Analyzing Traffic by Domain Name in the Data Plane

Jason Kim
Princeton University, USA
jjk7@alumni.princeton.edu

Hyojoon Kim
Princeton University, USA
hyojoonk@cs.princeton.edu

Jennifer Rexford
Princeton University, USA
jrex@cs.princeton.edu

ABSTRACT

Associating network traffic with human-readable domain names, instead of low-level identifiers like IP addresses, is helpful for measuring traffic by domain name, rate-limiting packets by domain, and identifying IoT devices. However, existing monitoring techniques require examining traffic at an external compute node, introducing overhead and privacy risks. In this paper, we introduce *Meta4*, a framework for monitoring traffic by domain name in the data plane by extracting the client IP, server IP, and domain name from DNS response messages and associating the domain name with data traffic from the subsequent client-server session. A data-plane implementation has the benefits of running efficiently at line-rate, enabling the switch to take direct action on the packets (*e.g.*, to rate-limit, block, or mark traffic based on the associated domain), and protecting the privacy of user information. We implemented Meta4 on an Intel Tofino switch and evaluated our prototype against packet traces from an operational network.

CCS CONCEPTS

• **Networks** → **Network management**; **In-network processing**; **Programmable networks**.

KEYWORDS

DNS, Domain name, P4

ACM Reference Format:
Jason Kim, Hyojoon Kim, and Jennifer Rexford. 2021. Analyzing Traffic by Domain Name in the Data Plane. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3482898.3483357

1 INTRODUCTION

Network monitoring usually groups related packets based on low-level identifiers like IP addresses, MAC addresses, or port numbers. Ideally, network operators should be able to monitor traffic by defining higher-level policies that match closer to their actual intent. For example, network operators should be able to specify a policy like *"count all traffic related to video-streaming services"*, rather than worrying about the low-level details of which flows are associated with various streaming services. That is, video-streaming services should be identified by their *domain names* (e.g., www.youtube.com and *.netflix.com), rather than IP addresses.

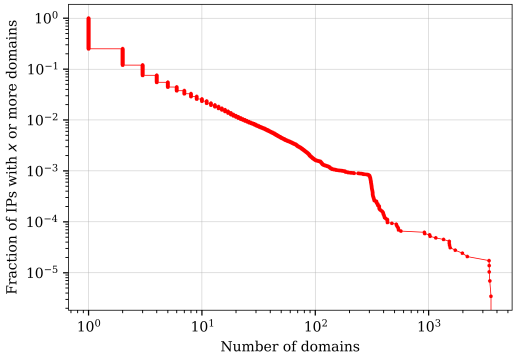

Figure 1: Complementary cumulative distribution function for the number of unique domain names associated with each server IP address over a week.

Unfortunately, monitoring by domain name can be expensive, because the headers of data packets contain IP addresses rather than the associated domain name. Plus, many domain names can map to the same IP address. To illustrate this problem, we analyzed a week of DNS response messages from our campus network. Figure 1 shows a complementary cumulative distribution function (CCDF) of how many unique domain names are associated with a particular server IP address. Out of 288,834 observed server IPs, around 25% of them are associated with multiple unique domain names in DNS responses over the course of the week. Some IP addresses, corresponding to web hosting platforms and Content Distribution Networks, are associated with hundreds or even thousands of domain names in the trace. Clearly, a static, one-time mapping of domain name to server IP address would associate data traffic with the wrong domain names.

Instead, the monitoring platform needs to "join" each DNS response message with the subsequent data traffic. For example, NetAssay [9] sends copies of DNS response messages received at an OpenFlow switch to an external controller, which then parses the packets to determine the IP address associated with each queried domain name (*e.g.*, 172.217.12.206 for youtube.com). Then the controller monitors the subsequent YouTube traffic by installing a match-action table rule that matches on the client and server IP addresses. Such an approach, however, relies on *off-switch* processing to find the IP address for every domain name. In this paper, we lay the groundwork for implementing domain-based traffic monitoring *within the data plane*, without the controller playing any role in parsing and mapping domain names to IPs.

The advent of programmable data planes allows us to parse and process packets without having to capture, store, and analyze large

SOSR '21, October 11–12, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/10…$15.00
https://doi.org/10.1145/3482898.3483357

amounts of traffic at a separate server. By parsing DNS response packets in the data plane, we can extract the client's IP address (the destination IP address of the DNS response), the server's IP address (found in the answer of the DNS response), and the domain name requested by the client (found in the query field of the DNS response). By joining the client IP and server IP pair with the domain name in the data plane, we are able to track client-server sessions by the associated domain name. Here, we define "session" as all the traffic a server/domain pair sends to a client for a given time *after* a single DNS lookup. A given session may in fact consist of multiple TCP connections over time due to DNS response caching (*e.g.*, browser caching) by the client.

Taking advantage of a PISA (Protocol Independent Switch Architecture) data plane, we introduce *Meta4*, a framework for domain-based monitoring in the data plane. Meta4 has several benefits over *off-switch* approaches. First, data-plane programs naturally process packets much faster than servers, allowing us to monitor large amounts of traffic without compromising performance. Second, implementing the analysis in the data plane allows the switch to take *direct* action on packets based on monitoring results, without waiting for an external controller to act. For example, a network operator can run a P4 application that uses Meta4 to detect IoT devices that communicate with a suspicious domain name and immediately re-route or block traffic, all performed in the data plane. Third, as domain-based monitoring happens in the data plane, there is no need to export raw packet traces to an external server. This provides better user privacy as DNS response packets, which are highly sensitive, and follow-on data packets, stay in volatile memory in the data plane rather than on a non-volatile disk space on an external server. Moreover, authorization and authentication for accessing switches and routers are usually more restrictive than servers, making them generally more secure.

However, implementing Meta4 in the data plane is challenging. PISA-based switch hardware imposes a number of restrictions that make implementing Meta4 difficult. For one, PISA switch parsers in real hardware are designed to parse fixed-length header fields, which makes dealing with variable-length domain names difficult. Furthermore, there are significant limitations in register memory as well as limitations on the amount of processing that can be done per packet due to limited stages in the processing pipeline. To parse variable-length domain names, we leverage the octet prefix that indicates the length of each domain label. We then combine different parser states with different parsing widths to match the indicated domain label length (Section 4.1). To efficiently utilize the limited switch memory, we use a multi-stage register data structure for storing the client IP, server IP, and domain information. We use a timeout value for overwriting stale entries when hash collisions occur (Section 4.2).

In this paper, we use a generic use case for presenting Meta4: measuring traffic volume by domain name. Network operators can define domain-based "traffic classes" such as "Netflix video" or "Zoom video chat" and monitor the traffic volume (in bytes or packets) sent by servers to clients for domains associated with these traffic classes. We implement this use case in P4 and run it on Intel's Tofino [14] programmable data plane. Our code is open-source, publicly available on Github [17]. We run Meta4 against two different packet traces captured at a campus network and present our performance evaluation in Section 5.

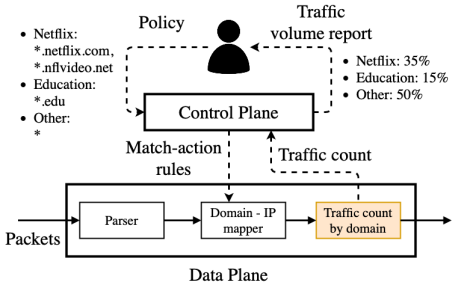

Figure 2: Meta4 architecture

Although measuring traffic volume by domain name is our running example, domain-based monitoring in the data plane allows network operators to create a variety of different applications that can run in the switch. With this vision in mind, we design Meta4's core functions, which are domain name parsing and matching follow-on traffic to domain names accordingly, to fit in the ingress pipeline in the PISA switch. This leaves the egress pipeline for developers to implement and run a variety of applications that use the domain name as a primitive. For example, we present a DNS tunneling detection use case in Section 7.1. By identifying clients who routinely make DNS requests with no subsequent traffic, we can easily identify client IP addresses that are suspects for using DNS for nefarious purposes. Another useful application is IoT device detection and fingerprinting. In Section 7.2, we show that by identifying domain names commonly associated with particular IoT devices, we can fingerprint user devices, or client IP addresses, that routinely exchange traffic with domains (and using particular TCP/UDP ports) that uniquely identify a type or model of IoT device.

