Meta4: Analyzing Internet Traffic by Domain Name in the Data Plane

Jason Kim
Advisor: Jennifer Rexford
Princeton University

Abstract

Associating network traffic with human-readable domain names, instead of low-level identifiers like IP addresses, is helpful for network operators who might need to rate-limit traffic by domain, conduct web-fingerprinting, identify IoT devices, measure traffic volume by domain name, and more. The problem is that most current methods of network monitoring require collecting and examining large amounts of network traffic in a way that may compromise user privacy. The emergence of high-speed programmable switches makes it possible to implement monitoring programs that run in the data plane at line-rate without revealing user information to network operators. In this paper, we introduce **Meta4**, a framework for implementing network monitoring by domain name in the data plane by extracting the client IP, server IP, and domain name from DNS response packets and using this information to identify the domain name associated with packets from the subsequent client-server session. A data-plane implementation has the benefits of preserving the privacy of sensitive user information, running efficiently at line-rate, and allowing network operators to take action on network traffic to rate-limit, block, or mark packets based on their associated domain. We implemented Meta4 on a Barefoot Tofino P4-programmable switch and deployed and assessed our implementation on the Princeton University campus.

1 Introduction

Monitoring network traffic usually is conducted 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 capture traffic based on policies like "capture all traffic related to video streaming services" rather than having to worry about low-level details of identifying which flows are associated with various streaming services.

Efforts at implementing this higher-level network monitoring or "intentional network monitoring" have been made, in particular, by Donovan and Feamster's NetAssay system [6] which implements intentional network monitoring using metadata engines at controllers receiving traffic redirected from switches across a network. In this paper, we seek to lay the groundwork for implementing domain-based intentional monitoring at the switch-level, within the data plane itself.

The advent of programmable data planes allows us to quickly parse and process packets at the switch without having to capture and store large amounts of extraneous network traffic. By parsing DNS response packets, we can extract the client's IP address (the destination IP address of the packet), 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 packet). By joining the client IP and server IP pair with the domain name in the data plane (hence referred to as the <cIP, sIP, D> tuple, referring to the client IP, server IP, and domain name, respectively), we are able to analyze client-server sessions by the domain that they are associated with. Here, we define "session" as all the traffic a given 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 connections over time due to DNS response caching by the client.

It is important to note that dynamically joining IP address pairs to domain names using DNS response packets is necessary as opposed to keeping a static mapping of server IP addresses to domain names. This is because multiple domain names may have the same IP (various domains may share the same server) or a single domain name can have multiple IPs (CDNs may load balance/geographically distribute servers across different clients).

Taking advantage of a PISA (Protocol Independent Switch Architecture)-based programmable data plane, we introduce **Meta4**, a framework for implementing domain-based intentional network monitoring. Meta4 has several benefits over control-plane applications:

- **Privacy:** 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. Furthermore, in the case of fingerprinting or detection-based applications, a network operator never needs to store or see traffic from users whose behavior does not result in positive detections, thus limiting breaches in user privacy to those clients that a network operator is specifically looking to flag or detect.

- **Efficiency:** Data-plane programs are very fast and can outperform what can be done in the control plane, allowing us to monitor large amounts of traffic without compromising network performance.

- **Direct Action:** In addition, implementing these applications within the data plane allows network operators to take direct action on packets in a network based on results from the application. In other words, network operators could use the results reported from a domain name heavy-hitters application to dynamically rate-limit or re-route traffic from particular domains. Furthermore, a network operator could dynamically block traffic from a client IP associated with suspicious DNS tunneling, malware injection, or other malicious behavior on a network.

Domain-based monitoring within the data plane allows for network operators to create a variety of applications such as:

- **Traffic Volume Measurement by Domain:** Network operators can define domain-based "traffic classes" such as "Netflix video traffic" or "Zoom video chat traffic" and monitor the amount of traffic volume (in bytes or packets) sent by servers to clients for domains associated with these traffic classes. (section 6.1)

- **DNS Tunneling Detection:** By identifying clients who routinely make DNS requests with no follow-up traffic, we can easily identify client IP addresses that are suspects for using DNS for nefarious purposes. (section 6.2)

- **IoT Device Detection and Fingerprinting:** By identifying domain names commonly associated with particular IoT devices, we can identify devices on a network that routinely send and receive traffic to domains that are unique identifiers of a type/model of IoT device. (section 6.3)

- **Webpage Fingerprinting:** By identifying the domain names requested by a client and quantity/size of packets received when downloading a specific web page, a form of web fingerprinting can be conducted within the data plane to identify when clients visit particular web pages of interest.

- **IDS Bypass by Domain:** In order to improve network performance, a network operator could allow traffic from certain trusted domains to bypass processing by an IDS (intrusion detection system).

- **Firewall Rules based on Domains:** A network operator could also create firewall rules based on domain name that restrict the flow, redirect, or drop packets associated with certain untrusted domains.

However, implementing Meta4 within the data plane is not a straightforward task. PISA-based switch hardware imposes a number of restrictions that make implementing Meta4 difficult. For one, PISA switch parsers are designed to parse fixed-length header fields which make dealing with variable-length fields like 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 the processing pipeline).

In section 2, we discuss the architecture of a basic Meta4 use case that measures traffic volume by domain as well as define in detail the challenges with creating a Meta4 implementation within the data plane. In section 3, we discuss the design and implementation of that Meta4 use case and detail how we overcame the challenges discussed in section 2. In section 4, we evaluate the efficacy and performance of the Meta4 application measuring traffic volume by domain name on a campus deployment and in section 5, we discuss the particular challenges of implementing and deploying Meta4 on hardware. In section 6, we describe how to implement several other Meta4 use cases and evaluate their utility. In section 7, we discuss related work that helped to inform the methodology and methods used in Meta4. In section 8, we conclude by discussing potential avenues for future work related to Meta4.

