– Independent Work Report Spring 2020 –

Aggregating Internet Traffic by Domain Name using Programmable Switches

Jason Kim
Adviser: Jennifer Rexford

Abstract

Being able to identify network traffic with human-readable domain names instead of IP addresses is helpful for network administrators who might need to monitor and identify traffic associated with certain domains. The problem is that most current methods of network monitoring are done outside the network and require collecting and examining large amounts of network traffic in a way that may compromise user privacy. The purpose of this project was to implement network monitoring by domain name on a programmable switch. A data plane implementation has the benefit of preserving the privacy of sensitive user information and behavior. In addition, performing network monitoring in the data plane is good for efficiency since data does not have to be shipped off the switch for analysis. Finally, a data plane implementation allows network operators to take action on network traffic. Network administrators could choose to rate-limit, block, or mark packets based on their associated domain. The creation of such an implementation in the data plane is technically challenging because the nature the hardware of programmable switches greatly restricts the capabilities of the programmer, especially with regard to the ability to read and write variable length strings of data.

1. Introduction

The majority of network traffic between clients and servers is communicated using IP addresses which identify the source and destination of packets. For network operators and administrators who want to monitor network traffic going to or from certain domains, this is a problem because internet domains can have multiple servers with different IP addresses and a single IP address can also host multiple services. For example, a cloud storage provider could geo-replicate data on multiple

servers around the world in order to provide better response time to clients. The need to monitor network traffic by domain name goes beyond just general interest. Network operators can use information on traffic associated with particular domains to identify traffic coming from potentially malicious domains or identify abnormal spikes in traffic going to or from certain domains.

In order to capture traffic of interest on a network, network operators often are required to collect a large amount of extraneous traffic in order to pinpoint traffic that is moving to and from a particular host or a particular domain. This could potentially compromise user privacy by collecting and examining more network traffic than technically needed [10]. In addition, this process is unnecessarily difficult and requires network operators to be concerned about low level details (IP addresses rather than domain names).

Intentional network monitoring is a concept that stresses the ability for network operators to monitor traffic according to higher levels of abstraction. The goal of intentional network monitoring is to give network operators the ability to monitor traffic according to high-level specifications such as a domain name, an individual user, or an individual device [10]. The essential goal of this project is to implement intentional network monitoring by domain name on a programmable network switch. There are a number of potential advantages of this approach. For one, matching traffic to domain names essentially requires matching DNS traffic to subsequent network traffic. Within the data plane, the act of joining information from DNS traffic to follow on network traffic can be done without exposing individual user IP addresses to network operators since that sensitive information never needs to be reported outside of the data plane, thus helping to preserve user privacy. In addition, since packet or IP address data does not need to be shipped outside of the data plane, a data plane implementation is vastly more efficient. The final and potentially most powerful reason is that monitoring traffic by domain name at the network switch level allows us to take action on packets while they are in the network. In other words, an implementation at the switch level would allow us to dynamically rate-limit, block, mark, or otherwise treat traffic differently by its associated domain name.

As a brief summary, the three primary benefits of performing network monitoring by domain name in the data plane as opposed to the control plane are:

- Privacy preservation
- Efficiency
- Direction action on packets in the network

As a practical example, if a network operator wanted to block traffic from known malicious domains or from domains associated with unwanted content (ads, for example), they could make use of a program that monitors traffic by domain name in order to accomplish this goal. This particular application is similar to the concept of a DNS sinkhole that blocks traffic coming from a set of blacklisted domains [7]. An implementation on a programmable switch, however, would have the added benefit of being more flexible in terms of the actions it can take on packets as suggested above.

There are a number of challenges, however, with creating this implementation within a programmable network switch in the data plane. For one, P4, the de facto language for programming in the data plane, does not behave like normal general purpose programming languages and is inherently limited in it terms of its ability to store and perform operations on variable length strings of data (such as domain names) due to the hardware limitations of the switches that P4 runs on. Operations that are usually taken for granted in normal programming languages are not possible in P4, especially when run on a hardware switch. For example, comparing two variables, accessing a register array twice, storing strings of bytes, and performing ternary matches all are limited or not allowed within P4 programs when run on hardware.

Ultimately, the implementation we created, dubbed "P4 NetAssay" was successful in identifying packets by their associated domain name. By using explicitly different parsing states for different length domain names and by implementing hash-table-like data structures inspired by past P4 projects, we were able to overcome the limitations of P4. Experiments run with real campus traffic from Princeton University showed that overall, P4NetAssay is able to collect relatively

accurate statistics on campus traffic as well as identify packets by domain as they go through the programmable switch.

2. Problem Background and Related Work

2.1. NetAssay

NetAssay is a system that was developed to allow for intentional network monitoring by domain [10]. The principle behind NetAssay was to match network traffic to domain names by examining DNS traffic. DNS requests are made by clients to translate human-readable domain names into IP addresses. As shown in steps 1 and 2 of Fig. 1, NetAssay examines DNS packets to match domain names to server IP addresses. Then, when a client sends an HTTP request to that server and the server responds, NetAssay is able to match this traffic with the domain name that it identified from the DNS packet.

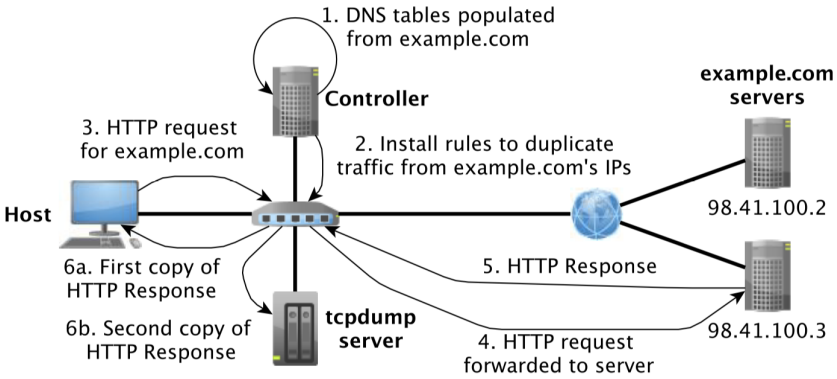

Figure 1: NetAssay metadata engines at the controller take DNS responses redirected from network switches and can create domain name - server IP address mappings. These mappings are then used to copy and redirect traffic from domains of interest to a network operator. (Diagram by Donovan, Feamster [10]).