We envision many other applications could make use of domain-based monitoring in the data plane. For instance, to improve network performance, a network operator could allow traffic from certain trusted domains to bypass processing by an intrusion detection system (IDS). Similarly, a network operator could create firewall rules based on domain name to redirect or drop packets associated with certain untrusted domains.

2 META4 DESIGN CHALLENGES

In this section, we provide an overview of the architecture of Meta4 and the challenges presented by trying to create a data-plane implementation. In Section 2.1, we start by describing Meta4 through the most basic use case where we use the IP address-domain name mapping to identify traffic associated with a particular domain name and track its traffic volume. In Section 2.2, we describe the challenges with implementing Meta4 in a PISA-based data plane.

2.1 Meta4 Architecture

The overall architecture for a Meta4 application is shown in Figure 2. The system is defined by three levels: the network operator, the control plane, and the data plane. In this specific application, a network operator wishes to measure traffic volume by certain traffic

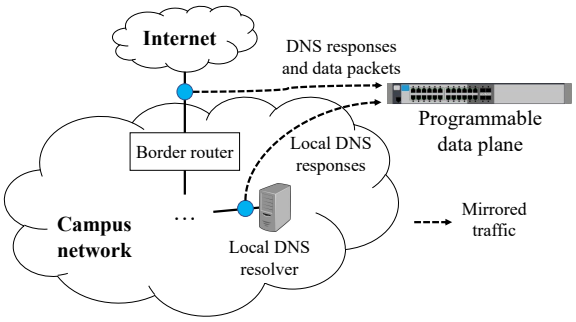

Figure 3: Required packet traces for Meta4.

classes. For instance, in Figure 2, the network operator wants to compare video streaming traffic from Netflix and education-related traffic. The network operator uses these traffic classes to define a known domain list: a list of domain names that encompass the traffic sources of interest. This list of domains is then converted into match-action table rules by the control plane which are then used by the data plane to decide if the packets it sees are relevant to the network operator's intended policy.

Using DNS response packets, the data plane parses domain names and uses the match-action rules to create Domain to IP address mappings. These IP address mappings are then used to identify the domains associated with the subsequent data packets. The data plane stores information like the number of bytes/packets seen for each of the traffic classes in registers. When the network operator wants to see the traffic volume results, the control plane polls the register values to retrieve the number of bytes/packets for each traffic class. The control plane can then interpret this data and report it to the network operator in terms of the original policy that was defined. Note that the control plane and the operator can only view the traffic count by domain as an aggregate, not the DNS queries made by individual clients or their associated traffic volumes.

Figure 3 shows the packet traces for Meta4 to output the most accurate analyses for traffic between an enterprise network and the Internet. Note that the programmable data plane should be able to see the traffic that goes between the enterprise network and the Internet, as well as the DNS response packets going to the clients in the enterprise networks. Thus, the programmable data plane should either be deployed in-line at a location in the network that can see all such traffic or have all relevant traffic mirrored to it. An alternate approach is to extract domain names from the Server Name Indication (SNI) included in TLS handshakes. The SNI field, however, turns out to be much more challenging to parse and extract within the data plane due to many variable-length and long "extensions" that often appear in TLS `Client-Hello` packets. Thus, we focus on using DNS response packets and leave other methods as future work.

2.2 PISA Implementation Challenges

The PISA-based switches, which contain a programmable packet parser, processing pipeline, and a packet deparser, enable us to implement Meta4 in the data plane. To preserve line-rate packet processing speed, however, there are limited computations and memory when parsing and processing packets in a PISA-based switch, as well as a limit on the number of pipeline stages. There are two main challenges under the capability and resource constraints of PISA hardware: (1) correctly parsing domain names and mapping flows to them, and (2) accurately collecting statistics.

Correctly parsing domain names and mapping flows: First, variable-length domain names have to be parsed correctly. Packets traverse a programmable packet parser, which extracts header fields and metadata from a packet. The processing pipeline can then use such metadata to make particular decisions for the packet. However, parsers in the latest PISA-based switches are yet not versatile enough for handling variable-length packet-header fields with ease. While the parser can make different parsing decisions by switching to different parser states based on header field values, it cannot parse through a variable number of bits. For Meta4, this poses a challenge with parsing domain name fields from DNS response packets since domain names are, by nature, variable-length strings.

The limited memory for match-action tables also poses constraints when mapping flows to domain names. Within the processing pipeline, a program can make use of match-action tables, which can match on packet-header fields and metadata that are extracted at the parsing stage. Upon a match (or miss), the program can apply some action: a forwarding decision, an update to header or metadata values, or an update to a register, which is a stateful memory in the data plane. In Meta4, match-action rules provide a natural way for us to match domain names from a DNS packet to check if a domain name (*e.g.*, netflix.com) belongs to one of our predefined traffic classes (*e.g.*, video streaming). However, match-action tables consume TCAM/SRAM space, and the data plane has a limited amount of memory. Such constraints limit the number of domain names we can store in the match-action table, but more importantly, it limits the total length of a domain name that we can match on.

The existence of variable number of CNAME entries, which can appear before the A record in DNS responses, also poses a challenge. The case-sensitivity of a domain name has to be dealt with, too; for example, `foo.com` and `FOO.com` should be considered the same.

Accurately collecting statistics: PISA-based switches have the ability to preserve some state in persistent memory across multiple packets by way of registers, or register arrays. Such register arrays can be used for keeping state for <client IP, server IP, domain ID> tuples from every DNS response packet in the switch. Within the processing pipeline, however, each stage contains a limited number of register arrays of limited and fixed width [15]. Furthermore, on a PISA switch, a program can only access (read/write) an entry in a register array once per packet. This naturally causes difficulties with many programs for PISA switches that need to perform more complex operations. Limitations in the total amount of memory available in the form of register arrays mean that it is likely impossible to store all <client IP, server IP, domain ID> tuples from every DNS response packet in the switch. This means that we n need a solution to intelligently evict stale entries to allow room for new DNS response entries if we want to collect statistics accurately against the traffic.

3 RELATED WORK

Intentional network monitoring. The basic design and functionality of Meta4 was inspired by intentional network monitoring, pioneered by the NetAssay system [9]. One of the specific applications of NetAssay, in particular, dealt with using DNS packets in order to match subsequent packets to their associated domain names, an idea that we implement in the data plane with Meta4. NetAssay itself was also inspired by the concept of intentional naming in networks. The Intentional Naming System or INS [4] allows systems to route traffic based on higher-level intent, such as a domain name, as opposed to a lower-level identifier like IP or MAC address. Meta4 differs from NetAssay in that it is implemented entirely in the data plane. We also present multiple use cases that take advantage of the information extracted from DNS response packets.

Parsing domain names. Previous work in developing solutions for parsing variable-length domain names in PISA-based switches helped us design and implement Meta4. P4DNS [25], an in-network DNS server in P4, parses different domain names in different, predetermined, fixed-length parser states. This results in having many parser states and metadata fields in different sizes to cover various domain names with different lengths (*e.g.*, 1 to 60 bytes or characters). The P4 Authoritative DNS Server [16] expands on P4DNS's idea and parses the separate labels of a domain name (the parts of the domain name delineated by periods) into separate fixed-length parser states. This reduces the overhead with metadata fields since each metadata field length is now limited to a single label. We further expand on these ideas in designing our domain name parser for Meta4 to optimize the efficiency and performance of our parser. In particular, we store domain name labels in header fields by combining only four different metadata fields with varying sizes: 1, 2, 4, and 8 bytes. This minimizes the amount of metadata we needed to store while also allowing us to maximizes the length of domain labels we could parse. Meta4 also deals with CNAME entries and case-sensitivity in domain names, which has not been done by previous works. Section 6.1 describes this in more detail.