2 Meta4 Architecture and 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. By doing this, the program can then identify how much traffic, measured in terms of packets or bytes, is associated with a particular domain. 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 1. The system is defined by three levels: the network operator, the control plane, and the data plane. The network operator defines high-level policies that the control plane converts into match-action rules for the data plane.

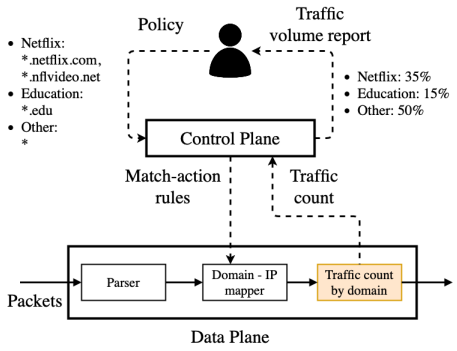

Figure 1: Meta4 is designed to take high-level policies defined by network operators and convert them to rules that can be used by the data plane to process packets. In this use case, policies define types of traffic that Meta4 measures volume for. The control plane converts these policies to match-action rules.

In this specific application, a network operator wishes to measure traffic volume by certain traffic classes (In Figure 1, 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 sources of traffic that the network operator is interested in. 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 what domains subsequent data packets are associated with. 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 (traffic volume of Netflix vs. education, for example).

2.2 PISA Implementation Challenges

The goal of implementing Meta4 within the data plane is possible thanks to PISA (Protocol Independent Switch Architecture) switches which contain a programmable packet parser, processing pipeline, and a packet deparser. The processing pipeline contains both an ingress and egress pipeline. Each can contain match-action tables for matching on packet headers or metadata, and register arrays for storing information. The processing pipeline can also modify packet header fields and metadata and make packet forwarding or dropping decisions. The processing pipeline is composed of a limited number of stages which limits the amount of operations that can be performed for each packet as it moves through the pipeline.

Parsing: Prior to the processing pipeline, packets go through a programmable packet parser capable of extracting header fields and metadata from a packet. The processing pipeline can then use information extracted from a packet to make decisions on storing or updating information in memory and take action on the packet. Primarily intended for parsing fixed-width headers, the parser is only able to parse through fixed number of bytes. While the parser can make different parsing decisions by switching to different parser states based on the values of header fields, it cannot parse through a variable number of bits. This constrains a programmer by forcing them to pre-define all potential parser actions that a program may require. In the context of Meta4, this poses a challenge with parsing domain name fields from DNS response packets since domain names are, by their nature, variable-length strings.

Match-Action Tables: Within the processing pipeline, a program can make use of match-action tables which can match on packet header fields and metadata. Upon a match (or miss), the program can choose to take some action such as a forwarding or dropping decision, an update to header or metadata values, or an update to a register. Each stage in the pipeline can store match-action rules in a limited amount of TCAM or SRAM. In the context of Meta4, match-action rules provide a natural way for us to match domain names from a DNS packet to check if a domain name belongs to one of our traffic classes. However, the limited height of TCAM memory, in particular, limits the total length of a domain name that we can match on within a stage.

Register Memory: PISA switches have the ability to preserve some state in persistent memory across multiple packets by way of registers. Within the processing pipeline, each stage contains a limited number of registers of limited and fixed width [9]. Furthermore, each stage can only perform a few concurrent memory accesses to its registers. In practice, this means that a program can only access (read/write) to a register once per packet. This naturally causes difficulties with many programs for PISA switches that need to perform more complex operations that depend on reading existing values from a register, taking some action, and then updating that same register value. Limitations in the total amount of memory available in the form of registers mean that it is likely impossible to store all <cIP, sIP, D> tuples from every single DNS response packet in the switch. This means that we will have to create a solution to intelligently evict stale entries to allow room for new DNS response entries.

Packet Resubmission: Normally, each packet goes through the processing pipeline once, and cannot repeat and move backwards to prior stages in the pipeline. This severely limits the number of operations that can be programmed for a

packet as well as the ability to access data structures multiple times for a single packet. One tool available within PISA switches that provides some flexibility for programmers is the ability to resubmit a packet. Resubmission, which can only be done once per packet, causes a packet to be be resubmitted to the same ingress port once it finishes its initial pass through the ingress pipeline. Resubmission, at the cost of some latency caused by the bandwidth consumed by having packets go through the pipeline multiple times, allows a programmer to use more stages to increase the complexity of a program and perform more processing on a packet. Resubmission will be a key tool in solving one of the primary restrictions of register memory. As mentioned before, limited memory forces us to evict stale entries to allow room for new ones. Resubmission, which allows us to access the same register entry twice, gives us the ability to check for a stale entry, and if the entry is stale, to replace that old entry.

3 Name-Based Monitoring in the Data Plane

In this section, we describe our design and implementation for a canonical Meta4 application that measures traffic volume in terms of bytes and packets sent by servers to clients by domain name via a PISA-based programmable data plane. We show how we solved hardware limitations discussed in section 2 of parsing (section 3.1), memory limitations (section 3.2), and information loss (section 3.3).

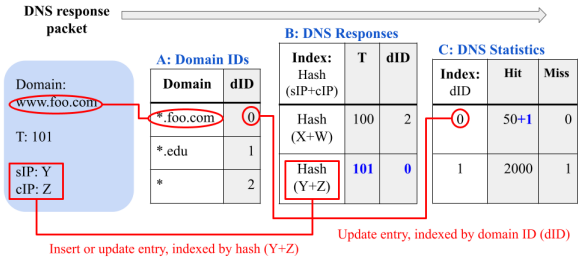

Figure 2: Meta4 response to a DNS response packet. D refers to a domain name, dID refers to a domain ID number, T refers to a timestamp, sIP and cIP refer to server and client IP addresses respectively.