The NetAssay runtime is theoretically designed to allow for intentional monitoring beyond just mapping IP addresses to domain names. For example, BGP routing tables can map Autonomous System numbers to IP prefixes and 802.1X servers can map usernames to device MAC addresses.

However, for the purposes of this paper, we are only concerned with intentional monitoring as it relates to associating domain names with their corresponding server IP addresses.

The way that NetAssay has been implemented is that the process of network monitoring occurs, in part, outside of the network itself. The primary contribution of this project is to implement all of the network monitoring within the data plane, allowing network operators to dynamically take action on packets while they are in the network.

2.2. Privacy Concerns and Limitations of Control Plane Network Monitoring

It should be noted that existing forms of intentional network monitoring have a few drawbacks in terms of both privacy preservation and in terms of usefulness to network operators. For one, by using DNS traffic to match network traffic to domains in the control plane, it is extremely easy for network operators to match individual client IP addresses to domain names which makes it a potentially trivial task to associate internet traffic with a specific domain with a specific individual or small group of individuals. The advantage of implementing domain name network monitoring on a programmable switch is that the act of matching DNS traffic to other packets is done within the data plane, out of sight of network operators, thus preserving the privacy of individuals and their browsing behavior.

2.3. The P4 Programming Language

Programmable switches are a potentially powerful way to implement intentional monitoring by domain by allowing programs to parse, extract information, and take action on packets directly as they pass through network switches. P4 is the most commonly used language for programmable switches and has been used to create network applications that detect and measure microbursts, anonymize personally identifiable information in packets, identify the OS of end-hosts on a network, protect against outside surveillance, and more [9].

P4 is quite different from most general purpose languages like C, Java, or Python in that it is a domain-specific language that is designed to control packet forwarding in network devices [4]. Most P4 programs consist of four parts: a declaration of various packet headers and structs, a parser, one

or more controls, and a deparser. The packet headers and structs are essentially structs and variables that are passed from block to block and can be accessed in the parser, controls, or deparser. These structs are often used to store metadata as well as data that has been parsed from a packet. The parser takes in a packet and is responsible for sequentially parsing through it. P4 is very flexible in that it protocol-independent and has no native support for even common protocols like IP, Ethernet, TCP, DNS, etc. Instead, P4 puts gives the programmer complete control on how to traverse and parse a packet. Control blocks allow a programmer to manipulate packet headers and metadata. Often, control blocks contain match-action tables that allow a P4 program to take certain actions on packets based on certain attributes of that packet. Finally, many P4 programs contain a deparser which re-constructs a packet to be outputted [11].

While P4 is a very powerful way to examine, process, and manipulate packets in a network, it also can be difficult to work with as the fact that P4 runs on a network switch means that it is inherently limited and many of the basic assumptions that programmers often take for granted in general purpose languages do not hold true for P4. Furthermore, the specific hardware of the network switch being used may also place further limitations on what can or cannot be done in a P4 program. Some limitations specifically encountered on this project include the inability to parse through a variable number of bits, limitations on the sizes of elements in P4 registers (analogous to arrays in Java or C), limitations on the total bits that can use a ternary match in a match-action table, lack of support for operations on variables with a variable size (strings, like domain names, for example, are very difficult to deal with), and a lack of native data structures.

2.4. Existing DNS-Related P4 Projects

As hinted at above, the lack of support for variable length parsing of variables in P4 makes dealing with DNS packets very challenging. Domain names, which are stored in DNS packets, are by their very nature, variable length, human-readable strings. Furthermore, in order to store mappings between domain names and IP addresses, we need support for a hash-table-like data structure which is not natively supported in P4. Thankfully, a number of P4 projects have already been created that

have encountered and come up with creative solutions for a number of these problems.

2.4.1. Barefoot P4 Authoritative DNS Server The Barefoot P4 Authoritative DNS Server was a project to essentially create an in-network DNS cache that can respond to a DNS request from a client faster than an external DNS server [12]. This project is useful for our purposes because it also requires a parser that processes DNS packet headers. In particular, the Barefoot DNS server, in addition to the related project P4DNS [13], implements a solution to deal with the need to parse a variable length domain name string. Domain names usually contain periods that separate different sections of the domain name. DNS packets will indicate the number of characters in a domain name "label", allowing the Barefoot parser to parse the number of characters expected in a domain name label and then use a pre-defined parser state to deal with that specific length of characters. This implementation solves the issue of parsing a variable length string of data, but comes at the cost of potentially not being able to parse domain names that might have abnormally long labels. The Barefoot implementation is able to parse domain names up to the size shown in Fig. 2.

Figure 2: Diagram by Xiaozhou Li [12]

2.4.2. PRECISION PRECISION is an algorithm implemented on a P4 programmable switch that uses probabilistic recirculation to find heavy flows on a network switch. While not directly related to DNS packet parsing or intentional network monitoring, PRECISION is useful in that it provides an implementation of a hash-table-like data structure in P4 which allows us to use P4 register arrays in order to store key, value pairs with a small chance for collision or inefficient space usage. The PRECISION table works by keeping three stages of parallel register arrays. Each set of register arrays contains a register array for each key and value. Three different hashes of the keys are computed which correspond to different indices for each of the three sets of register arrays. If the first hash yields an index that corresponds to an empty entry in the first table, then the program uses that entry. If that entry is full, the program tries the second and then the third if necessary.

In PRECISION, if all three indices for all three tables are full, then the program probabilistically evicts an existing entry [8]. The concepts behind the PRECISION table data structure are helpful for storing key, value pairs in our project as well. In particular, we use the main concepts behind the PRECISION data structure to store domain name/IP address pairings.

3. Approach

The main design consists of a P4 program that parses domain names from DNS response packets and matches them to server IP addresses contained within the DNS packet answer fields. Domain names will then be matched to a pre-determined known list of domains specified by a user. Matchings of client IPs, server IPs, and domain names will then be stored in a table in P4. Then, as subsequent packets are processed, the P4 program will use the table to match packets to domains and then count up the DNS packets, total packets, and total bytes of traffic associated with each domain name.