Data structure implementation in P4. PRECISION [5] and HashPipe [23], heavy-hitter detection programs in P4, provided the idea of separating a register data structure into separate stages to hedge against the possibility of hash collisions. In addition, HashPipe also proposed the idea of evicting table entries based on some sort of counter. Specifically, HashPipe evicts table entries for flows that have a low packet count value. We expand on these ideas for our implementation of our DNS response table to create a multi-stage data structure. We handle hash collisions differently: Meta4 evicts entries for having timestamp values that had not been updated for a certain period of time.

4 MONITOR BY NAME IN DATA PLANE

In this section, we describe Meta4's design. We use a canonical application as a running example: a Meta4-based program that measures traffic volume in terms of bytes and packets sent by server to clients by domain name. First of all, Meta4 treats DNS response packets and data packets differently.

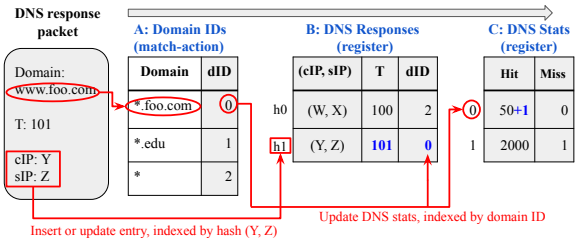

Figure 4: Storing IP addresses from DNS response. A is a match-action table whereas B and C are registers.

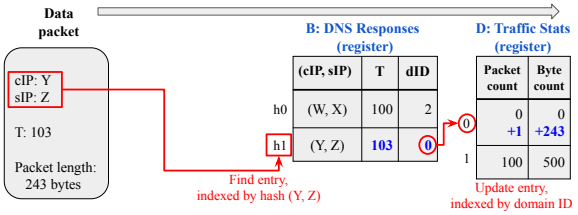

Figure 5: Updating traffic counts for data packet. Table B and D are registers.

For DNS response packets (Figure 4): Meta4 parses and extracts the domain name that was queried by the client. The extracted domain name is then matched against a domain *name-to-ID* match-action table, which is pre-populated with the domains the operator wants to track. By doing so, the data plane can now label and track domain names using a numerical identifier (dID) instead of using a string of characters. The server IP address (sIP), which is the IP address for the queried domain name answered by the DNS resolver, appears after the domain name string. Meta4 parses and extracts this address, too. Meanwhile, the client IP address (cIP), which is the destination of the DNS response packet, is extracted from the IP header. Meta4 then stores the <cIP, sIP, T, dID> tuple in the register, where T is the packet timestamp. Meta4 also keeps track of whether an entry for the incoming DNS response was able to get inserted into the register memory. If inserted, it is recorded as a *hit* in table C, otherwise a *miss*.

For data packets (Figure 5): Data packets are exchanged between the client and server *after* the client learns the server IP address for its queried domain name in the DNS response. When data packets arrive at the data plane, Meta4 extracts the source and destination IP addresses and uses them find a matching entry in the register memory, table B. When a match is found, the data packet and its bytes are counted towards the matching domain.

Meta4 is designed so that it is easy to "plug-in" a new domain-based monitoring application. Thus, core tables like A, B, and C are all implemented in the ingress pipeline while application-specific parts are in the egress pipeline. For example, table D, which is for monitoring traffic volume by domain name, is implemented in the egress pipeline.

We now describe how we tackle hardware limitations in parsing (Section 4.1) and memory (Section 4.2). We also present a simple method for correcting for missed DNS responses (Section 4.3).

4.1 Parsing Long, Variable-Length Names

Extracting variable-length domain names from DNS response packets is a challenging task in hardware since it requires significant flexibility in parsers. This is one of many things that is easily done in a software switch (*e.g.*, P4 behavioral model [20]), but not in PISA hardware.

In order to successfully parse domain names, it is important to understand how DNS encodes domain names. Domain names are stored within the DNS "question" field which separates the domain name into "labels", the parts of the domain name delineated by periods. For example, the labels of "example.foobar.com" are "example", "foobar", and "com". Within the DNS question field, each domain label is preceded by a single octet which gives the length, in characters, of the label. The last label is followed by the octet 0x00 which indicates that there are no more labels in the domain name. Thus, the domain "example.foobar.com" would be encoded as "(0x07)example(0x06)foobar(0x03)com(0x00)".

In order to parse domain names in Meta4, we take advantage of the octet prefix indicating the length of the subsequent domain label in order to parse a domain name in fixed-length segments. We fix the maximum number of labels allowed in a domain name and parse different label widths in different parser states in order to comply with the PISA parser restriction of parsing only fixed-length header fields. For example, for "example.foobar.com", the parser first reads the 0x07 octet and then transitions to a state that reads the next seven bytes into a fixed-width variable. Then, the parser reads the octet 0x06 and transitions to a state that reads the next six bytes. The parser then reads 0x03 and transitions to a state that reads three bytes. The parser then reads 0x00 and then transitions out of parsing the domain name.

Theoretically, this solution would allow us to parse any domain name up to 255 bytes, which is the max width of a label. However, in reality, memory such as TCAM is expensive, limiting the max width of table A in Figure 4. This drastically restricts the size of domain names that can be used in our programs. In our P4_16 implementation on the Tofino switch, we limit parsing domain names to a maximum of 60 bytes. Moreover, we use four labels with 15 characters each, which covers most of the domains we observe in our network. Note that different domain parsing configuration could work better on certain domain names. For example, four labels with 20, 30, 5, 5 characters respectively could work better for domains such as "www.nationalgeographic.com", which tend to have labels with many characters followed by short labels like "com" or "org". It turns out, however, that it is better to maximize the number of labels that we can parse instead of the maximizing the longest length for a particular label we can parse. We further discuss this in Section 5.1.

A DNS response may include multiple CNAME entries before the A records appear. Meta4 jumps through these entries using the parser counter feature in Tofino [19]. We describe this in more detail in Section 6.1. A DNS response packet may also include multiple A records, thus providing multiple IP addresses for a given domain name. Meta4 parses only the first IP address in the returned list. We assume most web browsers will behave similarly, which seems to be the default behavior of modern browsers (*e.g.*, Chromium [8]).

4.2 Efficient Memory Usage

Avoiding storing domain names in registers. Since register memory has even more limited width than match-action tables, it is difficult to store large and variable-length strings like domain names. To address this issue, Meta4 takes advantage of the domain names the operator provided through the control plane (Figure 2). An integer ID is associated with each domain name or traffic class. Using a match-action table, table A in Figure 4, we can match parsed domains to domains in the known list. If a match occurs, a domain is represented as its integer domain ID corresponding to its traffic class, allowing us to store a 32-bit integer and not the entire domain name. All references to a domain name in subsequent data structures use this domain ID as opposed to the full domain name. The domain ID is also the array index for the traffic statistics table in Figure 5 (table D), which stores the accumulated packet/byte-count for each traffic class.

Multi-stage register data structure. The <cIP, sIP, T, dID> tuple, extracted from a DNS response packet, is stored within a register data structure known as the DNS response table, which is table B in Figure 4 and Figure 5. The purpose of this data structure is to associate a data packet with a certain domain by matching the data packet's source and destination IP address to the client IP and server IP address in table B. Ideally, this data structure would accommodate an unlimited number of entries. However, due to the register memory constraints in PISA-based hardware, a large amount of DNS response traffic could quickly fill up the register, causing hash collisions between new entries and existing entries.

