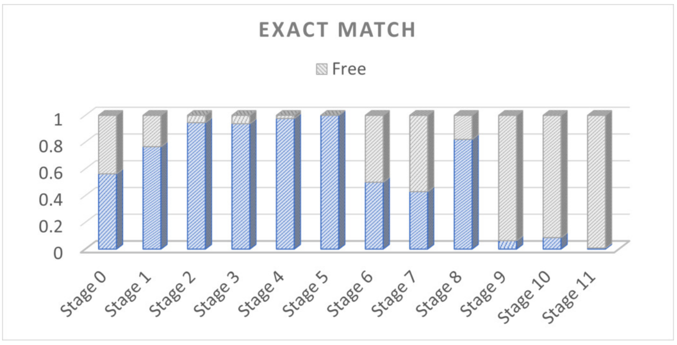


Fig. 13: Resources used by a DNS DPI application running on a P4 programmable switch.

The performance comparison between a DNS DPI application running over a programmable switch and a DNS DPI application running over Suricata is presented in Fig. 14. Suricata is a high-performance, open-source network analysis and threat detection software [26]. The graph to the left shows packet loss percentage with respect to DNS rates in Mbps. The packet loss percentage of the Suricata-based system increases with the increase in the DNS rate to reach around 60% at 400 Mbps. After that, the percentage starts to fluctuate between 52% and 65%. On the other hand, no packet losses are observed for the P4-based system.

The graph to the right shows the cumulative distribution function (CDF) of the latency. The latency represents the time a system requires to process a DNS domain. For the Suricata-based system, around 40% of the DNS domains are processed in less than 10 ms, 60% are processed in less than 200 ms, and 80% are processed in less than 250 ms. On the other hand, the latency required by the P4-based system is in the order of hundreds of nanoseconds.

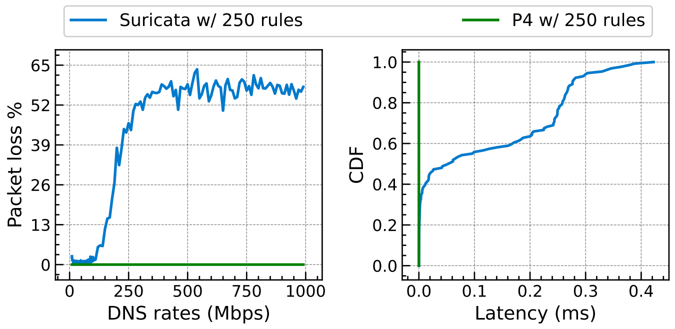


Fig. 14: Performance comparison between a DNS DPI application running over a P4 programmable switch and a DNS DPI application running over Suricata.

VI. Implementing Spoofing Defense in P4

Spoofing is a cybersecurity threat that involves an attacker disguising as a trusted entity to deceive a system or user. Spoofing is often employed in various cyberattacks, such as phishing, session hijacking, and DDoS amplification. It remains a challenge due to the evolving sophistication of attackers, underscoring the need for continuous monitoring and adaptive defenses. This section will focus on IP address spoofing attacks and defending mechanisms in P4.

Offloading spoofing defenses to the data plane has captured the attention of researchers as the existing defenses fail to cope with the increasing rate of attacks [1]. For example, using software-based solutions to handle typical attack scale ($\sim$Tbps) requires hundreds of server [27]. As the packet processing capabilities of a single PDP switch could match tens to hundreds of servers, multiple novel anti-spoofing defenses have been implemented on PDP switches. There are two main approaches to implementing anti-spoofing defenses in P4. The first approach uses the P4 PDP switch as a filter where whitelisting or blacklisting rules are hard coded by the control plane and require knowledge of the full topology [28]. The second approach utilizes the P4 PDP switch to dynamically build the rules to detect spoofed traffic and require minimal to no information about the topology [27, 29, 30].

A. *Hard Coding Anti-spoofing Rules in P4*

P4 applications following this implementation method accept a list of rules from the control plane to block/accept packets. Rules are described as hard coded because the are derived from the topology and not from the traffic exchanged at run time. The type of rules adopted by the P4 application is determined by the anti-spoofing mechanism being deployed. Some of the main anti-spoofing mechanisms that fall under this implementation method are Network Ingress Filtering (NIF) [31], Reverse Path Forwarding (RPF) and its variants [32], and Spoofing Prevention Method (SPM) [33].

Listing 6: Implementing NIF in P4.

```
table NIF_table {
    key = {
        hdr.ipv4.src_addr: ternary;
        ig_intr_md.ingress_port : exact;
    }
    actions = {
        NoAction;
        drop;
    }
    default_action = drop;
}
```

For NIF, the network administrator defines a set of IP prefixes per interface to accept packets. This mechanism can be implemented in P4 using a single match-action table that utilizes ternary matching as shown in listing 6. Ternary matching is used to specify the IP prefix of IP addresses to be permitted by the switch. If the IP address and ingress port of the packet do not match any entry of the NIF_table, the default action is to drop the packet. RPF depends on the router's forwarding table to accept packets. The P4 implementation is similar to NIF, where a single match-action table can be used to detect spoofed packets. The entries in this table are the forwarding rules.