3.1 Parsing Long, Variable-Length Names

As discussed in section 2.2, extracting variable-length domain names from DNS response packets is a challenging task within the data plane since parsers are designed with the intent of parsing fixed-length packet headers, not variable length strings of characters like domain names.

In order to successfully parse domain names, it is important to understand how DNS encodes domain names. Domain

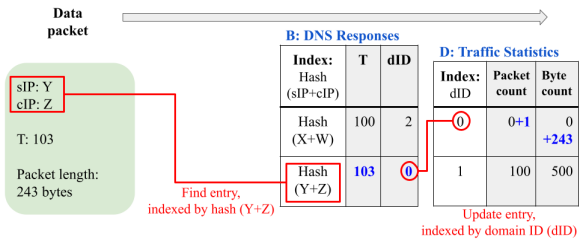

Figure 3: Meta4 response to a non-DNS data packet.

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 of seven bytes. 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 if we had the ability to parse a domain name with any number of labels and if we allowed the max width of a label to be parsed (255 is the max width of a label since it is the max value of an octet). However, in reality, hardware limitations, particularly with limitations in TCAM when trying to apply match-action table rules to a parsed domain name, drastically confine the size of domain names that can be used in our programs. In our P4_16 implementation on the Tofino switch, we limited parsing domain names to a maximum of four labels with 15 characters each.

Note that while a different domain parsing configuration such as four labels with 20, 20, 5, 5 characters respectively could work better on some domain names such as "www.nationalgeographic.com" which tend to have labels with many characters followed by short labels like "com"

or "org", it is better to maximize the number of labels that we can parse and keep the number of characters that can be parsed equivalent for each of those labels. Since domain names are parsed sequentially and we cannot tell beforehand how many labels are in a domain name, we cannot preconfigure the parser to work well for both a domain like "www.nationalgeographic.com" and a domain like "getpocket.cdn.mozilla.net". We evaluate the impact of this limitation in section 4.1.

3.2 Efficient Memory Usage

3.2.1 Avoiding Storing Domain Names

As discussed in section 2.2, register memory has a fixed width which means that once we have parsed domain names, storing them as variable-length strings can be quite difficult. The solution used was to take advantage of the list of domain names provided by the control plane and assign each traffic class an integer index ID (0, 1, 2, 3, and so forth) based off of what traffic class (as defined in section 2.1) the domain is associated with.

Using a match-action table as seen in the Domain IDs Table (labeled A in Figure 2), we can match domains as they were parsed to domains in the known list. If a match occurs, a domain can be represented as its integer domain ID corresponding to its traffic class, allowing us to store just the number and not the entire domain name. As depicted in Figure 2, all references to a domain name in subsequent data structures use this domain ID as opposed to the full domain name. As seen in Figure 3, the domain ID also is the array index for the Traffic Statistics table which stores accumulated packet/byte-count information for each traffic class.

3.2.2 Multi-Stage Register Data Structure

As depicted in Figure 2, the <cIP, sIP, D> tuple, extracted from a DNS response packet, is stored within a register data structure known as the DNS Response table (labeled B in Figure 2 and Figure 3). The purpose of this data structure is to match packets with specific client-server/domain sessions.

Ideally, this data structure would be able to accommodate an unlimited number of entries. However, as mentioned in section 2.2, due to the register memory constraints found in PISA-based architectures, a large amount of DNS response traffic could quickly cause a loss of information as new entries encounter hash collisions with existing entries.

To avoid hash collisions, which inevitably occur as the table fills up, we implement a solution inspired by the multi-stage data structure used in the PRECISION algorithm [2]. The basic premise is to divide the data structure into multiple sets of registers. If a hash collision occurs at one register set, the program then tries the next register set with a hash with a different salt and so forth until it finds an available entry (or until it tries every single register and fails). While this method uses the same amount of memory as just a single long register set, dividing the register into multiple sets can promote a more efficient use of register memory.

3.2.3 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 free 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 2. Subsequent packets that belong to the same client-server session are used to update the timestamp as seen in Figure 3. If a new table entry encounters a hash collision with an existing entry, but the old entry's timestamp is older than some timeout value T (T seconds have passed since the last packet sent to the client from the server), then the new entry is allowed to occupy that space in the table (see section 4.2 for explanation of how we experimentally determined a good timeout value). 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. As suggested in section 2.2, the operation of checking if a table entry is timed out is problematic for a PISA register because it depends on both reading an existing timestamp value and then potentially replacing it with a new timestamp value (if we choose to replace the existing register entry). Both of these operations cannot be conducted within the same stage which requires a packet resubmission operation in order for us to properly implement this data structure by allowing us to go through the ingress processing pipeline a second time. Because DNS response traffic typically consists of less than 1% of total traffic, the overhead incurred by resubmitting DNS response packets has minimal effect on network performance.

3.3 Correcting for Missed DNS Responses

There are two potential sources of error or information loss that can occur in Meta4. First, we have error due to a failure to store a certain <cIP, sIP, D> tuple due to hash collisions in our DNS Responses table. In the event that the existing entries in the DNS Responses table are not timed out, a given <cIP, sIP, D> tuple may never be stored in the first place, causing us to miss all traffic associated with this client-server/domain session.

The second source of error is due to a premature eviction of a <cIP, sIP, D> tuple from the DNS Responses table. A given entry in the DNS Responses table may be timed out (no traffic for this entry has been seen for more than say 100 seconds), causing a new <cIP, sIP, D> tuple to evict the old entry. In

this case, if there was in fact more traffic for the older session, we would miss this traffic due to the premature eviction.