To avoid hash collisions as the table fills up, we implement a solution inspired by the multi-stage data structure used in PRECISION [5]. The basic premise is to divide the data structure into multiple registers. If a hash collision occurs at one register, the program then tries the next register with a hash with a different salt and so forth until it finds an available entry or until it checks every register and fails to insert. While this method uses the same amount of memory as a single big register, dividing the register into multiple sets can promote a more efficient use of register memory by hedging against the possibility of hash collisions, giving new entries multiple chances to find an empty entry.

Freeing register entries. Even with more efficient register usage, the register data structure inevitably fills up as more traffic goes through the switch. To help free up table entries, the program makes table entries available once it has determined that a client-server session is no longer active. This is determined by maintaining a timestamp value for each table entry. When the table entry is initially created by a DNS response packet, the table entry is marked with the packet's timestamp as seen in Figure 4. Subsequent packets that belong to the same client-server session are used to update the timestamp as seen in Figure 5. If a new table entry encounters a hash collision with an existing entry and the old entry's timestamp is older than a predetermined timeout value, then the new entry is allowed to occupy that space in the table. To avoid unnecessarily evicting table entries, we lazily evict timed out entries. In other words, we do not evict an entry until a hash collision occurs. We further discuss about the appropriate timeout value in Section 5.2.

Packet resubmission. In reality, table B in Figure 4 and 5 is composed of multiple registers. This is because a <cIP, sIP, T, dID> tuple entry is too "wide", making it challenging to fit into a single register. In Meta4, we allot three separate parallel registers to store the tuple entries: register <cIP, sIP>, register <T>, and register <dID>. The hash index is identical among them for an entry in table B. This, however, poses a new issue for a PISA-based hardware. For evicting an entry, the program first reads the register <cIP, sIP> to see if the existing IP addresses in the entry match those of the new DNS response packet that indexed to that entry. If the IP addresses match, we can just update the timestamp <T> register and move on. However, if the IP addresses do not match, then we have to check the timestamp value for staleness and decide if we want to evict the existing entry. If we decide to evict the entry, we then need to update both the timestamp <T> and the existing <cIP, sIP> addresses. All of these operations cannot be conducted in a single packet pass through the PISA pipeline, especially because the register <cIP, sIP> has to be read once in one stage but might have to be written to in a following stage depending on the read operation outcome from the timestamp <T> register. Thus, Meta4 performs a packet resubmission operation, allowing the packet to go through the ingress pipeline *one more time* to finish updates to corresponding registers. Because this resubmission only happens with DNS response packets and when an entry in table B has to be replaced, it has negligible effect on latency or bandwidth consumption on the switch.

4.3 Correcting for Missed DNS Responses

Meta4 has two potential sources of error. First, Meta4 might not be able to insert a new <cIP, sIP, T, dID> tuple entry in table B due to limited register memory and hash collisions. If this happens, Meta4 is not able to track subsequent data traffic associated with this new client-server session. The second source of error is premature eviction of an existing entry in table B. For example, assume a client and server pair pauses exchanging traffic for 120 seconds but resumes afterwards. If the timeout value was set to 100 seconds and a hash collision occurs for this entry between 100 and 120 seconds, it will be replaced with a new one. This premature eviction leads to not tracking traffic anymore between this pair.

These two sources of error tend to work against each other. If we increase our timeout value, table B may become bloated, causing new <cIP, sIP, T, dID> entries to experience more hash collisions and fail to be stored. On the other hand, if we reduce our timeout value to accept more new entries, we may prematurely evict old entries more often.

While reducing the second source of error is a matter of fine-tuning the timeout value, there are other methods to reduce the first type of error. We introduce a mitigating measure to avoid potential under-representation of packet/byte counts associated with certain domains due to missed <cIP, sIP, T, dID> entries. The main idea is to calculate an approximation of the proportion of total packets/bytes missed for a certain domain/traffic class. For this, we leverage the DNS statistics table, or table C, in Figure 4, which keeps track of the number of successful as well as failed attempts to insert or update an entry in table B. This register table allows us to calculate the proportion of DNS responses that we were not able to monitor for each domain. Using this proportion as a scaling factor, we can scale up the number of bytes and packets for a particular domain in order to provide a more accurate representation of the amount of traffic associated with each domain. For example, assume Meta4 saw 100 DNS response packets in total for "example.com" in a day, but was only able to store 80 of them in table B. In this case, table C will have 80 hits and 20 misses. Then, we scale the counts for "example.com" by 1.25 (100/80). Thus, if Meta4 reported seeing 100 data packets and 1000 bytes for this domain in table D in Figure 5, we correct them by multiplying 1.25 to each, revising the count to 125 packets and 1250 bytes. We further evaluate this correction method in Section 5.

5 PERFORMANCE EVALUATION

In this section, we assess how well Meta4 performs within data-plane constraints such as parsing limitations and register memory constraints. In order to provide a comparison for Meta4's performance under varying conditions, most of the experiments in this section are performed in a Python program that emulates Meta4's behavior in the data plane.

We use two packet traces, both captured from a campus network, for evaluating Meta4. The first trace is a three-hour trace that was captured on August 19, 2020 between 8:00am-11:00am local time. The trace has an average of 138,383 packets per second with an average 205 DNS responses per second. The second trace is a 15-minute trace captured during a period of heavier load. This trace was captured on April 7, 2020 between 3:00pm-3:15pm local time. The trace has an average of 240,750 packets per second with an average 2,151 DNS responses per second. Unless otherwise stated, all experiments are run on a three-hour trace. Note that the packet traces are collected during the global COVID-19 pandemic. Although the list of popular domains might slightly differ from pre-pandemic, we do not believe this would significantly affect the performance evaluation done in this section.

Ethics: All packet traces were inspected and sanitized by a network operator to remove all personal data before being accessed by researchers. The client IP addresses, or internal host IPs from our campus, are anonymized in a consistent manner using a one-way hash. The server IP addresses in DNS response packets are not anonymized since they are publicly available via DNS queries anyway. Our research was approved by our university's institutional review board.

5.1 DNS Response Parsing Limits

The first experiment assumes unlimited register memory and applies various parsing constraints to the domain name parser. We measure the number of DNS response packets that contain domain names that cannot be parsed under various parser configurations. We also measure the number of data packets and the number of bytes from those packets that are missed in case we are unable to first successfully parse the DNS packet. All domain name parser configurations were limited to a maximum of four labels with an equal number of bytes allocated to each label. In Figure 6, for example, 60 bytes on the x-axis corresponds to a domain parsing configuration of four labels with 15 bytes allocated to each label.

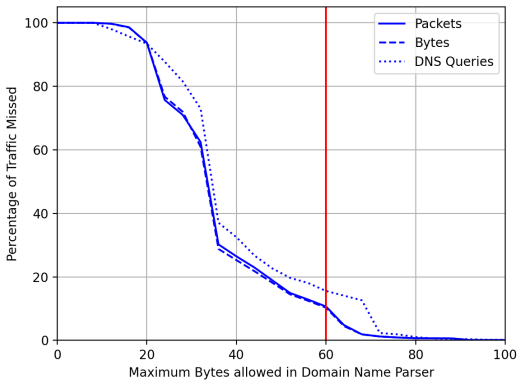

Figure 6: Traffic missed vs. max name length. The red line shows the max length in our Tofino prototype.