Listing 7: Implementing SPM in P4.

```
action set_spm_key(bit<16> key){
    hdr.ipv4.id = key;
}

table SPM_arrival {
    key = {
        hdr.ipv4.src_addr: ternary;
        hdr.ipv4.dst_addr: ternary;
        ig_intr_md.ingress_port: exact;
        hdr.ipv4.id: exact;
    }
    actions = {
        NoAction;
        drop;
    }
    default_action = drop;
}

table SPM_departure {
    key = {
        hdr.ipv4.src_addr: ternary;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {
        NoAction;
        set_spm_key;
    }
    default_action = NoAction;
}
```

For SPM, multiple routers should cooperate to detect and drop spoofed packets. This technique is deployed at the Autonomous System Boundary Routers (ASBRs). Communication between neighbor Autonomous Systems (ASs) is performed through ASBRs. ISPs maintain globally unique keys for each pair of ASs, such that packets exchanged between the two ASs should be market with this key. The P4 implementation of SPM is described in listing 7. Table *SPM_arrival* verifies that packets coming from a remote AS carry the legitimate SPM key by maintaining a mapping between the source-destination IP prefix pairs and the key. If the packet hits the table, no action will be taken, and the packet will be forwarded. Otherwise, the packet will be dropped. Table *SMP_departure* stamps the packets departing from the AS with the legitimate SPM key. The key is stored in the identification field of the IP header.

B. Dynamically Populating Anti-spoofing Rule in P4

P4 applications following this implementation method determine the rules to detect spoofed packets from the live traffic at run time. One anti-spoofing mechanism that falls under this implementation method is Hop-Count Filtering (HCF) [34, 35]. HCF mitigates spoofed IP traffic by maintaining IP-to-Hop-Count (IP2HC) mapping table [27]. The number of hops traversed by the packets is compared with the IP2HC table. If they match, the packet will be forwarded. Otherwise, the packet will be classified as spoofed and dropped. Implementing this approach in P4 requires the interference of the control plane. Because of the hardware limitation of the P4 PDP switches, the approach maintains a case for the most observed IP addresses in the data plane and utilizes the control plane to keep track of the less active IP addresses. The data plane maintains three modules: the IP2HC inspecting module, the TCP session monitoring module, and the cache statistics modules. The IP2HC module inspects the validity of the packets where the data plane compares the number of hops traversed by the packets with its records. It is implemented in at least three different stages in the P4 pipeline. The first stage hosts a table to match packets on their IP address and returns the cache index associated with that IP. The second stage utilizes a register array to retrieve the recorded hop count for that IP address. The last stage/s compares the time-to-live field to the extracted hope count and verifies that the packet traversed the correct number of hops; thus, it is not a spoofed packet.

The TCP session monitoring module is responsible for updating the number of hops per IP address at run time. This module monitors the sequence and acknowledgment numbers of the TCP sessions to validate their integrity. TCP sessions that completed the 3-way handshake and are transmitting data are classified as trusted traffic and used to determine/update the IP2HC table entries. The P4 implementation of this module requires two register arrays. The first array maintains the TCP sequence and acknowledgment numbers. The second register array maintains the TCP flags of the TCP session, which allows it to determine the state of the connection. Finally, the cache module is responsible for identifying the IP addresses and prefixes to be cached on the data plane and those that should be maintained by the control plane. The cache is built using counters that monitor the number of hits on the different entries of the IP2HC table. If an entry has more hits than a specified threshold, the IP prefix associated with that entry is considered *hot* and will continue to be cached by the data plane. However, if an entry has fewer hits than the threshold, it will be moved to the control plane.

The main drawback of implementing the HCF approach in P4 is the limited number of IP prefixes that can be maintained in the data plane. If the IP address of an incoming packet does not match an entry in the IP2HC table, the data plane forwards the packet to the control plane. The control plane processes the packet and takes an action on whether to forward or drop the packet. Because the control plane adds processing time overhead for legitimate packets, P4 security applications that leverage external processing capabilities (i.e., using the CPUs of external servers) adopt two modes of operation. The first mode focuses on collecting data and monitoring the network without taking action. If the P4 application detects malicious behavior, it switches to the second mode, which drops suspicious packets.

Another approach to detecting spoofed packets is to use the source IP and source MAC of the incoming packets as a filter. The source IP and MAC of the arriving packets are hashed, and the hash is compared to the list of permitted entries. This approach can be deployed in P4 using the match-action tables-based implementation or stateful registers-based implementation. These two implementation methods are discussed extensively in section III. The considerations for each implementation are the same for anti-spoofing security applications and stateful packet filter security applications.