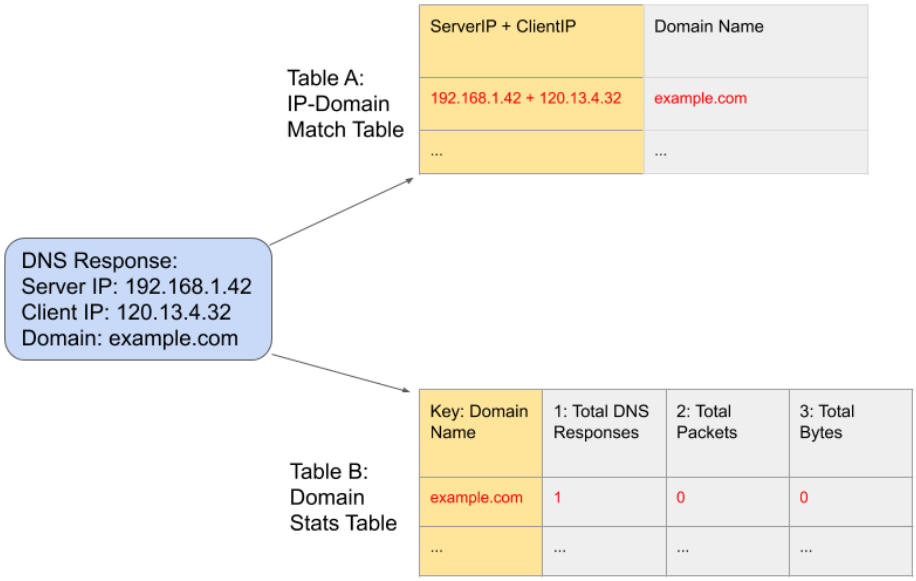

Figure 3: Overview of how P4NetAssay should behave when dealing with a DNS response. Server IP and Client IP and domain name are stored as an entry in the IP-Domain Match Table. The entry for the domain in the Domain Stats Table increments the total number of DNS responses for this domain.

Fig. 3 shows how P4 NetAssay should behave when it sees a DNS response packet. P4 NetAssay should extract the client IP, domain's server IP, and the domain name. This data should then be

stored in the IP-Domain Match Table where the server IP and client IP serve as a key and the domain name is the value in the table entry. Then, in the Domain Stats table, the program should update the entry for this domain (example.com) by incrementing the total number of DNS responses seen for this domain.

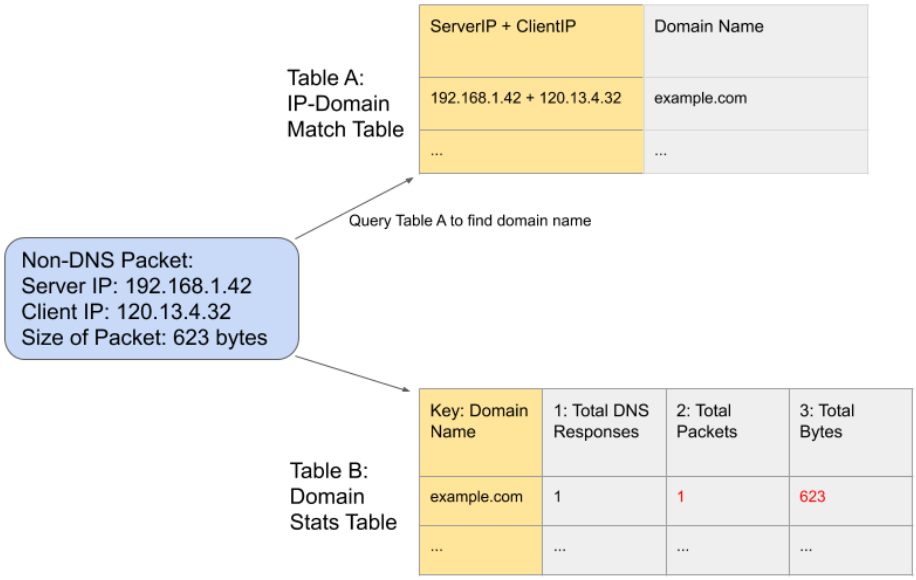

Figure 4: Overview of how P4NetAssay should behave when dealing with a non-DNS packet. Server IP and Client IP are used to key into the IP-Domain Match Table and find out which domain this packet is associated with. The domain name is then used to key into the Domain Stats table where P4 NetAssay updates the total packets and total bytes seen for this domain name.

Fig. 4 shows how P4 NetAssay should then behave when it sees a non-DNS packet. P4 NetAssay should extract the server IP and the client IP (these will just be the source and the destination IPs though not necessarily in that order). These IP addresses are then used to key into the IP-Domain Match table and read the domain name associated with this pair of IP addresses. This domain name is then used to key into the Domain Stats table where P4 NetAssay updates the total packets and total bytes seen for this domain name.

Fig. 5 shows again how P4 NetAssay behaves when it sees a DNS response packet. As before, it creates a new entry in the IP-Domain Match Table. Note that there may exist multiple entries in the IP-Domain Match Table for a single domain. This is because there can exist multiple clients requesting the same domain and there can also exist multiple different servers for the same domain.

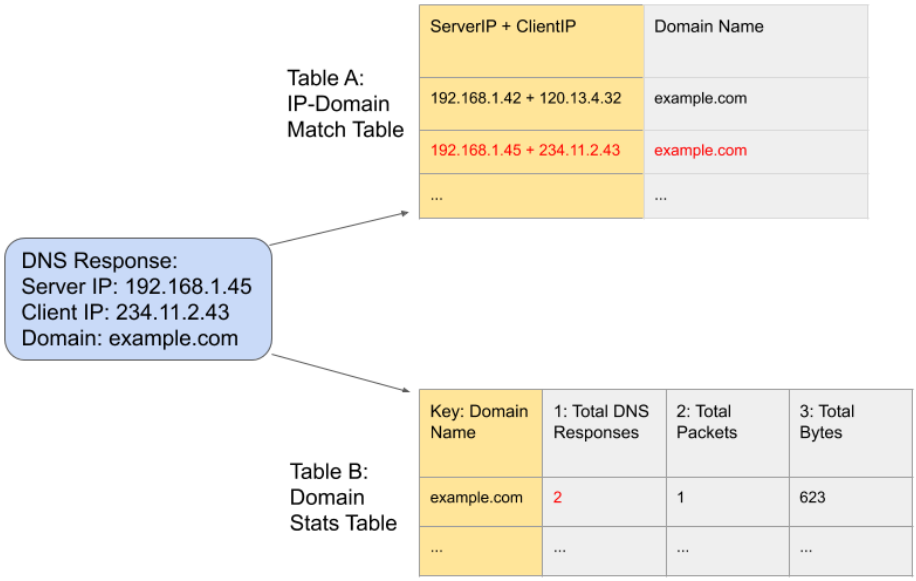

Figure 5: P4 NetAssay behavior when interacting with another DNS response packet for example.com

4. Implementation

This section will cover an in-depth discussion of the implementation of P4 NetAssay. The primary points of discussion will be the parser and how it was made to parse variable length domain names, the main data structures of P4 NetAssay, and how non-DNS packets are processed.