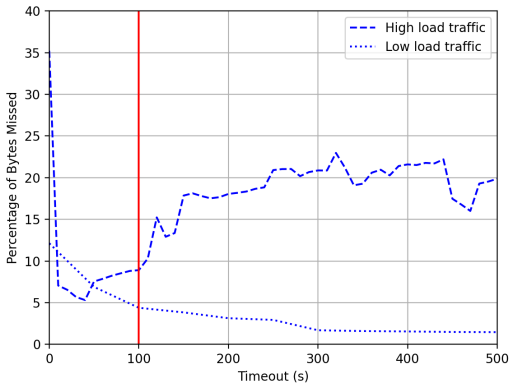

Figure 7: Traffic missed vs. DNS responses timeout. The red line is for our prototype.

Figure 6 shows the results of the experiment. In our Tofino implementation with the 60-byte limit, we are unable to parse about 18% of DNS queries/responses but only miss about 10% of the data traffic. This indicates that the domains we are not able to parse contribute proportionally less traffic than the domains that we are able to parse.

A closer inspection reveals that each individual domain that Meta4 cannot parse contributes, on average, only 0.0037% of DNS responses, 0.0049% of packets, and 0.0048% of bytes. Among the domains that Meta4 cannot parse, the greatest contributor in terms of byte and packet count was "m-9801s3.ll.dash.row.aiv-cdn.net": it contributes 4.0% of byte traffic and 3.4% of total packets. The greatest contributor in terms of DNS responses was "124.230.49.37.4vw5p-iqz5ksoolyszwje5tobsi.sbl-xbl.dq.spam-haus.net", which contributes 5.2% of total DNS responses. However, there were no data packets for this domain. This is because this DNS response is actually a spam blacklist [3] using the DNS protocol for convenience. Many of the DNS response packets that Meta4 cannot parse are from queries made by non-human clients, which is why they proportionally contribute less traffic than the domains we are able to parse. Overall, the vast majority of domain names that cannot be parsed contribute little or no traffic when inspected individually. Thus, Meta4 stands as an effective tool for monitoring and measuring major sources of traffic volume. We also envision that the next generation of switches will have fewer parsing restrictions.

5.2 Evaluating the DNS Table Timeout

We now look into how different timeout values for the DNS response table influence Meta4's accuracy when monitoring traffic. Recall that a long timeout value will prevent new <cIP, sIP, T, dID> entries from getting stored while a small timeout value might prematurely evict existing entries. Striking the right balance with the timeout value is imperative in the face of limited memory resources. In this experiment, we measure the amount of traffic, in bytes, that would be missed by Meta4 with varying timeout values. We limit the DNS response table to 2^{16} total entries and two stages, which we use in our Tofino implementation.

We use two different packet traces for this evaluation: the three-hour trace and the 15-minute trace. Notice in particular that traffic density of DNS responses is ten times greater in the 15-minute trace. This means that the frequency of DNS response packets, each creating a new <cIP, sIP, T, dID> entry, is tenfold in the 15-minute trace.

The results in Figure 7 for the three-hour trace suggest that a longer timeout is better for capturing more traffic. However, this is not true for the higher density 15-minute trace. Because the 15-minute packet trace has a significantly higher traffic rate, the DNS response table fills up rather quickly, causing many more hash collisions. Increasing the timeout at some point prevents stale table entries from being evicted, leading to more traffic being missed. Thus, we ultimately decided on a timeout of 100 seconds as a good balance that performs well under conditions of both high and low traffic density. A timeout of 100 seconds was also chosen for a number of structural reasons. For one, many browsers like Chrome [2] and Firefox [1] cache DNS results for a default of one minute. In addition, most users are likely to engage with a particular domain name in a single, sustained period of time where the biggest gaps of time come from the user's "think time" in between transactions with a server. A user's "think time" usually falls much under 100 seconds [21], allowing us to effectively capture traffic from most client-server sessions within our chosen timeout value.

5.3 DNS Response Table Memory Limits

We evaluate how Meta4 performs when varying the total amount of register memory and the number of stages. Note that varying the number of stages does not vary the total amount of memory; it just splits the total memory across stages. We first fix a timeout value to 100 seconds, based on the results in Section 5.2. We then assess the percentage of traffic missed, in bytes, under the varying memory configurations.

Figure 8 shows that at the memory limitation of 2^{16} entries, the total memory used in our Tofino implementation, we miss under 5% of traffic when using two stages. Also note that when so little traffic is missed, there are negligible, or even no, benefits to using

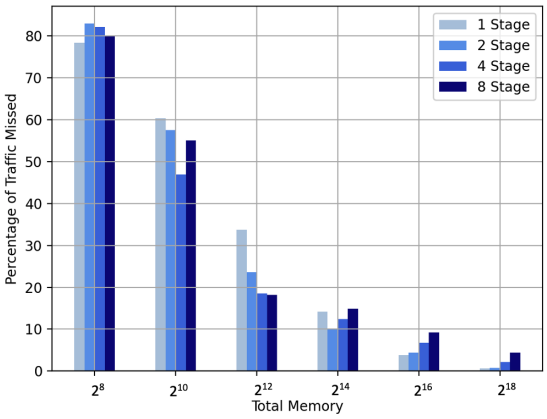

Figure 8: Traffic missed vs. memory size constraints

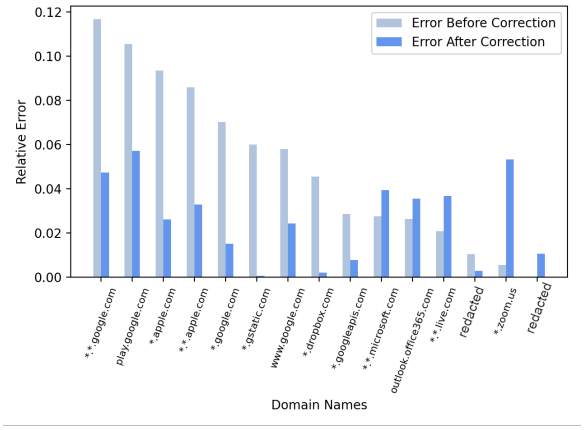

Figure 9: Relative error for the top 15 domains by DNS response count with 2^{16} max register entries.

more stages. However, for the memory limit of 2^{10} and 2^{12} entries, there are significant improvements with four stages as opposed to one stage. This occurs because the amount of traffic just begins to overwhelm the system when the limit is 2^{10} or 2^{12} entries. As the data structure fills up, the probability of hash collisions increases, thus providing extra stages to resolve hash collisions can provide a significant boost in the performance of the data structure. However, as we further increase the number of stages, the actual benefit of hedging against hash collisions starts to diminish due to smaller memory allocation per stage. For example, in the case of 2^{12} memory length, the greatest benefit comes from moving from one to two stages, but the gain from going to eight stages is rather small.

Through further analysis, we realized that our current Meta4 implementation does not fully utilize the extra stages. Our algorithm currently evicts the first timed-out entry that it encounters even if there might be a more stale or even a free entry in later stages. As shown in Figure 8, this leads to some instances where adding more stages actually worsens the performance because each stage's memory becomes smaller. Our Meta4 implementation for real hardware performs well in general when using 2^{16} table entries and two stages. Nonetheless, we believe there are opportunities to improve, and we leave this as future work.

5.4 Correction for DNS Response Misses

Finally, we evaluate the efficacy of the correction method described in Section 4.3. In a nutshell, we scale up the number of packets and bytes estimation for a domain based on the number of successful and unsuccessful insertions to the DNS response table for that particular domain.