VII. Implementing Cryptography in P4

Cryptography is crucial in network security, where it ensures the confidentiality, integrity, and authenticity of data as it travels across networks. By encrypting data packets, cryptography prevents unauthorized parties from intercepting or tampering with sensitive information such as login credentials, financial data, or private communications. Cryptographic primitives, such as symmetric ciphers and hash functions, are typically executed on end hosts, either using general-purpose CPUs or specialized cryptographic accelerator co-processors. Although deploying cryptographic algorithms in the data plane improves the privacy and security of networks, P4 PDP switches have multiple hardware constraints that make the P4 implementation of the algorithms challenging. This section focuses on achieving integrity and confidentiality in P4 by describing the implementation of some hash functions and symmetric ciphers. To the best of the author's knowledge, there is no P4 implementation of asymmetric ciphers that target a hardware PDP switch.

A. Achieving Confidentiality in P4 by Implementing Symmetric Ciphers

Symmetric ciphers achieve confidentiality by encrypting data with a single shared secret key that both the sender and receiver use for encryption and decryption. This process ensures that only parties with the correct key can access the original information. The encryption transforms plaintext into ciphertext, rendering it unreadable to anyone without the key, while decryption reverses this process to retrieve the plaintext.

The main challenge of implementing symmetric cipher algorithms in P4 PDP switches arises from the limited number of stages and the available arithmetic operations natively supported. Symmetric ciphers require processing the data in multiple rounds, such that the output of one round is fed to the next round. This chain of dependancy prevents the P4 PDP switches from encrypting or decrypting the data in a single pipeline pass due to the limited number of stages, where recirculation is necessary. Besides, some of the arithmetic operations required by symmetric ciphers are not supported by the P4 PDP switches (e.g., division). To address this limitation, Scrambled Lookup Table (SLT) technique is used [36], where the complex arithmetic operations are precomputed and installed in the data plane using match-action tables.

Advanced Encryption Standard (AES) is a standardized symmetric algorithm that has been successfully implemented in P4 [36]. AES is a widely used symmetric encryption algorithm that encrypts and decrypts data in fixed block sizes of 128 bits. It supports key lengths of 128, 192, or 256 bits, providing varying levels of security. AES operates through a series of transformations, including substitution, permutation, and mixing, which enhance data security. The data block input to AES is organized as a 4-by-4 matrix, where each cell contains a byte of data. During an encryption round, the data block is first XORed with a block of the key (*AddRoundKey* step). After that, each byte from the resulting matrix is replaced by another byte using a Substitution box (S-box) (*SubBytes* step). Then, each row in the data block is cyclically shifted by 0, 1, 2, or 3 locations (*SubBytes* step). Finally, the columns are mixed together (*MixColumns* step). The number of rounds to be performed on the input depends on the AES variant, where 10, 12, and 14 rounds are needed for AES-128, AES-192, and AES-256, respectively.

Implementing AES in P4 is based on the observation that *AddRoundKey* and *MixColumns* steps can be deployed using a series of XORs; *ShiftRows* step can be deployed by changing variable names; and transformation table (T-table) construction can combine S-box, variable renaming and polynomial multiplication. The T-table is a precomputed lookup table that combines the S-box (substitution) and the finite field multiplication operations involved in the *MixColumns* step of the AES algorithm. Thus, each AES round can be implemented using four lookup tables and a series of XORs. Although this approach removes the obstacle of performing complex mathematical operations in P4 using the T-table constructions, it does not deal with the long dependency chain created from the series of XORs. In order to fit a full AES encryption round in a single pipeline pass, the *AddRoundKey* and *SubBytes* steps are combined using multiple SLTs. The SLTs are built by permuting and storing 16 look-up tables per encryption round. Each one byte of data to be encrypted will pass by the SLTs first to perform the *AddRoundKey* and *SubBytes* steps. The output of the SLTs will then be fed into a look-up table that performs the *ShiftRows* and *MixColumns* steps. Finally, the resulting four bytes are XORed to obtain the final result. The result of the first encryption round will then be fed to the SLTs of the second encryption round.

AES implementation in P4 trades off the number of operations with the amount of memory used by compressing the *AddRoundKey* and *SubBytes* steps into 16 tables per round. Each table has 256 rules. If AES-256 is used, then there are 14 rounds, which is translated into 14 (rounds) x 16 (tables per round) x 256 (rules per table) = 57,000 rules. Each rule uses 6 bytes of memory. Thus, in total, this approach uses 344 kilobytes (KB) or around 15% of the available memory resources of the P4 PDP switch. Another important aspect to consider is the number of match-action tables utilized. P4 PDP switches support a high, yet limited, number of match-action tables. In a single pipeline passes, the packet data undergoes two encryption round, which requires 32 match-action tables or what is equivalent to 25% of the supported number of match-action tables. Finally, P4 PDP switches have a limit on the number of arithmetic operations to be performed in a single pipeline stage. In each encryption round, the 16 SLTs produce 16 four-byte values grouped into four rows. Each row requires 3 XOR operations to combine the four four-byte values. Thus, each round requires 12 XOR operations. If two rounds are implemented in a single pipeline pass, then 24 XOR operations are needed. This value is around 10% of the supported number of operations.