4.1. Parsing Variable Length Domain Names

The parser is the first portion of the P4 program and is responsible for extracting key information from packets. The primary challenge in this phase of the implementation was parsing variable length domain names in DNS response packets. P4 parsers are usually meant to extract information from packet headers which are usually fixed-length. Domain names, on the other hand, are human-readable, variable-length strings of bytes which can be difficult to extract in P4.

The parser first extracts the Ethernet frame of the packet and checks the EtherType field to ensure that it is 0x0800 (corresponding to an IPv4 packet). The parser then extracts the IPv4 header which includes the destination and source IP addresses of the packet. Note that if this packet is a DNS response packet, the destination IP corresponds to the client IP. Next, the program checks the IPv4

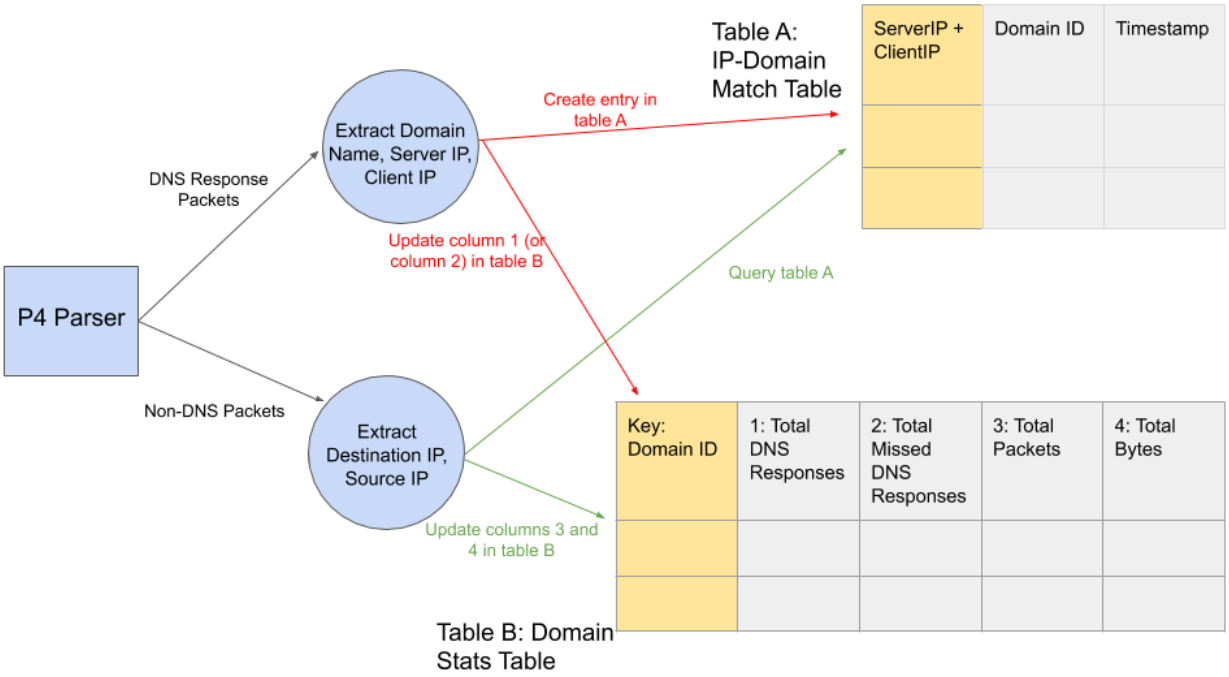

Figure 6: Overview of the P4 NetAssay implementation

protocol number and the source port. A protocol number of 17 corresponds to UDP [5]. DNS traffic usually uses the UDP protocol with a source port of 53 from the DNS server. If the packet matches these values, the parser continues. If the packet does not match these values, we treat it as a non-DNS packet and move on to the control block of the P4 program.

At this stage, the parser then moves on to parse the DNS packet itself. DNS packets contain five sections or fields: the header, question, answer, authority, and additional fields. For the purposes of this project, we are only concerned about the header, question, and answer. The header contains general metadata about the DNS packet, the question contains the domain name that the original query requested, and the answer contains the IP address of the server associated with that domain. The only piece of information we need from the DNS header is a single QR bit that indicates if the DNS packet is a query (0) or a response (1). We are only looking for response packets as they are the only packets that actually contain both the domain name and its associated server IP address [2].

If the packet is a DNS response, we then proceed to the DNS question field. The DNS question has three parts to it: the QNAME, QTYPE, and QCLASS. The QNAME is where the domain name is stored. Fig. 7 shows how domain names are encoded in a QNAME field.

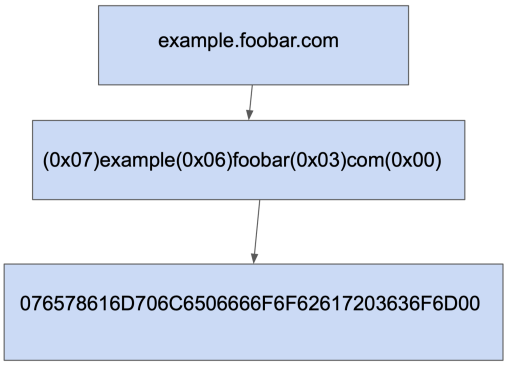

Figure 7: DNS questions store the requested domain name by separating the domain name into "labels", the parts of the domain name separated by periods. Each label is preceded by an octet that indicates the number of characters in that label. The last label is followed by the octet 0x00 which indicates that there are no more characters in the domain name. This diagram shows how "example.foobar.com" is encoded in a DNS question field.

The problem with parsing a domain name is that it is inherently a human-readable, variable-length field which does not translate well into a P4 parser. P4 is designed primarily to parse fixed-width packet headers and is not often used to actually parse through the contents of a packet, much less the human-readable aspects of a packet. One potential solution that was initially considered was the varbit base type that is offered in P4 [3]. Varbits are a data type in P4 that have a dynamically computed width. In other words, unlike other P4 data types, the width of a varbit is set at runtime. However, the primary problem with using varbit is that it is not a supported data type in many P4 hardware switches. In addition, the varbit is extremely limited in terms of the operations that can be performed. The only operations that can be performed on a varbit is extraction from a packet, assignment to another varbit, and insertion into a packet being constructed [3]. Without the ability to actually read the contents of the varbit, the functionality is extremely limited.