These two sources of error tend to work against each other. If we increase our timeout value to reduce the second type of error, we may cause our DNS Responses table to bloat, causing new <cIP, sIP, D> entries to experience more hash collisions and fail to be stored. On the other hand, if we reduce our timeout value to allow more frequent eviction of old entries and allow more new <cIP, sIP, D> entries to be stored, we may prematurely evict old entries more often. However, while reducing the second type of error is just a matter of fine-tuning the timeout value, there are other ways to reduce the first type of error. While not a perfect solution, we introduced a mitigating measure to avoid potential under-representation of packet/byte counts associated with certain domains due to missed <cIP, sIP, D> entries.

The goal is to calculate an approximation of the proportion of total packets/bytes missed for a certain domain/traffic class. To do this, we turn to a relatively easy metric: the proportion of <cIP, sIP, D> tuples that we are unable to store in the DNS Responses Table. To do this, we introduce an additional data structure to keep count of the number of DNS response packets seen for each domain and to keep count of the number of DNS responses seen for each domain that are not able to fit within the IP-Domain Match table. The DNS Statistics Table (labeled C in Figure 2) allows us to calculate the proportion of DNS packets, and hence client-server sessions, for a particular domain that we were not able to monitor. Using this proportion, 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, as seen Figure 2 if Meta4 records three DNS response packets for "example.com" but is not able to store the client IP, server IP, domain ID tuple for one of those packets, we then scale the byte/packet counts for "example.com" by 1.5 $(1/(2/3))$. Thus, if Meta4 records seeing nine packets and a total of 1320 bytes for "example.com", it reports an estimate of $9 \times 1.5 = 13.5$ packets and $1320 \times 1.5 = 1980$ bytes.

4 Performance Evaluation

In this section, we evaluate how well Meta4 performs using our traffic volume measurement application example from section 3. In particular, we assess how well Meta4 performs within data-plane constraints such as parsing limitations and register memory constraints as discussed in section 2.2. In order to provide a comparison for Meta4's performance under varying quantities of memory/parsing resources, most of the experiments in this section were performed in a Python program that emulates Meta4's behavior in the data plane. Unless otherwise stated, all experiments were run on a three-hour trace from Princeton University's campus. The traffic trace was anonymized and sanitized to obfuscate personal data before being used by researchers, and our research was approved by the university's institutional review board. A repository with code for these experiments can be found below[1].

4.1 DNS Response Parsing Limits

The first experiment assumes unlimited register memory and applies various parsing constraints to the domain name parser. The source of this limitation is discussed in section 3.1. 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 (non-DNS packets that are part of the associated data transfer between a client and a server discovered through DNS) and the number of bytes from those packets that are missed as a result of being unable to successfully parse a 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. For example, in Figure 4, 60 bytes on the x-axis corresponds to a domain parsing configuration of four labels with 15 bytes allocated to each label.

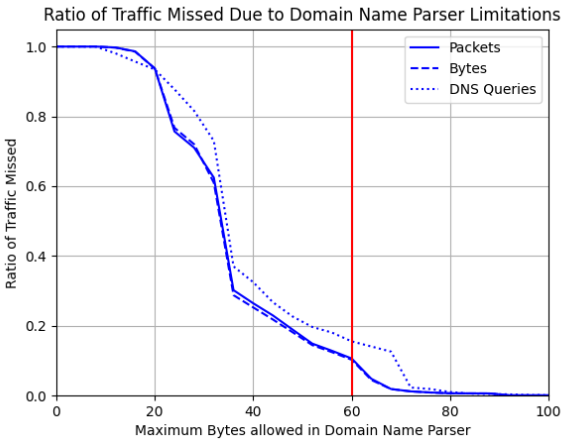

Figure 4: The x-axis shows the maximum width of a domain name allowed in bytes in the parser and the y-axis shows the amount of traffic missed with 1.0 meaning that all traffic is missed and 0.0 meaning that no traffic is missed. The red line marks the parser limit of our Tofino implementation.

Figure 4 shows the results of the experiment. Note that under the domain parsing constraint used in our Tofino implementation, we are unable to parse about 18% of DNS queries/responses but only miss about 10% of actual traffic associated with those requested domains. This indicates that the domains we are not able to parse contribute proportionally less traffic than the domains that we are able to parse.

[1] https://github.com/jkim117/Meta4

A closer inspection reveals that on average, each individual domain that we were not able to parse contributes only 0.0037% of DNS responses, 0.0049% of packets, and 0.0048% of bytes. The domain that was the single greatest contributor of byte/packet traffic out of the domains we were not able to parse in our test campus trace (m-9801s3.ll.dash.row.aiv-cdn.net) contributed 4.0% of byte traffic and 3.4% of total packets. The domain that was the single greatest contributor of DNS responses out of the domains we were not able to parse in our test campus trace (124.230.49.37.4vw5piqz5ksoolyszwje5tobsi.sbl-xbl.dq.spamhaus.net) contributed 5.2% of DNS responses but 0 packets or bytes. This is because this DNS response is actually a spam blacklist [14] using the DNS protocol for convenience. Many of the DNS response packets that we are not able to parse are not actually normal DNS responses made by human clients requesting domains which is why they proportionally contribute less traffic than the domains we are able to parse. Overall, the vast majority of unparseable domain names contribute little or no traffic, so missing this traffic has a relatively small effect on the overall effectiveness of Meta4 as a tool to measure major sources of traffic volume.