B. Achieving Integrity in P4 by Implementing Secure Hash Functions

Cryptographically secure hash functions achieve integrity by generating a unique fixed-size output (called a hash or digest) from input data. When data is sent or stored, a hash of the original data can be calculated and sent alongside it or saved. Upon retrieval or receipt, the hash can be recalculated and compared with the original hash. If the hashes match, the data has not been altered; if they differ, this indicates that the data may have been tampered with. Unfortunately, P4 PDP switches lack support for secure hash functions, instead offering only CRC16 or CRC32 hashes. The limited length of these hashes makes them vulnerable to the birthday bound [37], rendering them susceptible to collisions. Furthermore, unlike secure hash functions, the CRC family does not provide preimage resistance, which is the property that makes it difficult to recover the input message from the hash output. In the case of a CRC32 function with known polynomials and a single output, an adversary would only need to try a maximum of 2^{32} messages to identify the corresponding input message. If the polynomials are unknown, the adversary could attempt at most 2^{32} polynomials to determine the ones being used, given one input message and the resulting output [38].

As an alternative to CRC, SmartCookie [39] has implemented HalfSipHash-2-4 [40] in P4. HalfSipHash-2-4 is a cryptographic hash function designed for efficiency and security, particularly suited for short inputs. It produces a 64-bit (8-byte) output and employs a structure that includes 2 computation rounds and 4 finalization rounds for each 32-bit word. The algorithm is seeded with secret keys to initialize 4 variables. The 2 computation rounds process the 4 variables using Add, Shift, and XOR operations. The output of the computation rounds is fed to the 4 finalization rounds. Finally, the algorithm XOR the 4 variables to produce the hash value. A main challenge in deploying HalfSipHash-2-4 in the data plane is the long dependency chain. Each computation round requires 14 operations, and the output of one arithmetic operation is

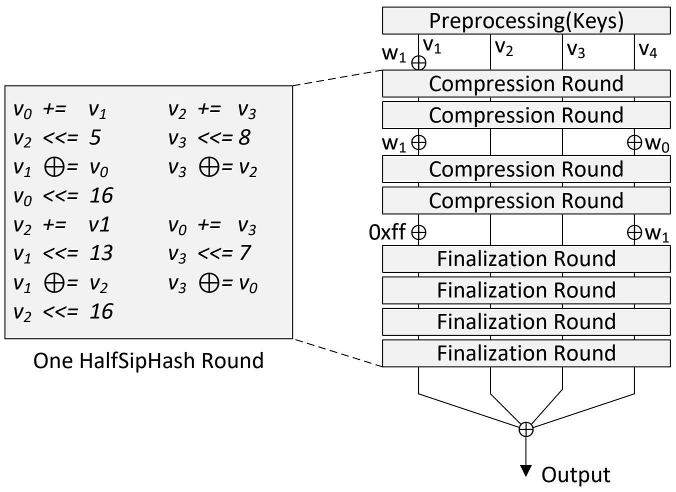


Fig. 15: The structure of HalfSipHash-2-4 [39].

fed to the next operation. Another challenge is the need for performing 6 circular left shifts in each computation round (as shown in Fig. 15), which is not natively supported by P4 PDP switches [39]. In order to overcome the first challenge, the dependency chain of HalfSipHash-2-4 can be shortened to 4 by manually distributing the operations over the stages and by using the recirculation feature of P4 PDP devices. Manually optimizing the distribution of operations over the stages allows the P4 switch to perform a computation round in four stages only per 32-bit word. If w words are to be hashed, the total number of compression and finalization rounds is $2w + 4$. The number of pipeline passes required to hash w words is $w/2$ if both the ingress and egress pipelines are used for hashing or $w + 1$ if only the ingress pipeline is used.

VIII. Lessons Learned

This section presents the lessons learned from implementing security applications on P4 PDP switches. The lessons are summarized in Table II.

A. Multiple P4 Implementations

Security applications can be implemented in multiple ways. The programmer should decide on the implementation based on the requirements. In the case of implementing a stateful packet filter, the programmer can use match-action tables-based implementation if the objective is to have full control and per-event visibility (i.e., being notified for every flow accepted, for every flow dropped, and for every entry added or removed from the match-action table). On the other hand, utilizing registers allows the programmer to run the packet filter entirely on the data plane, increasing its performance. In the case of implementing a DDoS defense, the programmer should decide if detecting the IP address of the attacker is a priority. In that scenario, traffic statistics (e.g., the rate of SYN packets and the rate of ICMP ECHO packets) should be grouped through the source IP address (for the traffic destined for the internal network). Otherwise, using destination

TABLE II: Lessons learned from implementing security applications on P4 PDP switches