The solution that was ultimately implemented was based on the parser from the Barefoot P4 Authoritative DNS Server [12]. The idea is to have a fixed maximum number of labels allowed in a domain name and to parse different label widths in different parser states. For example, for "example.foobar.com", the parser would first read the 0x07 octet and then switch to a state that reads the next seven bytes into a fixed width variable of 7 bytes. Then, the parser reads the octet 0x06 and switches to a state that specifically reads in the next 6 bytes. The parser then reads 0x03

and switches to a state that reads three bytes. The parser then reads 0x00 and then terminates the domain name parsing phase. Theoretically, we could encompass all possible domain names by allowing for an extremely large number of labels and allow the max width of a label to be parsed (255 is the max width of a label). However, as we will see in section 4.3, limitations in P4 switch hardware limit the possible width of a domain name. For now, note that the actual implementation used allows for a maximum of 4 labels of up to 15 characters each. If a domain name does not meet these restrictions, it will not be successfully parsed and the information from this particular packet will not be used.

After parsing the domain name from the DNS packet question, the parser moves on to find the server IP address associated with that DNS packet. The DNS answer field consists of a variable number of answers. Each answer is formatted as shown below in Fig. 8.

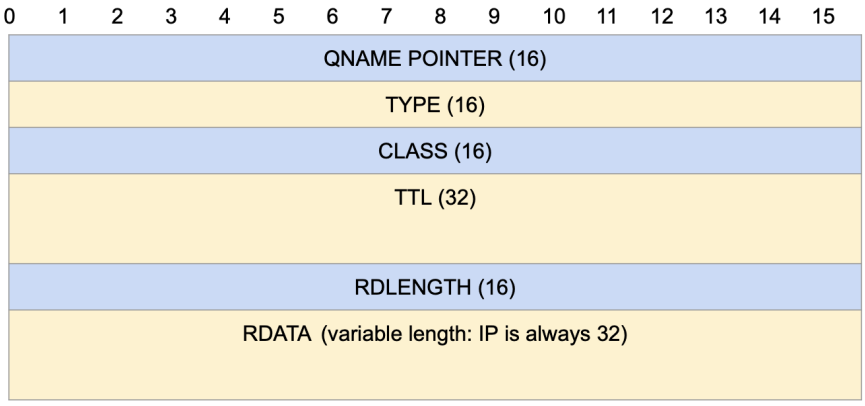

Figure 8: DNS answers first contain a 16 bit pointer to the QNAME from the DNS question field. Next is a 16 bit code to the type of answer (common types include 0x0001 for A record, 0x0005 for CNAME, 0x0002 for name servers, and 0x000f for mail servers [2]). The 16 bit class field specifies the class of the data in the RDATA field. This field is usually 0x0001, corresponding to Internet addresses. The TTL is 32 bits which determines the number of seconds the results can be cached. The RDLENGTH is 16 bits and gives the length in bytes of the RDATA field. THE RDATA field is how many number of bytes indicated by RDLENGTH.

DNS response packets may contain multiple answers. Most often, a DNS packet may contain a number of CNAME entries before giving a server IP address answer. CNAME entries identify a requested domain as an alias for another domain and then identifies the DNS entry for the domain

given by the CNAME. In the P4 implementation, the parser goes through the DNS answer fields and ignores any CNAME entries until it finds the first A record (IPv4 address record) which is indicated by a type of 0x0001. The parser then extracts the RDATA field which corresponds to the server IP address. Note that a DNS response may actually include multiple A records. In this implementation, we extract only the first IP address since it is difficult for a P4 parser to parse a variable number of records. Most clients (82% of clients and 91.5% of packets as determined from an anonymized Princeton network trace) only use the first IP address from a DNS response packet. In a future iteration of P4 NetAssay, the parser could be made to extract all IP addresses from the DNS response by recirculating the packet through the switch.

4.2. Known Domain List and the Match Action Table

As indicated in Fig. 6, the next step after parsing a DNS response packet is to store the server IP, client IP, and domain name into a table entry. The problem is that it is again quite difficult to deal with a variable length domain name and store it into a P4 register which only accepts pre-defined, fixed-width entries. The solution used was to have the user specify a known list of domain names that they are interested in monitoring. That way, we can equate domain names to ID numbers and represent and store the domain names as integers. This also allows us to group multiple domain names under a single ID which is useful if there are multiple domains that belong to the same service (e.g., aaplimg.com and apple.com which are both associated with the same family of services). In our specific tests as described in section 5, we used a known list of the domains that comprised the top 50% of traffic by both bytes and packets in the traces captured on the Princeton network. In addition, we also included domains from the list of top 500 domain names [6] in our known list. The known list implementation allows for wildcard labels in the domain names. For example, "*.example.com" allows for any domain with three labels that includes "example.com" as the last two labels. Furthermore, "*.*.*.com" allows for any 4 label domain that ends in "com". The inclusion of wildcarding also allows for domains that belong to similar services or the same company to also be grouped together (e.g., m.youtube.com and youtube.com). Users

can specify their domains in a list in order of precedence that they want the domains to be matched (e.g., *.google.com should be matched above *.*.com).

In the P4 program, a match action table is used to match domains from DNS response packets to domains within the known list. If a match occurs, an ID number is returned which is used to refer to that domain within the P4 program. Note that ternary matching is used to allow for wildcards in the domains. The problem is that P4 hardware switches often have a set limit on the number of bits that can be used in a ternary match action table. This unfortunately restricts the P4 implementation to allow the parser to only parse domains of up to four labels of 15 characters each as noted in section 4.1.

4.3. Implementing the IP-Domain Match Table

At this point, we have the client IP (the destination IP address for the DNS response packet), the server IP (the RDATA field from the DNS response asnwer), and the domain name (in the form of an integer ID from the match action table). We need to store these results in a table keyed by the client and server IP addresses where the value is the domain name ID integer. We need to use a hash-table-like data structure that will allow for efficient look-ups. In addition, we have to be aware that existing data structures within P4 (particularly P4 registers) must have a limited, pre-defined length and cannot be expanded during runtime. The solution in this case is to create a data structure based off of the PRECISION table described in section 2.4.2 [8].