4.2 Evaluating DNS Table Timeout

The next experiment was designed to determine a good timeout value as described in section 3.2.3 with the goal of missing a minimal amount of traffic due to evicting register entries before a server finishes sending packets to a client. At the same time, we want to keep the timeout relatively low in order to encourage new <cIP, sIP, D> entries to evict stale ones. Thus, in this experiment, we sought to determine the amount of traffic that would be missed if we evicted <cIP, sIP, D> entries if the inter-arrival time for packets corresponding to a particular <cIP, sIP, D> exceeded some timeout value.

As seen in Figure 5, most of the benefits of increasing the timeout value end after a timeout of 100 seconds. This is the case for multiple reasons. For one, many browsers like Chrome [5] and Firefox [12] cache DNS results for a default of 1 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 [13], allowing us to effectively capture all traffic from most client-server sessions.

Note that the result from Figure 5 does not reflect what actually happens in Meta4. In Meta4, we do not destroy table entries that are timed out until a new, incoming <cIP, sIP, D> tuple experiences a hash collision with an existing timed out entry. This "lazy eviction" allows for the possibility of timed out entries to still be used and become "un-timed out". This allows us to capture a significantly larger portion of traffic.

In addition, Meta4, unlike in this experiment, has limited

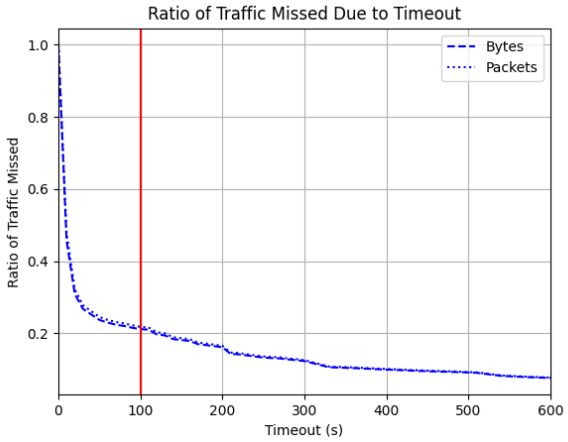

Figure 5: The *x*-axis shows the timeout applied on <cIP, sIP, D> tuples and the *y*-axis shows the amount of traffic missed with 1.0 meaning that all traffic is missed and 0.0 meaning that no traffic is missed. The red line marks the timeout used by our Tofino implementation.

memory resources. That means that a smaller timeout may actually be advantageous as it allows for stale <cIP, sIP, D> entries to be evicted more frequently, allowing new entries to take their place so that Meta4 can monitor new traffic for newer <cIP, sIP, D> tuples.

In order to determine the proper timeout value under limited memory resources, we ran a similar experiment but with memory constraints[2] and did not automatically evict timed-out entries (until a hash collision forced an eviction of a timed-out entry). We ran this experiment on two separate traces. The first trace was the same 3-hour trace that we have been using for all previous experiments. The second trace was a 15-minute trace, also anonymized and sanitized from Princeton's campus, but captured during a period of greater traffic density. In Table 1, we show a comparison of the the two packet traces. Notice in particular that traffic density of DNS responses is 10 times greater in the 15-minute trace. This means that the frequency of DNS response packets, each creating a new <cIP, sIP, D> entry, is tenfold in the 15 minute trace.

Time of Day	**Packets/s**	**DNS responses/s**
15:00-15:15 APR 7, 2020	240750.23	2151.41
08:00-11:00 AUG 19, 2020	138382.92	205.26

Table 1: Comparison of the traffic density for the 15-minute (top) and 3-hour (bottom) packet captures

The results shown in Figure 6 for the 3-hour trace seem to confirm the similar results from when there were no memory

[2]We limited the DNS Responses table to 2^{16} total entries and 2 stages. This is the same configuration that we used in our hardware implementation as explained in section 4.3

constraints (see Figure 5) that a longer timeout is better for capturing more traffic. However, the results for the higher density 15-minute trace show that increasing the timeout does not lead to a consistent improvement in performance of Meta4. Because the 15-minute packet trace has a significantly higher traffic rate, the DNS Responses 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. Because of these results, 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.

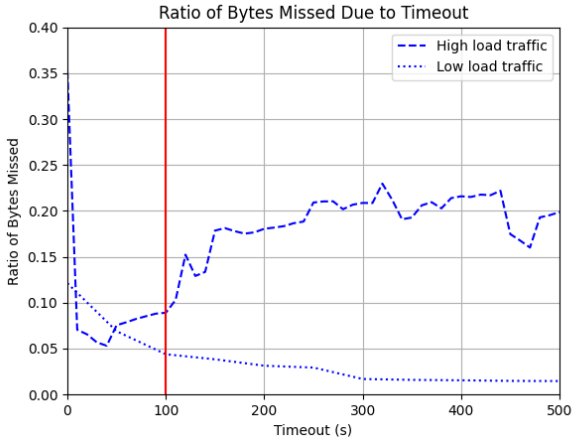

Figure 6: Ratio of traffic missed with limited memory under varied timeout values for the 15-minute high load vs 3-hour low load traces. The red line marks the timeout used in our Tofino implementation. (Missed traffic does not include traffic missed due to parser limitations).

4.3 DNS Response Table Memory Limits

The next set of experiments involved testing Meta4 under a variety of memory constraints by varying the total amount of register memory in addition to the number of stages as discussed in section 3.2.2. Fixing a timeout of 100 seconds (as chosen in section 4.2), we assessed the ratio of traffic missed (in bytes) under the varying memory configurations.