Lesson	Description	Impact on Security Applications
Multiple P4 implementations	Stateful security applications can be implemented in P4 either following the tables-based or the registers-based approaches	Tables-based implementation provides higher visibility, while registers-based approaches provide higher response time
Accuracy, resources, and performance trade-off	Resource constraints require the security applications to have a trade-off between their accuracy and their performance	Security applications like stateful packet filters and anti-spoofing mechanisms trade the number of security policies with the number of tables and matching criteria
Resource allocation	The distribution of the resources over the component of the P4 application should be provisioned before compiling the application	Insufficient planning can lead to hash collisions in flow tracking, risking inaccurate flow identification and reduced protection against attacks like DDoS
Parsing variable length headers	P4 does not provide built-in functions to efficiently parse variable length headers	DPI applications might require multiple pipeline passes (re-circulation) to extract variable length headers
Matching on variable length headers	P4 does not support tables with variable length matching keys	DPI applications should partition a variable length header into a set of fixed-size headers to process them, imposing additional processing overhead
Managing dependencies	P4 supports a limited number of dependent actions to be performed in a single pipeline pass	One or more packet recirculations are needed by security applications with long dependency chains (e.g., cryptography), which reduces the maximum achievable throughput
Control plane and data plane interaction	The interaction between the control plane and the data plane is unavoidable for P4 applications that require computation-intensive tasks	P4 security applications depend on the control plane to deploy complex detection algorithms
Cryptographic-based functions	The set of supported arithmetic operations, and the number of stages are the main challenges for implementing cryptographic functions in P4	Cryptographic functions in P4 entail multiple packet recirculations and restrict the size of data input

IP addresses is preferred in order to reduce memory utilization, allowing more DDoS defenses to be integrated into the same P4 program.

B. Accuracy, Resources, and Performance Trade-off

There is a trade-off between the accuracy, the resources, and the performance of the security applications. Security applications that utilize match-action tables to implement some custom functionalities trade the number of different matching keys (i.e., keys with different components) with accuracy and performance. As the number of enforced policies increases, more matching criteria and tables are required, thus, increasing the resources required in the switch. At the same time, using more keys provides more flexibility and enhances the application's performance. For example, increasing the number of keys while implementing a stateful packet filter using match-action tables allows the packet filter to accept ICMP flows that belong to an ongoing session using the identifier field of the ICMP header. The identifier field should be only part of the s2c matching key. In such a scenario, the c2s and s2c flows will have different tables. On the other hand, using a single table for c2s and s2c flows reduces the amount of resources used at the cost of preventing the packet filter from using the identifier field of the ICMP header. Not using the identifier field in a stateful packet filter prevents the packet filter from checking if an ICMP s2c flow belongs to the existing ICMP c2s flow. Consequently, an attacker can use this vulnerability to trick the stateful packet filter into blocking legitimate s2c flows.

C. Resource Allocation Planning

Another primary consideration while implementing a security application is resource allocation. During the compilation, the compiler checks if the P4 program can fit in the P4 switch. The compilation will be successful if the resources can be allocated by the compiler to the program. Otherwise, there will be a compilation error. After compilation, the resources utilized by a P4 program cannot be modified (e.g., the size of tables and the number of register entries). The programmer should be aware of the amount of resources provided by the switch and how these resources should be distributed on the tables (i.e., the sizes of the tables or the number of entries), on the registers (i.e., the number of indices), and on the other components of the programmable pipeline. For example, the programmer should consider the number of flows before defining the number of bits to index a register array. Flows are indexed using their hash values. As the number of flows increases, the length of hashes should increase. Otherwise, collision occurrences might increase. Regardless of the size of the input to a hash function, the output will be fixed based on the used algorithm. For instance, a hash algorithm with 16 bits long output produces a hash value between 0 and 2^{16}. Because the set of output is finite, the probability of having a collision increases as the number of hashed values increases.

D. Parsing Variable Length Headers

Programmable switches have no built-in functions to efficiently parse variable length headers. Although the *varbit* data type allows the parser to parse headers with variable length, a varbit header is not allowed to be used as a key in match action tables. This keeps the programmer with two methods. The first method is to define headers for each possible length. For example, if a variable length header can have 10 different possible lengths, the programmer should define ten different headers, each covering one possibility. Although this method is effective, the parser might not have enough resources to cover all possible lengths.

The other method is to define headers with sizes as exact powers of 2 and use one or more defined headers to cover all other possible lengths. For example, if the length of a variable header is three, then it can be parsed by using the defined headers of lengths one and two. Note that although this method significantly reduces memory resources compared to the first method, it requires the same number of transitions in the parser. For instance, a header with variable lengths between one and seven requires only three header definitions (headers with lengths 1, 2, and 4), it still needs one state per possible length (i.e., seven different states in the parser).

E. Matching on Variable Length Headers

As mentioned before, the *varbit* data type cannot be used as a key in match-action tables. The compiler requires a discrete number representing the amount of resources needed and will produce an error if a variable length header is used as the key in match-action tables.

A variable length header should be parsed into fix-sized headers, and then use the headers as the lookup key. An example is identifying the labels of DNS domains. Assuming that the P4 program supports labels with at most 15 characters, then the parser should define four headers per label of lengths 1, 2, 4, and 8 bytes, respectively. The keys for the match-action tables should match on the four headers simultaneously. This means that even if a label is three characters long and can be extracted using the headers of lengths 1 and 2, the four headers should be set to valid. Consequently, 120 bits will be allocated for a 24-bit header. It is worth noting that some researchers avoid wasting resources by matching only on the headers utilized by the parser. However, this means that more operations are needed to detect the valid headers, which might constrain the number and length of labels that can be identified by the match-action pipeline.