The principle behind this data structure is to hedge against the possibility of hash collisions by having four "stages" of P4 register tables. The client IP and server IP are hashed in four different ways (all four hashes use crc16 but with differing salts added to each hash input). The hashes give four indices for the four different stages. The program first tries the first stage. If an entry already exists, the program tries the second, then third, then fourth. If an empty entry is found at any stage, the program stores the client IP, server IP, domain integer ID, and a timestamp in the entry. If all four stages are tried and are full, then the client IP/server IP pair for this domain cannot be stored. Instead a separate register that contains counters for the number of missed DNS responses for each

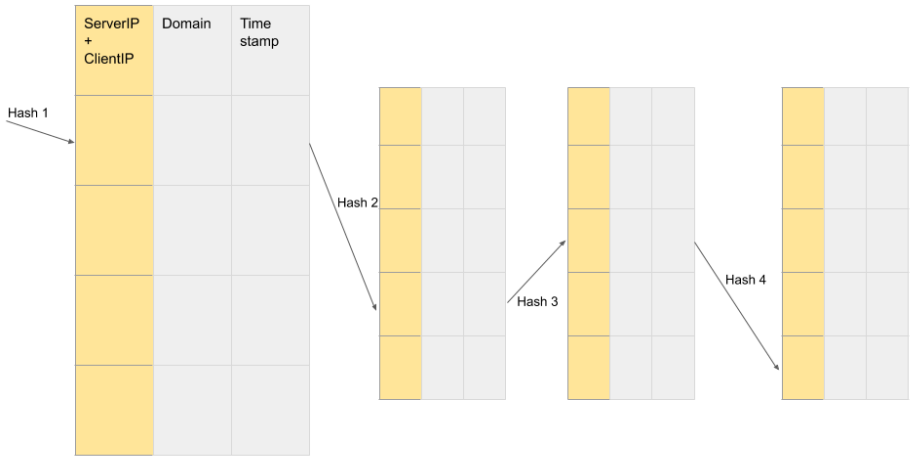

Figure 9: Four P4 register tables are used, identical in format (the first is expanded to show the table header labels). Four different hash functions are used to hash the server IP/client IP to an index in each table. If an entry is not open in the first table, the second table is tried, and so on.

known list domain is incremented. This register will be used later to estimate the amount of traffic that was not counted by the P4 program (see section 4.4). This at least allows us to estimate the amount of lost traffic for this particular domain. Note that the timestamp field is updated every time a table entry is accessed and used (either by a DNS or non-DNS packet). If a timestamp is over 5 minutes old, the entry is considered open and available for re-use. The principle is that if a client has not interacted with a server for over 5 minutes, their session is likely over and the IP address-domain match entry can be forgotten. Specifically, 5 minutes or 300 seconds is usually how long most operating systems store DNS response results [1].

4.4. Actions taken on non-DNS Packets

Most of the discussion so far has been related to actions taken on DNS response packets. In this final subsection, we will discuss actions taken on normal, non-DNS packets. As mentioned in the discussion of the parser in section 4.1, the parser extracts the source and destination IP addresses for all packets. In order to query the IP-Domain Match table, we need to be able to identify if the destination or source IP address is the client IP or the server IP. One possible solution to this issue is to simply try both matches. First assume that the source IP is the server IP and the destination IP is the client IP and hash the results and attempt to key into the IP-Domain Match table. If this

does not work, try assuming that the source IP is the client IP and the destination IP is the server IP and hash the results and attempt to key into the table again. The problem with this solution is that it requires reading P4 registers more than once which is not allowed within a single stage in a P4 program when run on hardware.

The solution that we ultimately used was to always hash the concatenation of the server IP and the client IP in the order of the larger value first. IPv4 addresses are 32 bits and thus can be represented as 32 bit integers. Whichever IP address is the larger value when represented as a 32 bit unsigned integer is the first in the concatenation. The concatenation of the two addresses is then what is hashed to be the index where entries are stored in the IP-Domain match table. Then, when we are dealing with non-DNS packets where we are not sure if the destination or source IP corresponds to the server IP or client IP, we can just determine if the source or destination IP is the larger value when represented as a 32 bit unsigned integer and hash the concatenation of the two addresses in that order to key into the table.

Once we key into the table, we can identify which domain name ID corresponds to this packet. We then use that result to update values in the Domain Stats table as shown in Fig. 6. The index in this P4 register table corresponds to the domain name ID. For each packet, we want to increment the number of total packets for this domain as well as update the total number of bytes of traffic based on the size of this packet.

As noted in section 4.3, we also keep track of the number of DNS responses that were unable to be stored for each domain within the IP-Domain Match table. Using the ratio of the number of missed DNS responses to the number of total DNS responses for each domain, we can estimate the true number of total packets and total bytes of traffic associated with each domain name. The accuracy of this estimation is assessed in section 5.1.2.

5. Evaluation

5.1. Evaluating Limitations of the P4 Implementation

As discussed in the previous section on the P4 NetAssay implementation, there are a number of limitations in the P4 program that may cause it to not accurately keep track of all network traffic. The two primary limitations are the parsing limit and the DNS response table memory limit. Note that the length of the Domain Stats Table from Fig. 6 is not a concern since it has a fixed length (the number of domains in our known list). After creating the P4 implementation, a number of experiments were run on an anonymized trace from the Princeton network to assess the performance of the P4 implementation and evaluate the severity of these limitations.

5.1.1. The Parsing Limit The parsing limit refers to the limitation in that the P4 parser can only parse domains that fit a certain specification. Specifically, due to the limit of the amount of bits that can be used within a ternary match match-action table (as described in sections 4.1 and 4.2), the parser only allows for domains of up to four labels of 15 characters each. In other words, any domain that follows the specification of *.*.*.* where each * can be up to 15 characters. For example, foo.bar.example.com would be allowed but nationalgeographic.com would not be allowed. To assess how much traffic is being lost due to the parsing limit, we created a python version of the P4 NetAssay implementation. This implementation is not limited by the parsing limit or the DNS response table memory limit. We then artificially applied the parsing limit to assess the amount of traffic being lost using the anonymized trace from the Princeton network.

Fig. 10 shows the results of the experiment. The X axis shows the maximum number of bytes allowed in a domain name under the parser limitation using four labels. For example, a value on the X axis of 60 refers to a parser limit of four labels of 15 characters each ($4 \times 15 = 60$). The results of the experiment show that around a limitation of 70 bytes, the parser is able to capture all traffic. Note that the current implementation has a maximum width of 60 bytes. Under this limitation, 93% of DNS traffic is successfully captured, 85% of packets are successfully captured, and 80% of bytes of traffic are successfully captured.