In Figure 9, we show the relative error for the amount of bytes recorded for each of the top 15 domains by DNS count, both before and after our correction applied. We excluded domains with only a single defined label (domains like *.com, *.*.edu, etc). For this evaluation, we use the same memory configuration, 2^{16} max entries, and timeout value, 100 seconds, as our Tofino implementation. We see that the correction applied makes a significant difference for most of the domains, often cutting down the error by more than half. This improvement, however, is not universal. For instance, for the domains "*.zoom.us" or "outlook.office365.com", the scaling correction actually increases the relative error. This suggests that for some domains, the amount of packets and bytes transmitted to a client is not very consistent per DNS request. Since the scaling correction works by essentially determining the average number of packets and bytes a client receives per known session for a given domain, a domain that provides an inconsistent or essentially random amount of traffic per session with high standard deviation is not going to work well with the scaling correction: the correction can overcount or undercount significantly. This makes sense for services such as Zoom or Microsoft Teams, which provide varying amounts of traffic per user depending, among other things, on the length of a video call; this is rarely consistent between various client-server sessions. Domains that primarily serve webpages, on the other hand, are relatively consistent in the number of packets and bytes served per DNS request. While this scaling correction can be a useful tool, it is important to use it selectively on domains that are known to be consistent in the amount of traffic that they send per client-session and also only on domains for which Meta4 has been able to collect a large sample of DNS responses and traffic to make a more accurate scaling correction.

Parameter	Hardware Implementation Value
Domain Parser	4 labels, 15-byte each (60 bytes total)
DRT Timeout	100 seconds
DRT Length	2^{16} entries
DRT Stages	2

Table 1: Final parameters of Meta4 hardware implementation for Tofino switch. DRT stands for DNS Response Table.

6 PROTOTYPE AND DEPLOYMENT

In this section, we discuss our experiences implementing and deploying Meta4 in hardware. The final parameters of our hardware implementation are summarized in Table 1. Based on traffic lost both due to parsing limitations and memory limitations, this hardware configuration of Meta4 had a final percentage of traffic loss of

14.2%, by bytes, when run on our three-hour trace. We discuss our hardware implementation challenges and experience in Section 6.1 and Section 6.2. In Section 6.3, we share our experience with deploying our running example, measuring traffic volume by domain name, on our campus network. Our code is open-source, publicly available on Github [17].

6.1 Parsing DNS Packets

Dealing with CNAME entries. One issue with parsing DNS response packets that is specific to the hardware implementation is the challenge of dealing with CNAME entries. A DNS response providing IP addresses (A records) for a domain name often includes a variable number of preceding CNAME records that map a domain name alias to other domain names. As indicated above, it is a challenge to parse a variable-length field like a domain name. It is even more challenging to parse through a variable number of variable-length fields. Thankfully, each CNAME entry is preceded by an octet which gives the length of the CNAME record in bytes. We use this field to set a parser counter to the length of a CNAME record. A parser counter is a feature in Tofino [19], which allows us to skip through a CNAME field byte by byte while decrementing the counter until the counter reaches 0. We repeat this process for each CNAME record we find until we finally reach the A (IP address) record.

Case-sensitivity in domain names. Another problem we encountered is case-sensitivity of the domain name parser. In the packet traces used for evaluation and testing (see Section 5.2), a small portion of the domain names contained capital letters in unexpected places. To account for potentially unexpected capitalization, we edited the masks used for the ternary match-action table rules for domain names to make them case-insensitive. Domain names in DNS are encoded using ASCII, so the difference between capitalized and non-capitalized letters is just a single bit (the third most significant bit in the eight bit code). For example, the letter "a" is encoded as 01100001 and the letter "A" is encoded as 01000001, so by using match-action rule: 01*00001, we can match both on "a" and "A". It should be noted, however, that DNS response packets that contain capital letters are an anomaly, contributing only 1.6% of DNS response packets in our test traces. Furthermore, most of the <cIP, sIP, T, dID> tuples from these DNS responses had no follow-on traffic, contributing only 0.00044% of total traffic in terms of packets.

Encrypted DNS packets. Meta4 cannot parse encrypted DNS response packets that are exchanged through DNS-over-TLS (DoT) [12] or DNS-over-HTTPS (DoH) [11]. Thus, Meta4 ignores such packets. To gauge the amount of encrypted DNS response packets, we analyzed our campus traces to detect them. We used server port number 853 to detect DoT traffic. For detecting DoH traffic, we used server port 443 to capture traffic from well-known public DNS servers that support DoH, including Cloudflare (1.1.1.1, 104.16.248.249) and Google DNS (8.8.8.8, 8.8.4.4). In our 15-minute campus packet trace, there are 1,710 encrypted DNS response packets, which is around 0.09% of total (encrypted and raw) DNS response packets. For the three-hour trace, there are 27,938 encrypted DNS response packets, which account for 1.2% of total DNS responses. Our analysis shows that Meta4 does not miss many DNS responses in our campus trace.

	Domain Match Action Table	<cIP, sIP, D> Tuple Registers	DNS Queried / DNS Total Registers	Packet Count by Domain Register	Byte Count by Domain Register
Hash Dist Unit	5.6%	55.6%	0.0%	16.7%	16.7%
Logical table ID	6.3%	25.0%	0.0%	18.8%	9.4%
Meter ALU	0.0%	25.0%	25.0%	25.0%	25.0%
SRAM	0.0%	28.0%	2.5%	2.5%	2.5%
TCAM	66.7%	0.0%	0.0%	0.0%	0.0%
Exact Match Input Xbar	4.5%	15.4%	0.0%	1.6%	3.5%
Ternary Match Input Xbar	63.7%	0.0%	0.0%	0.0%	0.0%

Figure 10: Data-plane resource usage in Tofino

Monitoring traffic when encrypted DNS protocol is in use, however, is a general issue among network operators, not just for Meta4. We leave this as future work.

Combining four header field types for domain labels. As previously mentioned, we allow four domain labels, where each domain label can be up to 15 characters, or bytes, in our hardware implementation. In a parser state, Meta4 has to store each domain label into header fields, which are later used as match keys in the domain match-action table. One way is to use a single fixed max-sized header field (15 bytes) to store a domain label with any length. However, this wastes bit-space when a domain label is not exactly 15 characters. Another way is define 15 different header field types to cover every possible domain label length, which is also a suboptimal method. Instead, we only define and use four different header field types: 1, 2, 4, and 8-byte header fields. We then combine them to cover domain labels with any length. In fact, it is possible to store domain name labels of any length up to, and including, 15 bytes using only these four header fields, which are in powers of two. For example, if a domain label is seven bytes long, we stored it in the 1-, 2-, and 4-byte header fields. If a domain label is ten bytes long, we stored it in the 2- and 8-byte header fields. Header fields that are not used for a particular domain label are set to be invalid, which indicates that they do not represent any characters in the domain name. This solution allows us to maximize the number of domains we are able to parse while minimizing the total amount of TCAM in the match-action table we consume. Figure 10 shows TCAM resource consumption by the domain match-action table.

6.2 Utilizing the Limited Number of Stages

The processing pipeline for a PISA switch is restricted to a limited number of stages. A programmer is also limited by the arithmetic logic unit (ALU) to a restricted number of operations per stage. Furthermore, a packet cannot access or modify a register entry in a pipeline more than once. As noted in Section 4.2, this particular restriction is problematic for our <cIP, sIP, T, dID> tuple data structure. Thus, we utilize the packet resubmission feature, which allows us to resubmit specific packets to go through the ingress pipeline once more. In particular, we determine if a <cIP, sIP, T, dID> entry needs to be replaced on the first pass for a DNS response packet. If

so, we resubmit the DNS response packet and replace the old <cIP, sIP, T, dID> entry on the second pass. Because we are only resubmitting DNS response packets, any overhead due to resubmission is negligible. For example, in our three-hour trace used for testing in Section 5, DNS response packets made up 0.14% of total packets. In our 15-minute trace, DNS response packets only made up 0.89% of total packets. With these packets contributing very little to the total corpus of packets traversing the switch, resubmitting these packets adds little additional overhead. Besides, we only resubmit a DNS response packet when it has to replace a stale entry after a hash collision, which further reduces the overhead.