The results from Figure 7 show that at the memory limitation of 2^{16} entries, the limitation 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 if nonexistent benefits to using 1 vs 2 vs 4 vs 8 stages. However, when the amount of traffic begins to overwhelm the system (as in the case where there are 2^{12} or 2^{10} entries), using 2 stages as opposed to 1 stage can cut down the ratio of traffic missed significantly (0.33 to 0.23) in the case of 2^{12}. This occurs because as the data structure

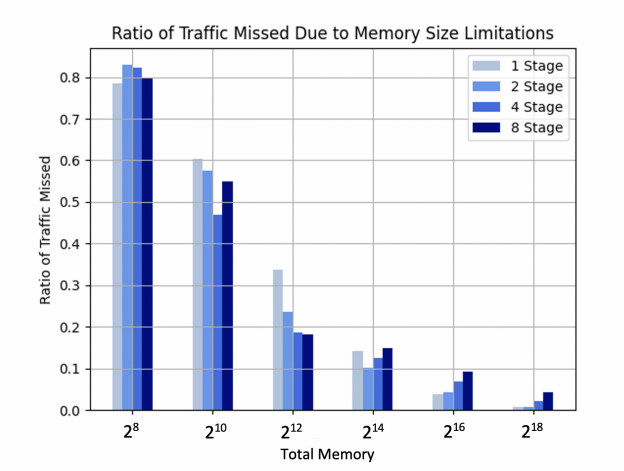

Figure 7: The x-axis shows selected total memory lengths. The y-axis shows the ratio of traffic missed for different memory configurations of Meta4 (not including traffic missed due to parser limitations).

fills up, the probability of hash collisions increase. Providing at least one extra stage to hedge against hash collisions can provide a significant boost in the performance of the data structure under these conditions. However, as you increase the number of stages to 4 or 8, the actual benefit of hedging against hash collisions starts to become outweighed by the smaller amount of memory in each stage. Thus, in the case of 2^{12} memory length, the greatest benefit in data structure performance comes from moving from 1 to 2 stages, but the benefit wrought from going to 4 or even 8 stages is rather minimal.

4.4 Correction for DNS Response Misses

Finally, we also wanted to assess the efficacy of the data correction based on the number of DNS responses missed as discussed in section 3.3. Using the ratio of the number of missed DNS responses for a particular domain to the number of DNS responses that we were able to fit in our DNS Response table, we scale up the number of packets and bytes seen for that domain.

In Figure 8, 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 is applied. We excluded domains with only a single defined label (domains like *.com, *.*.edu, etc). In this case, we are running on the configuration that our Tofino implementation uses (2^{16} total memory length). We see that the correction scaling applied makes a significant difference for most of the domains, often cutting down the error by more than half. This improvement is not universal, however. For example, for the domains *.zoom.us or outlook.office365.com, the scaling correction actually in-

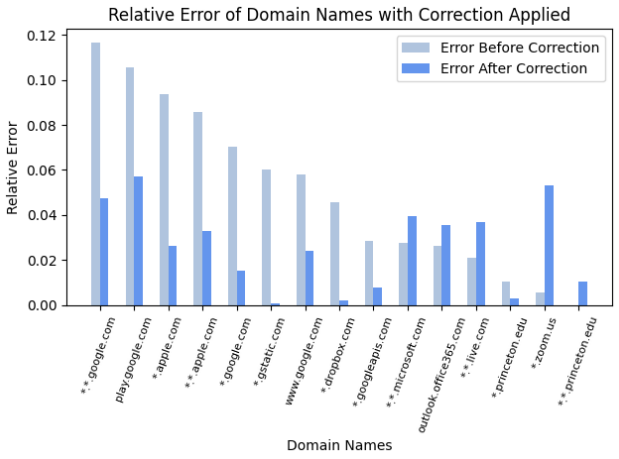

Figure 8: For the top 15 domains (by DNS response count), we show the relative error compared to the ground truth number of bytes for each domain before and after our scaling correction is applied. This graph was produced with a memory limitation of 2^{16} total register entries.

creases 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. This makes sense as services like Zoom or Microsoft Teams provide varying amounts of traffic per user depending, among other things, on the length of a video call. 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.

5 Hardware Prototype and Deployment

In this section, we discuss the process and challenges of implementing and deploying Meta4 in hardware. Meta4 was implemented using P4-16 for a Tofino switch. The development process was an iterative process, requiring many tests to see how much we could stretch the limits of the hardware's resources. The final parameters of our hardware implementation are summarized in Table 2. Based on traffic lost both due to parsing limitations and memory limitations, this hardware configuration of Meta4 had a final ratio of traffic lost (by bytes) of 0.142 when run on our 3-hour test trace.

Meta4 was deployed using the infrastructure provided by P4Campus [10], a Princeton University initiative to help researchers run experiments for P4 programs on live hardware switches on a campus network. Specifically, we tested Meta4

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

Table 2: Final parameters of Meta4 hardware implementation for Tofino switch. Note that "DRT" stands for "DNS Response Table".

on the P4Campus passive analytics testbed where anonymized mirrored traffic is delivered.

Below, we discuss two challenges of our Tofino hardware implementation: parsing DNS response packets (section 5.1) and dealing with limited stages in the processing pipeline (section 5.2). A repository with code for the hardware implementation can be found below[3].

	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 9: An overview of resource usage in Tofino by Meta4 in the processing pipeline.