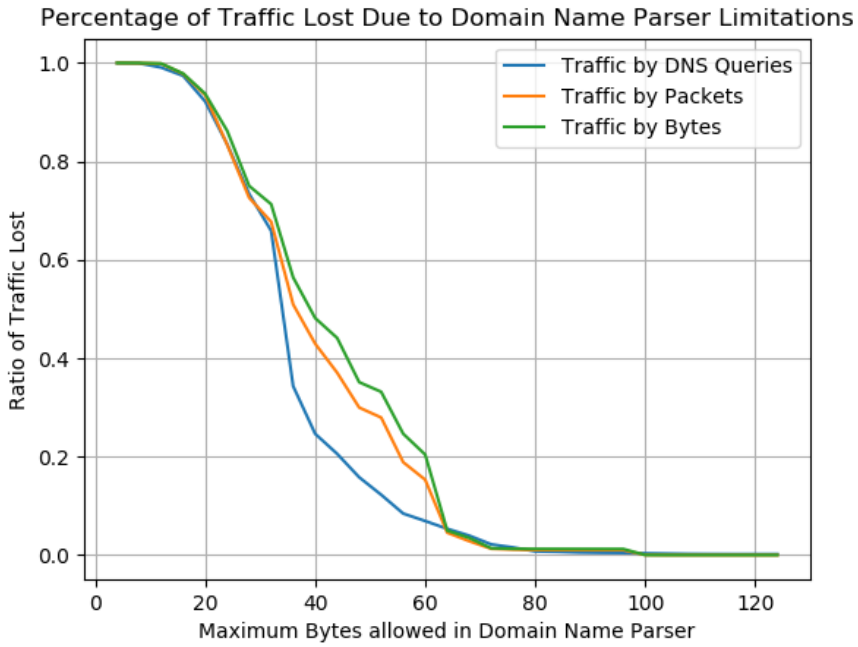

Figure 10: The X axis shows the maximum width of a domain name allowed in bytes in the parser and the Y axis shows the amount of traffic being lost with 1.0 meaning that all traffic is lost and 0.0 meaning that no traffic is lost. Three curves, for DNS queries/responses, packets, and bytes are shown.

5.1.2. The DNS Table Memory Limit The other main limitation of the P4 implementation is the capacity of the PRECISION-Style table used to store server IP/client IP matchings to domain name IDs (see section 4.3). As mentioned above, there are four different stages in this table. The actual length of each of the registers that make up each stage is an arbitrary value that must be empirically derived. In order to derive the proper length of each register to minimize the amount of lost traffic, the P4 implementation was run multiple times on the same anonymized Princeton network trace with varying register lengths.The relative error of the amount of traffic (in bytes and packets) was calculated for all the domains in the known list. These relative errors were calculated from true values determined by the python implementation described in section 5.1.1.

As seen in Fig. 11, at around a length of 16000, the median relative error in the number of bytes of traffic approaches around 0.

The following graph in Fig. 12 tell a similar story. The four graphs show the CDF of relative error for four different table length configurations. As can be seen, about 90% of domains have a

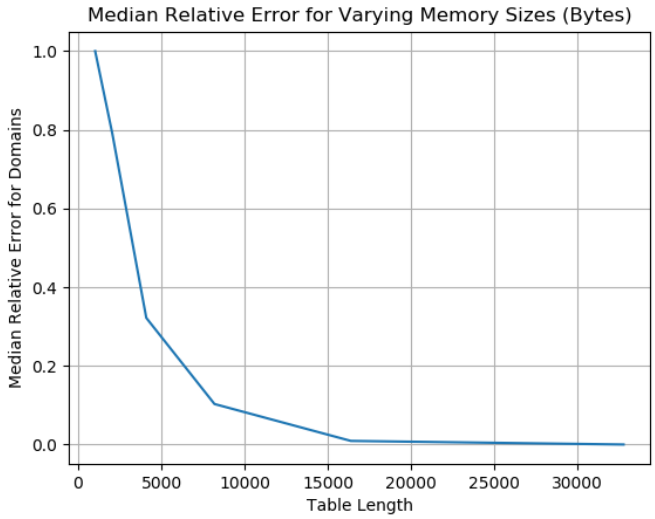

Figure 11: Median relative error in the number of bytes of traffic for differing table lengths.

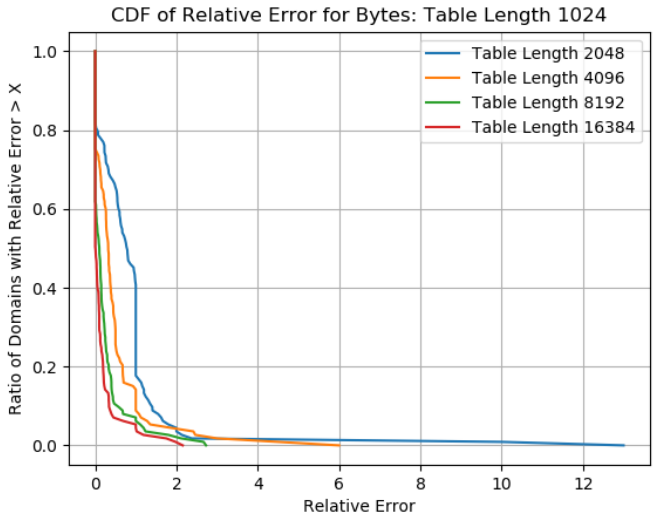

Figure 12: CDF of relative error of the number of bytes of traffic for domains for differing table lengths.

relative error of below 1.0 in the 2048 and 4096 table length configurations. About 90% of domains have a relative error of below 0.5 in the 8192 table length configuration. In the 16384 table length configuration, about 90% of domains have a relative error of below 0.25.

While the limitations of the P4 implementation cause a slightly inaccurate characterization of the network traffic it sees, as seen in these two experiments, the P4 implementation is still relatively accurate as a whole in assessing and determining the volume of traffic for a known list of domains in a network. Once the table length is large enough, even if DNS responses are occasionally, missed,

the correction operation used allows for almost nonexistent relative error.

5.2. Characterizing Princeton Network Traffic

The final task was to actually use the P4 implementation to characterize the network traffic captured in the anonymized Princeton network trace. Included below in Table 1 is a list of the top domains (by bytes of traffic) and the amount of traffic associated with each.

In addition, to see how much of the total traffic is contributed by the top domains, a CCDF of traffic volume and domain rank (where rank is determined by the domains that contribute the most traffic) was created as seen in Fig. 13. This graph shows how the top domains contribute the vast proportion of total traffic. The top 20 domains contribute about 90% of all traffic.