F. Managing Dependencies

Although the limited resources of P4 programmable switches are one of the barriers against implementing applications that require complex computation, the programmer will face more compilation problems caused by the limited number of stages. Because the switches should sustain nanosecond processing speed, they contain a few stages such that a limited number of operations can be performed per stage. The stages are connected sequentially so that a packet can only move forward (unless a recirculation is applied). For this, an action that takes as input the output of another action (there is a dependency) is placed in a later stage(s) by the compiler. If a stage cannot handle all the operations of an action, the compiler will distribute the action over multiple stages. If the longest sequence of operations is longer than the available stages on a programmable switch, the P4 program will fail to compile. While developing a P4 application, the programmer should minimize the number of operations per action and the dependencies between actions.

The resources of a P4 programmable switch are divided on the stages, such that the resources of one stage cannot be accessed from another stage. Having dependencies between actions might prevent the P4 compiler from utilizing all the resources of a stage before moving to the next one. For example, assume that a P4 program should apply some actions on SYN packets only. To do this, the program should first check if the packet has the IP header, then it should check if the TCP header is valid, and finally check if the packet is a SYN packet. Checking if the IP header is valid requires one stage. Checking if the TCP header is valid requires another stage. Checking if the TCP packet is SYN requires one or more stages depending on the algorithm (e.g., only considering the flags field of TCP or considering the size of the packet as well). The resources required by the actions to be performed on the SYN packets can only be allocated in the stages after which the type of packet is identified. Consequently, the resources of the first few stages might not be fully utilized.

G. Control Plane and Data Plane Interaction

A task running on the data plane achieves significant processing speed and granularity over a task running on the control plane. The data plane operates through the hardware of the programmable switch, while the control plane is constrained by the software capabilities (e.g., limited processing, granularity, and scalability). Typically, all the tasks should be offloaded to the data plane. However, due to its limited resources, it is unfeasible to run computation-intensive security applications fully on the data plane, and consequently, the applications are partitioned over the data and the control planes.

The two planes interact through the APIs provided by the vendors of the programmable switches. The data plane should be responsible for the tasks that require high precision and line-rate processing speed. The control plane should be responsible for the data aggregation and storage tasks, as well as any computational unfeasible tasks on the data plane (e.g., machine learning algorithms). An example of this interaction is when implementing a DDoS defense in P4. The data plane collects statistics over the packet over a short period (e.g., every one second or 100 milliseconds) and pushes the statistics to the control plane. Although the data plane can use fixed thresholds to mitigate DDoS attacks, the control plane can run more complex algorithms to enhance the accuracy of the detection. The control plane can also archive the data for longitudinal data analysis.

H. Cryptographic-based Functions

The implementation of cryptographic functions on P4 PDP switches faces two primary challenges. The first challenge stems from the limited set of operations supported by P4 switches, which include addition, subtraction, and XOR but exclude multiplication and division. This limitation restricts the data plane's ability to perform computations essential for many cryptographic algorithms. A common solution to this constraint is to precompute outputs for a range of possible inputs and store these results in match-action tables. While this enables the data plane to implement cryptographic functions like AES, it imposes restrictions on the range of input sizes that can be processed.

The second challenge involves the limited number of pipeline stages. To support high throughput, P4 PDP devices restrict the number of stages available in a single pipeline pass, limiting the number of sequential actions that can be performed on packets. When one action depends on the result of another, they are considered sequential and cannot execute concurrently, prompting the P4 compiler to place them in consecutive stages. Issues arise if a P4 application requires more stages than the hardware provides in a single pass. Consequently, implementing cryptographic functions in P4 often necessitates multiple pipeline passes to compensate for the low stage count per pass.

IX. Future Directions

A. Expanding Memory Resources

The limited memory resources on P4 PDP switches prevent security applications from maintaining per-flow records in the data plane (e.g. DDoS defenses and stateful firewalls). A novel approach suggested extending the memory of P4 PDP switches using the remote Dynamic Random Access Memory (DRAM) technology [41, 42]. Remote DRAM allows the P4 PDP switches to access external memory resources over the network. An access channel is maintained by the data plane of the switch and the remote DRAM, using protocols like RoCE (RDMA over Converged Ethernet). Experiments show that the latency added by RoCE to store or retrieve data is around 1-2 microseconds. Following this technique, the switch can extend the number of security rules and the number of flows monitored by the security applications. However, this approach has a few limitations that could be addressed in the future. First, the switch should explicitly specify the address from which the information should be stored or retrieved. Thus, the data plane should maintain a data structure to map flows to memory addresses, which requires a considerable amount of resources. Future work should focus on building a system that allows the switch to efficiently manage the remote memory. Second, the switch can store or retrieve information from the remote memory using a single request. On the other hand, updating the values in the remote memory is expensive as it entails the request to fetch the data, update it in the data plane, and then store it back. Future work should identify a more efficient way to update values on the remote memory. Finally, packet loss can significantly degrade the performance of the system as it might lead to synchronization issues between the switch and the remote memory. Future work should aim at defining a mechanism to handle packet losses in P4 as well as building well-defined APIs to facilitate the interaction between the data plane and the remote memory.