Even with resubmission, however, our entire running example does not fit into the limited stages of the ingress pipeline. This further motivates the design choice of keeping Meta4's core modules in the ingress pipeline while the application-specific part goes into the egress pipeline. This design choice also enables easier integration of other DNS-based monitoring use cases with Meta4. For the Meta4 traffic volume measurement program, for example, we move the data structures counting packets and bytes indexed by domain ID to the egress pipeline. In general, we keep the DNS responses table in the ingress for all Meta4 programs. Any subsequent data structures unique to a particular use case are realized in the egress pipeline, such as the DNS statistics table. Meta4 defines a custom packet header, which is used for delivering the each packet's domain ID information from the ingress pipeline to the egress pipeline.

6.3 Measuring Traffic Volume by Domain

We revisit the use case from our running example. We ran our traffic volume measurement program on a Tofino switch with the same 15-minute and three-hour traces from the campus network that we used in Section 5.2. The 15-minute trace was captured at 15:00-15:15 on April 7, 2020 and the three-hour trace at 08:00-11:00 on August 19, 2020. The 15-minute trace was captured while school was in session during mid-afternoon, whereas the three-hour trace was captured while school was out of session during the morning. The monitored traffic is not necessarily representative of campus traffic in normal operations. We believe the ongoing global pandemic had some impact on traffic patterns, like popular domains, seen on campus. To create our policy configuration, we used historic heavy-hitter analysis to determine the most popular domains requested on campus. For each trace, we used Netify's application lookup tool [18] to map particular domain names (*e.g.*, ytimg.com) to a known service or application (*e.g.*, YouTube).

In the 15-minute trace, the top service both by packet and byte counts was Steam, which is an online game hosting platform [24]. Interestingly, the DNS traffic for Steam only accounted for 0.005% of total DNS traffic, but the packet and byte counts were 17.5% and 16.7% of total traffic, respectively. Facebook, Google Ads, and Limelight CDN traffic were the next biggest, followed by Skype/Microsoft Teams, Google (general), Reddit, iCloud, and Instagram. It was noteworthy that Google-related services have produced the most DNS traffic, accounting for 36.5% of total DNS traffic.

In the three-hour trace, the top service by traffic volume was Skype/Microsoft Teams, which accounted for one-third of all packets and bytes. Google Ads, Google (general), Apple (general), Zoom, and YouTube traffic were the next heavy-hitters. Microsoft Teams is used heavily by the campus IT department staff. In addition, it is important to note that oftentimes, traffic associated with certain domains is not specifically requested by actual users. For example, the prevalence of Google Ads is not due to users specifically requesting the Google ad services domain, but rather due to the high frequency of Google Ads (including still image banners and videos) across the web, including non Google-affiliated websites. Similar to the 15-minute trace, Google-related services have produced the most DNS traffic, accounting for 26% of total DNS traffic. Also, Zoom, while contributing a significant portion of traffic, is not as dominant as one might have expected for campus traffic. The lack of Zoom traffic can largely be attributed to classes not being in session during the period the three-hour trace was captured and an increasing shift to pre-recorded lectures over Zoom-hosted lectures during the period the 15-minute trace was captured.

7 ADDITIONAL USE CASES

We now turn to other use cases of Meta4 to show how various applications can be built off of the same fundamental framework described in Sections 2 and 4. We also evaluate their performance based on our experiments with real hardware.

7.1 DNS Tunneling Detection

DNS tunneling is a method of using the DNS protocol to bypass security protocols/firewalls to send or receive normally restricted traffic. While DNS is not usually intended for data transfer, the fact that it is allowed to bypass most firewalls makes it a potential tool for malicious users seeking to "tunnel" unwanted traffic [6].

To detect potential incidents of DNS tunneling, we need to identify client IP addresses that are making an abnormal number of DNS requests with no subsequent traffic to those server IP addresses contained in the DNS response. This particular Meta4 use case is unique in that it does not require a list of domain names and a corresponding match-action table as we are interested in all DNS response packets and not just those for specific domains. Yet, the domain name in the DNS response has to be successfully parsed so that Meta4 can extract the server IP address that follows it.

For DNS response packets, the program uses a hash of the server and client IP addresses to find a table entry in the DNS response table. Unlike the traffic volume case for Meta4, however, there is no need to store the domain ID. Then, using the hash of the client IP as index, an entry in a counter register called the client IP table is accessed, either creating a new entry when empty or increasing the counter when an entry already exists.

For a non-DNS data packet, the source and destination IPs are hashed to key into the DNS response table. If there is a match, the timestamp is set to 0 to indicate that the table entry is free and available for re-use. The counter in the client IP table is then decremented to indicate that the DNS response was followed by actual traffic. If a DNS response is never followed by additional data packets, the entry in the DNS response table will eventually time out, after five minutes, and the counter in the client IP table will never be decremented. Obviously, an entry in the client IP table with a high count is a suspect for DNS tunneling.

To test our Meta4-based DNS tunneling application, we generated two instances of DNS tunneling traffic using the open-source

dnscat2 tool [7] and embedded them within normal, benign traffic. First, the client made an SSH connection through a DNS tunnel and performed simple UNIX commands (*e.g.*, `cd`, `ls`). For the second case, the client made an SCP connection through a DNS tunnel to make a data transfer around 5 MB. We made a control-plane script that queried the registers every 30 seconds to see if there were any positive detection. A positive detection was defined as a client that has a counter value of five or more in the client IP table, which means there were five more DNS response packets than data packets. In our test, we had zero false positives and were able to successfully detect both SSH and SCP traffic that went through DNS tunneling.

DNS packets that are being used for tunneling are often significantly longer than normal DNS packets as they hold relatively large amounts of data where the domain name would usually be stored in a DNS packet. Because of this, it is often difficult for the P4 parser to completely parse through a DNS tunneling packet. Nonetheless, our program was able to parse through and detect about half of the DNS packets used for tunneling, allowing us to catch instances of DNS tunneling in less than a minute once started. In the case of the SSH tunneling traffic, we were able to detect 132 out of 306 DNS tunneling packets. In the case of the SCP tunneling traffic, we were able to detect 35,089 out of 75,078 DNS tunneling packets.

7.2 IoT Device Fingerprinting

It is possible to fingerprint particular IoT devices by tracking the domains they talk to. For example, Saidi *et al.* proposed a reliable method for fingerprinting particular IoT devices by tracking the domains names they reach out to and the server's source port number when sending traffic to clients; with 33 devices from different manufacturers, 97% were able to be detected within 72 hours of active monitoring [22].

We implement the same mechanism in the data plane using Meta4. Using the public IoT signatures [10] from this previous work, we generate a known list of domains that populates the match-action table rule in Meta4. An IoT device type can have multiple signatures or rules; matching more rules increases the detection accuracy. Our Meta4-based IoT detection program processes DNS response packets identical to the traffic-volume measurement use case. The domain name from the DNS response packet is parsed and a match-action table is used to find the domain's associated ID. An entry is accessed in the DNS response table using a hash of the server and client IP addresses. The timestamp is also set using the DNS packet's timestamp. The domain ID is set from the ID matched in the match-action table.

When a data packet from the server to the client has a source and destination IP addresses that match an entry in the DNS response table, we set the timestamp in the entry to `0` to indicate that the entry is free for re-use. We then generate a special *report packet* that includes the domain ID, client IP address, and the server's source port number. By accumulating a long-term record of such report packets, an operator can detect particular IoT devices. Such reports can be either saved in a separate register in the data plane or sent to an external controller, where former is better for privacy.