Domain Name	Total DNS Queries	Total Packets	Total Bytes
..com	6751	57269	44779753
..*.com	1955	17858	12948470
..*	2163	13976	11942436
..org	293	9919	11597882
..net	1128	10243	9640016
*.instagram.com	9	7794	7733679
..llnwd.net	4	6891	7224908
..fbcdn.net	49	5641	6096455
www.google.com	352	3526	2592811
*.ytimg.com	23	2427	2569622
*.office365.com	1384	5958	2375578
*.twimg.com	27	2405	2329962
*.bing.com	74	2211	1890266
*.gstatic.com	617	2600	1828662
..*.net	1052	3195	1798220
*.net	108	1953	1730014
*.twitch.tv	12	1730	1461939
..*.*	260	5277	1378039
*.amazon.com	48	3054	1348721

Table 1: Top domains by bytes from anonymized Princeton network trace

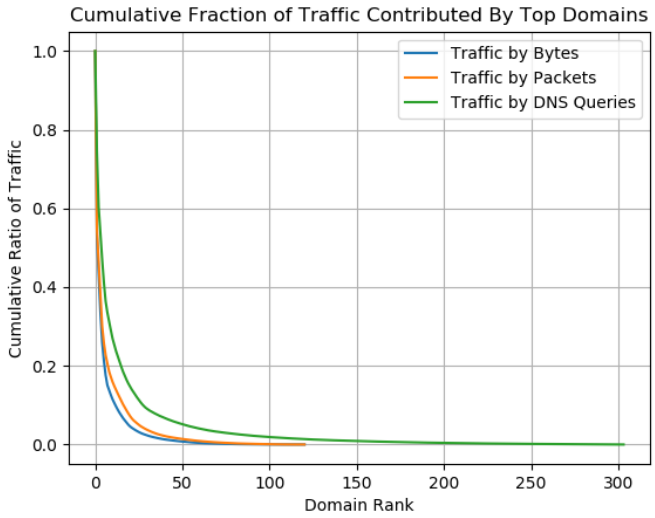

Figure 13: CCDF of total traffic volume contributed by top domains.

6. Future Work

As suggested before, while overall successful, there are still a number of limitations with the P4 implementation. As such, there are a number of possible directions for future work. First, the current limit of 60 characters (4 labels of 15 character each) for domain names could potentially be expanded if multiple match action tables were used instead of a single match action table. It is possible that separate match action tables could be used for each label of a domain and the matching results of each match action table could then be combined to determine if a domain matches a domain from a known list.

Another issue is with the use of a known list of domains at all. The use of a known list inherently limits the power of traffic analysis because network operators have to specify a list beforehand of domains that they are interested in monitoring. One possible future project could create an implementation that builds off of the known domain list implementation by dynamically updating the known list by subdividing "heavy-hitters". For example, as seen in Table 1, *.*.com and *.*.*.com make up the two top domains in terms of traffic. However, these two domains don't actually tell a lot of interesting information about network traffic. If the P4 program was able to dynamically create new known domains and add them to the list based on past traffic, a greater specificity in domain traffic information could be achieved.

Another potential direction is to actually use the P4 program to take action on certain packets based on their associated domains. As suggested in section 3, the power of implementing NetAssay in the data plane is the ability to take action on packets while they are in the network. The P4 program could, for example, be easily modified to drop packets that are associated with malicious websites or unwanted content (ads for example).

Another project that is tangentially related to this original project is to use DNS query information and patterns of packet size and behavior to fingerprint webpages. Currently, P4 NetAssay allows us to match packets to domains, but one realization made during this project is that it is potentially possible to use patterns of DNS queries and the frequency and size of packets to actually determine the webpages and specific content that a user is viewing. This project could potentially be used maliciously to violate user privacy but it could also be a powerful way to combat the dissemination of illegal content on the web.

7. Conclusions

This project implemented the principles behind intentional network monitoring to the data plane by creating a P4 implementation of NetAssay. This allows network operators to specify a known list of domains to monitor the quantity of traffic in terms of DNS requests, total packets, and total bytes associated with those domains. In addition, the creation of this implementation in the data plane theoretically allows for network operators to take action on certain packets based on their associated domains. Thus, if network operators want to rate-limit or block packets associated with certain domains, this P4 implementation allows network operators to take such actions on packets in a way that is not normally possible with most forms of network monitoring.

8. Acknowledgments

I want to thank Professor Jennifer Rexford and Dr. Hyojoon Kim for being incredible helpful and always willing to offer insightful guidance and feedback on my project. Even with complications regarding a move to an online semester, Professor Rexford and Dr. Kim were very patient, under-

standing, and encouraging. Even with the current events of the semester, I had a great IW experience and am incredibly grateful.

9. Honor Code

This paper represents my own work in accordance with University regulations. - Jason Kim

References

[1] (2002) Configuring domain name system cache storage. [Online]. Available: https://www.itprotoday.com/cloud-computing/how-can-i-configure-how-long-dns-cache-stores-positive-and-negative-responses

[2] (2016) Computer networks 365: Lab 4: Domain name system primer notes. [Online]. Available: https://www2.cs.duke.edu/courses/fall16/compsci356/DNS/DNS-primer.pdf

[3] (2017) P4 16 language specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[4] (2019) P4 language consortium. [Online]. Available: https://p4.org

[5] (2020) Assigned internet protocol numbers. [Online]. Available: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

[6] (2020) The moz top 500 websites. [Online]. Available: https://moz.com/top500

[7] (2020) Pi-hole network-wide ad blocking. [Online]. Available: https://pi-hole.net

[8] R. B. Basat *et al.*, "Efficient measurement on programmable switches using probabilistic recirculation," *IEEE ICNP*, 2018. Available: https://p4campus.cs.princeton.edu/pubs/precision_icnp_paper.pdf

[9] J. Brassil *et al.* (2019) P4 campus: P4 applications for campus networks. Available: https://p4campus.cs.princeton.edu/index.html

[10] S. Donovan and N. Feamster, "Intentional network monitoring: Finding the needle without capturing the haystack," in *ACM SIGCOMM HotNets Workshop*, 2014. Available: http://conferences.sigcomm.org/hotnets/2014/papers/hotnets-XIII-final102.pdf

[11] S. Ibanez, B. O'Connor, and M. Arashloo. (2018) P4 language tutorial. Available: https://github.com/p4lang/tutorials

[12] X. Li, "P4 authoritative DNS server," *Barefoot Networks*, 2017.

[13] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4DNS: In-network DNS," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2019.