B. Extending Arithmetic Operations

Most cryptographic functions require arithmetic operations that are not supported by P4 PDP switches, such as division and floating-point calculations. To implement these functions in P4, developers typically precompute values and store them in match-action tables, but this approach limits the input data size and often requires packets to be recirculated multiple times. Instead of precomputing values, exploring the capabilities of smartNICs could complement P4 applications by expanding the range of supported arithmetic operations. SmartNICs (Smart Network Interface Cards) are advanced NICs equipped with domain-specific processors optimized for infrastructure tasks, such as compression, decompression, encryption, and decryption. Future research should focus on integrating P4 PDP switches with smartNICs to enable complex computations within the data plane. Additionally, software-based acceleration techniques should be investigated to further enhance P4-based security systems. Notably, some studies have already demonstrated that integrating software-based accelerators can improve the memory efficiency of P4 security applications [39]. Future work could extend these techniques to mitigate arithmetic computation constraints in P4.

C. Enabling Pattern Matching

Pattern matching is essential for DPI security systems that block traffic containing specific keywords. For example, if a network administrator sets a rule to block all HTTP traffic with the term *Google* the security system examines each packet's URL to check for matches to the pattern "*Google*." Here, the asterisks (*) act as wildcards, allowing any characters before or after *Google*. However, while this type of flexible matching is useful, P4 does not natively support it. The main difficulty with using wildcards in P4 is that the number of characters for a match is set during implementation and cannot be changed after the P4 program is compiled. A workaround is to use ternary matching, allowing patterns like "??Google," where each "?" can represent any single character. Unfortunately, this method is inefficient because covering one wildcard rule may require multiple "?" rules to account for the maximum pattern length. Since patterns in P4 are transformed into matching keys, their length must be defined in advance. Besides, multiple tables might be required to cover the different possible numbers of characters in a wildcard operation. Future work should aim to enable pattern matching in P4-based security applications either through a workaround in P4 or utilizing the regular expressions accelerator of smartNICs.

D. Parsing Variable Length Protocols

Various approaches have been developed to parse protocols with a single variable-length header, such as the domain name in DNS or the URL in HTTP. However, to the authors' knowledge, no P4 application is currently capable of parsing protocols with multiple variable-length headers like Transport Layer Security (TLS). TLS is a cryptographic protocol designed to provide secure communication over a network, ensuring data privacy, integrity, and authenticity between two endpoints, like a web browser and a server. Security applications often monitor encrypted traffic by examining the SNI (Server Name Indication) field in the TLS protocol, an extension that allows clients to specify the hostname they are connecting to during the TLS handshake. TLS includes multiple variable-length headers, such as the session ID, cipher suite list, and compression method, which present additional challenges for parsing because each header requires a unique set of states in the parser. Parsing TLS in P4 is, therefore, significantly more complex than parsing simpler protocols like DNS or HTTP. Future research should focus on optimizing P4 parsing logic to support protocols with multiple variable-length headers.

X. Conclusion

The complexity associated with developing P4 applications has often served as an obstacle for researchers. This paper tackles the issue by providing a comprehensive guide to the process of designing P4 security applications. It goes beyond theory and delves into practical implementation by showcasing the creation of several security applications on P4 PDP switches and sharing the P4 code used for these applications. Furthermore, the paper provides key insights and outlines the critical factors to consider when developing P4 security applications.

References

[1] A. AlSabeh, J. Khoury, E. Kfoury, J. Crichigno, and E. Bou-Harb, "A survey on security applications of P4 programmable switches and a STRIDE-based vulnerability assessment," *Computer Networks*, vol. 207, p. 108800, 2022.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic {DDoS} defense," in *24th USENIX security symposium (USENIX Security 15)*, pp. 817–832, 2015.

[5] A. Panchenko, L. Pimenidis, and J. Renner, "Performance analysis of anonymous communication channels provided by tor," in *2008 Third International Conference on Availability, Reliability and Security*, pp. 221–228, IEEE, 2008.

[6] P. Zilberman, R. Puzis, and Y. Elovici, "On network footprint of traffic inspection and filtering at global scrubbing centers," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 5, pp. 521–534, 2015.

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

[8] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE access*, vol. 9, pp. 87094–87155, 2021.

[9] J. Gomez, E. F. Kfoury, J. Crichigno, and G. Srivastava, "A survey on TCP enhancements using P4-programmable devices," *Computer Networks*, vol. 212, p. 109030, 2022.

[10] A. Mazloum, E. Kfoury, J. Gomez, and J. Crichigno, "A Survey on Rerouting Techniques with P4 Programmable Data Plane Switches," *Computer Networks*, vol. 230, p. 109795, 2023.

[11] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4ddpi: Securing P4-programmable data plane networks via DNS deep packet inspection," in *NDSS Symposium 2022*, 2022.

[12] E. Kfoury, J. Crichigno, E. Bou-Harb, and G. Srivastava, "Dynamic Router's Buffer Sizing using Passive Measurements and P4 Programmable Switches," in *2021 IEEE Global Communications Conference (GLOBECOM)*, pp. 01–06, IEEE, 2021.