To evaluate our IoT fingerprinting use case, we ran our program with 10 hours of traffic from a IoT traffic dataset downloaded from IMPACT (Information Marketplace for Policy and Analysis of Cyber-risk and Trust) [13]. This packet trace contains traffic from seven IoT devices that are present in the public IoT signature list [10]: Alexa Fire TV, Alexa Echo Dot, Philips Hub, TPlink smart bulb, TPlink smart plug, Amcrest camera, and Wansview camera. Our Meta4-based IoT detection program successfully detected all seven devices. For each of the seven devices, we were able to get at least one positive detection without any false device detection, confirming the effectiveness of our fingerprinting application. The Alexa-based devices were the easiest to detect as they have the longest list of known signatures. The Philips Hub was the next, followed by the remaining devices.

8 DISCUSSION AND FUTURE WORK

Ambiguity with the correction strategy: As discussed in Section 5.4, our correction strategy described in Section 4.3 generally works well, but is not perfect. Domains with highly variable traffic amount during a client-server connection can make the scaling factor unreliable. Also, a domain name insertion "miss" (due to full a register array) followed by a "hit" for the same domain can result in largely incorrect counting, especially for long-lived flows. For example, consider a situation where a client downloads two large files from a domain named `foo.com`, and assume each download takes 60 minutes. Let's say the client requests one file first and then requests the other one after 10 minutes. Now assume the first DNS response was missed but the second DNS response was a hit and got inserted into the <cIP, sIP, T, dID> register array. Meta4 will start to count data packets once the second DNS response got inserted, actually counting data packets for the first file download flow too, only missing the first 10 minutes. In result, the packet and byte counts for domain `foo.com` will be for 110 minutes. However, our correction scheme will only consider the fact that the first insertion has failed and "correct" this by doubling the count. Now, the byte and packet counts for the domain will be for 220 minutes while for 120 minutes is actually the correct one.

Although rare, the situation like above can certainly happen. One way to tackle this is to be more consistent when deciding a hit or miss for a particular domain. With this in mind, a promising strategy is to perform *sampling* by tuple (<cIP, sIP, dID>) with some probability p. In other words, either apply a fixed sampling rate on a tuple or never account for it. Then, rather than correcting the counts by multiplying with a scaling factor calculated by hits and misses, multiply by $1/p$. The advantage of this approach is that the scaling factor is a consistent, unbiased value based on probability p. We leave the exploration and implementation of different correction strategies as future work.

Multiple domains using a shared server IP: The prevalence of Content Distribution Networks (CDNs) in today's Internet ecosystem can lead to multiple different domain names sharing a single server IP address. To deal with this ambiguity, Meta4 uses the client and server IP pair (<cip, sip>) to map packets to a domain instead of using just the server IP. By doing so, the ambiguity arises only when a single client concurrently (or in close succession) requests and visits two or more domains that share a server IP address. One common scenario for a client to request multiple different domains in close succession is when the client visits a website that has

closely related but different components that need to be loaded. For example, visiting "`www.bbc.com`" subsequently makes a client to send DNS requests for "`bbc.map.fastly`" and "`fig.bbc.co.uk`", and all three domains map to the same server IP address. In such a scenario, however, the associated traffic would correspond to the same "class" anyway, thus would have the same domain ID. Besides, shared hosting among totally different domain names is expected to be less common for popular domains.

In any case, it is important to note that multiple domains using a shared server IP is a known issue for domain-based monitoring in general, not just for data-plane approaches like Meta4. An interesting future work is to investigate how to clear this ambiguity. One approach is to analyze the traffic pattern to further fingerprint a website only when such ambiguities arise. Meta4 can possibly interact with such data-plane or control-plane solution in real time.

9 CONCLUSION

This paper outlined the design and implementation of Meta4, a network monitoring framework in the data plane that allows an operator to associate traffic by domain name. Developed for PISA-based programmable switches, Meta4 parses DNS response packets to dynamically extract client IP, server IP, timestamp, and domain name at line rate, which are then stored on registers in the switch. Using a register data structure, Meta4 associates subsequent data packets to a domain by matching the destination and source IP addresses to the client and server IP address in a register entry.

We implemented Meta4 on Tofino hardware using the P4 language and deployed it on a campus network. We evaluated Meta4 and showed that even under the hardware restrictions and limitations of the switch, we were able to achieve a relatively high accuracy in terms of traffic volume measurement by domain name. We also presented two more use cases: DNS tunneling detection and IoT device fingerprinting.

Meta4 highlights an interesting approach for realizing intent-based networking with programmable data planes. In particular, Meta4 presents a unique approach of "joining" two different packet types within the data plane, using a common IP header field. We believe this form of intentional network monitoring can be expanded beyond DNS, and we hope this work shed some light on discovering unique opportunities with programmable data planes.

ACKNOWLEDGMENTS

We want to thank Xiaoqi Chen, Mary Hogan, Mengying Pan, Ross Teixeira, Yufei Zheng, and the anonymous reviewers for their helpful feedback. We also thank our shepherd, Kai Gao, for his great guidance. We want to thank Princeton University's Office of Information Technology, Office of Institutional Research, and the Institutional Review Board for reviewing our work and helping us evaluate our work with anonymized campus traffic. This material is based upon work supported by NSF Grant number CNS-1704077 and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0107.

REFERENCES

[1] MozillaZine knowledge base: DNS cache expiration. http://kb.mozillazine.org/Network.dnsCacheExpiration, 2010.
[2] Chromium issue 164026: DNS TTL not honored. https://bugs.chromium.org/p/chromium/issues/detail?id=164026, 2018.
[3] The Spamhaus Project. https://www.spamhaus.org, 2021.
[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 186–201, 1999.
[5] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols (ICNP)*, pages 313–323, 2018.
[6] K. Borders, J. Springer, and M. Burnside. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium*, pages 365–379, 2012.
[7] R. Bowes. DNSCAT2. https://github.com/iagox86/dnscat2, 2013-2015.
[8] Chromium open source code: address_list.h. https://source.chromium.org/chromium/chromium/src/+/master:net/base/address_list.h.
[9] S. Donovan and N. Feamster. Intentional network monitoring: Finding the needle without capturing the haystack. In *ACM SIGCOMM HotNets Workshop*, 2014.
[10] Github repository for IoT signatures: A haystack full of needles: Scalable detection of IoT devices in the wild. https://github.com/IoTrim/iot_wild, 2021.
[11] P. Hoffman and P. McManus. DNS queries over HTTPS (DoH). RFC 8484, RFC Editor, October 2018.
[12] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, RFC Editor, May 2016.
[13] Information Marketplace for Policy and Analysis of Cyber-risk and Trust (IMPACT). IoT bootup and operation traces. https://www.impactcybertrust.org/dataset_view?idDataset=1144, https://www.impactcybertrust.org/dataset_view?idDataset=1491, 2020.
[14] Intel Tofino Chip. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2021.
[15] 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 *ACM SIGCOMM*, pages 90–106, 2020.
[16] X. Li. P4 authoritative DNS server. *Barefoot Networks*, 2017.
[17] Meta4 public code repository on Github. https://github.com/Princeton-Cabernet/p4-projects/tree/master/Meta4-tofino, 2021.
[18] Netify application lookup tool. https://www.netify.ai/resources/applications, 2021.
[19] Intel Open Tofino Github repository. https://github.com/barefootnetworks/Open-Tofino, 2021.
[20] P4 behavioral model (BMv2). https://github.com/p4lang/behavioral-model, 2021.
[21] PerfMatrix. Think time: Importance of think time in performance testing. https://www.perfmatrix.com/think-time/, 2019.
[22] S. J. Saidi, A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann. A haystack full of needles: Scalable detection of IoT devices in the wild. In *ACM Internet Measurement Conference*, pages 87–100, 2020.
[23] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *ACM Symposium on SDN Research (SOSR)*, pages 164–176, 2017.
[24] Steam: Game hosting platform by Valve. https://store.steampowered.com, 2021.
[25] J. Woodruff, M. Ramanujam, and N. Zilberman. P4DNS: In-network DNS. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2019.