5.1 Parsing DNS Packets

In section 3.1, we discussed how we were able to parse variable-length domain names using different fixed-length parser states for parsing domain labels. Since the parsed header fields that we store a domain's labels in have to be matched against match-action table rules to assign a domain ID, we had to ensure that the parsed domain labels can all collectively fit within the TCAM restraints of the Tofino hardware. Thus, we were relatively judicious in how we stored domain name labels in our parsed packet header fields. Instead of having a separate field for every possible domain label length, we instead stored header fields for 1, 2, 4, and 8 bytes for each domain label. With these header fields in powers of two, we were able to store domain name labels of any length up to, and including, 15 bytes. For example, if a

[3] https://github.com/jkim117/Meta4

domain label was 7 bytes long, we stored it in the 1, 2, and 4 byte-long header fields. If a domain label was 10 bytes long, we stored it in just the 8 and 2 byte header fields. This solution allowed us to maximize the number of domains we were able to parse while minimizing the total amount of TCAM in the match-action table we consumed (see Figure 9 for TCAM resource consumption by the domain match-action table).

Another 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 entries) for a domain name often includes a variable number of 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 P4 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.

5.2 Limited Number of Stages

As discussed in section 2.2, the processing pipeline for a PISA switch is restricted to a limited number of stages. A programmer is limited by the ALU to a restricted number of operations per stage. Furthermore, one cannot access or modify a register in a stage more than once. This particular restriction proved problematic for our implementation of our <cIP, sIP, D> tuple data structure. For a given DNS response packet, the program first checks to see if the client IP/server IP pair matches for an entry in the register data structure. If it is a match, the program just updates the timestamp. If the IP addresses do not match, the program then checks the timestamp of the entry. If the timestamp is timed out, the program replaces the entry with the new <cIP, sIP, D> tuple.

The problem is that replacing the entry requires replacing the client IP, server IP, and timestamp entries, all of which have already been accessed by the program. Since this operation is not allowed within a stage, we had to use a solution involving packet resubmission, a feature of PISA switches discussed in section 2.2. This allows us to essentially double the number of stages the program has to work within the ingress pipeline. In particular, we determine if a <cIP, sIP, D> entry needs to be replaced on the first pass for a DNS response packet. If we decide the entry should be replaced, we resubmit the DNS response packet and replace the old <cIP, sIP, D> entry with the new DNS response packet on the second pass. Because we are only resubmitting DNS response packets, any overhead due to resubmission is negligible. For example, in our 3-hour trace used for testing in section 4, 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 going through the switch, resubmitting these packets adds very little additional overhead.

Even with resubmission, however, we ran into problems trying to fit the entire Meta4 implementation into the limited stages of the ingress pipeline. In order to further expand the number of stages available to us in the processing pipeline, we divided the program between the ingress and egress pipeline. For the Meta4 traffic volume measurement program, for example, we moved the data structures counting packets and bytes indexed by domain ID to the egress pipeline. In general, we decided to keep the DNS Responses Table in the ingress for all Meta4 programs. Any subsequent data structures unique to a particular use case were confined to the egress pipeline (such as the DNS Statistics Table). The result is that we can again double the effective number of stages available for processing for a single packet. This particular method was possible because of our deployment of Meta4 within the passive analytics testbed of P4Campus where we conducted passive monitoring on mirrored traffic.

6 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 3 and will evaluate their performance based on campus deployments at Princeton University.

6.1 Measuring Traffic Volume by Domain

First, we re-visit the use case that we have been using as our running example. We ran our traffic volume measurement program on a Tofino switch with the same 15-minute/3-hour traces from Princeton University's campus network that we discussed in Section 4.2 (see Table 1). It is important to note that the 15-minute trace was captured while school was in session during mid-afternoon while the 3-hour trace was captured while school was out of session during the morning. To create our policy configuration, we used previous campus traffic, processed off the switch, to determine the most popular domains requested through DNS. In Table 3 and Table 4, we show the top 10 services by bytes for the 3-hour and 15-minute traces respectively. Note that we omitted domains with only one specified label (*.com and *.*.net for example) from this list. It is worth noting that Microsoft Teams, which contributes a third of traffic seen in the 3-hour trace, is a service that is used heavily by the university's Office of Information Technology (OIT).

In addition, as described in section 2.1, we used these results to determine the amount of traffic that was associated with various "traffic classes". In particular, we defined 12

Domain Name	DNS	Packets	Bytes
Skype/Microsoft Teams	15.3%	33.4%	33.4%
Google Ads	21.4%	27.5%	30.0%
Google (general)	4.1%	16.8%	14.6%
Apple (general)	0.5%	4.2%	4.1%
Zoom	0.4%	3.3%	3.3%
YouTube	0.5%	1.5%	1.6%
Facebook	0.3%	1.4%	1.4%
Microsoft (general)	0.5%	1.1%	1.2%
Akamai	0.03%	1.0%	1.0%
Instagram	0.02%	1.0%	1.0%

Table 3: Top services by bytes from anonymized 3-hour Princeton network trace. This trace was captured from 08:00-11:00 in the morning.

Domain Name	DNS	Packets	Bytes
Steam Games	0.005%	17.5%	16.7%
Facebook	1.3%	11.8%	12.0%
Google Ads	21.1%	7.0%	7.1%
llwnwd (CDN)	0.01%	6.2%	6.4%
Skype/Microsoft Teams	3.6%	6.6%	6.3%
Google (general)	15.4%	6.0%	6.1%
Reddit	0.05%	3.7%	3.8%
iCloud	0.5%	3.6%	3.7%
Instagram	0.1%	3.5%	3.7%
constitution.org	0.0002%	3.1%	3.3%

Table 4: Top services by bytes from anonymized 15-minute Princeton network trace. This trace was captured from 15:00-15:15 in the afternoon.

different traffic classes and grouped the domains from our known domain list under each of those classes. The result of measuring traffic volume by traffic class are shown in Table 5 and Table 6 for the 3-hour and 15-minute traces respectively.

6.2 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 [3].

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.