[13] A. AlSabeh, K. Friday, J. Crichigno, and E. Bou-Harb, "Effective DGA Family Classification using a Hybrid Shallow and Deep Packet Inspection Technique on P4 Programmable Switches," 2023.

[14] A. Mazloum, E. Kfoury, S. Sur, J. Crichigno, and N. Ghani, "Enhancing Blockage Detection and Handover on 60 GHz Networks with P4 Programmable Data Planes," 2023.

[15] E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4Tune: Enabling Programmability in Non-Programmable Networks," *IEEE Communications Magazine*, vol. 61, no. 6, pp. 132–138, 2023.

[16] E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4BS: Leveraging Passive Measurements from P4 Switches to Dynamically Modify a Router's Buffer Size," *IEEE Transactions on Network and Service Management*, 2023.

[17] E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4CCI: P4-based online TCP congestion control algorithm identification for traffic separation," in *ICC 2023-IEEE International Conference on Communications*, pp. 4007–4012, IEEE, 2023.

[18] J. Crichigno, E. Kfoury, E. Bou-Harb, N. Ghani, Y. Prieto, C. Vega, J. Pezoa, C. Huang, and D. Torres, "A flow-based entropy characterization of a NATed network and its application on intrusion detection," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2019.

[19] A. Mazloum, "P4 codes." [Online]. Available: https://github.com/AliMazloum/P4-Security-Applications.git, Accessed on 11-7-2023.

[20] Palo-Alto, "Security policy fundamentals." [Online]. Available: https://tinyurl.com/528ufppz, Accessed on 5-10-2023.

[21] B. Spang and N. McKeown, "On estimating the number of flows," *BS*, vol. 19, p. 4, 2019.

[22] P. Networks, "Understanding DoS protection in PAN-OS." [Online]. Available: https://tinyurl.com/3dap6dhb, Accessed on 9-20-2023.

[23] CAIDA, "The CAIDA anonymized Internet traces 2019 dataset." [Online]. Available: https://data.caida.org/datasets/passive-2019/, Accessed on 9-20-2023.

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

[25] A. Kaplan and S. L. Feibish, "Practical handling of DNS in the data plane," in *Proceedings of the Symposium on SDN Research*, pp. 59–66, 2022.

[26] Suricata, "Suricata is far more than an IDS/IPS." [Online]. Available: https://suricata.io/, Accessed on 10-9-2023.

[27] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan, "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering," in *2019 IEEE 27th international conference on network protocols (ICNP)*, pp. 1–12, IEEE, 2019.

[28] H. Gondaliya, G. C. Sankaran, and K. M. Sivalingam, "Comparative evaluation of IP address anti-spoofing mechanisms using a P4/NetFPGA-based switch," in *Proceedings of the 3rd P4 Workshop in Europe*, pp. 1–6, 2020.

[29] G. Simsek, H. Bostan, A. K. Sarica, E. Sarikaya, A. Keles, P. Angin, H. Alemdar, and E. Onur, "DroPPPP: a P4 approach to mitigating dos attacks in SDN," in *Information Security Applications: 20th International Conference, WISA 2019, Jeju Island, South Korea, August 21–24, 2019, Revised Selected Papers 20*, pp. 55–66, Springer, 2020.

[30] P. Kuang, Y. Liu, and L. He, "P4DAD: Securing duplicate address detection using P4," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2020.

[31] D. Senie, "Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing," *RFC 2827*, 2000.

[32] F. Baker and P. Savola, "Ingress filtering for multihomed networks," tech. rep., 2004.

[33] A. Bremler-Barr and H. Levy, "Spoofing prevention method," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 1, pp. 536–547, IEEE, 2005.

[34] C. Jin, H. Wang, and K. G. Shin, "Hop-Count Filtering: an effective defense against spoofed DDoS traffic," in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 30–41, 2003.

[35] H. Wang, C. Jin, and K. G. Shin, "Defense against spoofed ip traffic using hop-count filtering," *IEEE/ACM Transactions on networking*, vol. 15, no. 1, pp. 40–53, 2007.

[36] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pp. 8–14, 2020.

[37] P. Flajolet, D. Gardy, and L. Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Applied Mathematics*, vol. 39, no. 3, pp. 207–229, 1992.

[38] L. Wang, P. Mittal, and J. Rexford, "Data-plane security applications in adversarial settings," *ACM SIGCOMM Computer Communication Review*, vol. 52, no. 2, pp. 2–9, 2022.

[39] S. Yoo, X. Chen, and J. Rexford, “SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes,” in *USENIX Security*, 2024.

[40] J.-P. Aumasson and D. J. Bernstein, “SipHash: a fast short-input PRF,” in *International Conference on Cryptology in India*, pp. 489–508, Springer, 2012.

[41] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, “Generic external memory for switch data planes,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pp. 1–7, 2018.

[42] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, “Tea: Enabling state-intensive network functions on 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*, pp. 90–106, 2020.