Traffic Type	DNS	Packets	Bytes
Video Call/Communication	15.9%	37.4%	37.3%
Advertising	21.4%	27.5%	30.0%
Misc Technology	3.6%	20.0%	17.8%
Other	20.0%	4.5%	4.5%
Social Media	0.5%	2.7%	2.6%
Entertainment	0.6%	2.2%	2.3%
Web Search	1.6%	2.4%	2.2%
Productivity/Education	35%	1.6%	1.7%
Cloud Computing/Storage	0.7%	1.0%	0.9%
Shopping	0.1%	0.4%	0.3%
News	0.03%	0.2%	0.3%
Government	0.4%	0.04%	0.06%

Table 5: Traffic volume by traffic class from anonymized 3-hour Princeton network trace. This trace was captured from 08:00-11:00 in the morning.

Traffic Type	DNS	Packets	Bytes
Other	15.1%	22.1%	22.3%
Entertainment	1.8%	22.6%	21.7%
Social Media	1.9%	20.4%	20.9%
Misc Technology	17.1%	8.4%	8.7%
Video Call/Communication	6.6%	9.0%	8.7%
Advertising	21.2%	7.0%	7.1%
Cloud Computing/Storage	5.7%	5.4%	5.5%
Productivity/Education	26.0%	2.3%	2.3%
Web Search	2.3%	1.4%	1.4%
Shopping	1.6%	0.7%	0.7%
News	0.3%	0.6%	0.6%
Government	0.3%	0.1%	0.1%

Table 6: Traffic volume by traffic class from anonymized 15-minute Princeton network trace. This trace was captured from 15:00-15:15 in the afternoon.

As seen in Figure 10, for DNS response packets, the program uses a hash of the server and client IP addresses (just like for the traffic volume use case for Meta4) to create a table entry with an associated timestamp. Unlike the traffic volume case for Meta4, there is no domain ID that needs to be stored. After creating the entry in the DNS Response table of Figure 10, a counter register keyed by the client IP is incremented.

For a non-DNS data packet, the source and destination IPs are hashed to key into the DNS Response table as seen in Figure 11. 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 from Figure 10 was followed by actual traffic. In the case that a DNS response is never followed by additional traffic, the entry in the DNS Responses table will eventually time out (after five minutes) and the counter in the

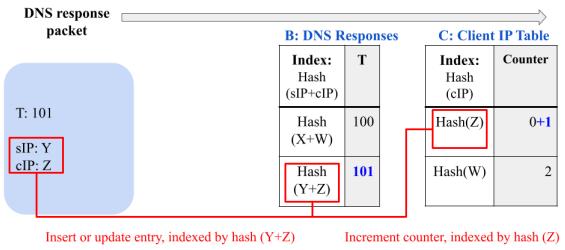

Figure 10: Meta4 DNS-Tunneling response to a DNS response packet.

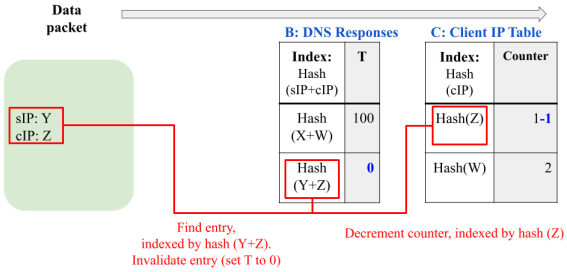

Figure 11: Meta4 DNS-Tunneling response to a non-DNS data packet.

Client IP table will not be decremented, indicating that the DNS response was never followed by additional traffic. A client entry in Client IP table with an abnormally high count is a suspect for DNS tunneling.

To test our DNS Tunneling application, we generated two instances of DNS tunneling traffic using the open-source dnscat2 tool [4] and embedded them with normal, benign traffic. The first was a case where the client made an ssh connection through a DNS tunnel and performed simple commands like "cd" or "ls". The second was a case where the client made an scp connection through a DNS tunnel to make a data transfer of 3.5 MB. We made a control-plane script that queried the registers every 30 seconds to see if there were any positive detections. A positive detection was defined as a client having more than five DNS response packets not followed by additional traffic. In our test, we had zero false positive detections and were able to succesfully detect both instances of 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 of the initial attack. 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 35089 out of 75078 DNS tunneling packets.

6.3 IoT Device Fingerprinting

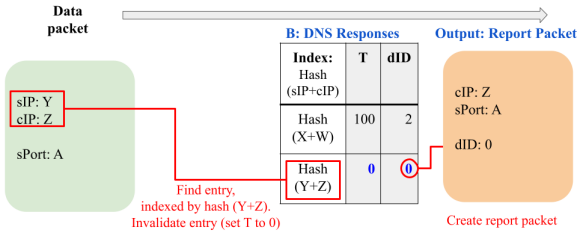

Figure 12: Meta4 IoT-Fingerprinting response to a non-DNS data packet.

By tracking what domains a device speaks to, we can relatively easily detect if it is a particular IoT device. Previous work by Saidi et al. has shown that fingerprinting IoT devices by using rules that define a domain and server port can be quite accurate. In an experiment by Saidi et al. involving 33 devices from different manufacturers, 97% were able to be detected within 72 hours of active monitoring [15].

We sought to implement the same system within the data plane using Meta4. By using the domain names found by Saidi et al. associated with IoT device traffic, we are able to generate a known list of domains (and thus match-action table rules). The response of this program to a DNS response packet is nearly 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 then created for the DNS Responses table using a hash of the server and client IP addresses to create the key for the entry. 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 is seen that has a source and destination IP address that matches an entry in the DNS Response table, (see Figure 12), we set the timestamp in the entry to 0 to indicate that the entry is free for re-use. We then take the domain ID, client (destination) IP address, and the source port of the data packet and forward a report packet to the control plane. By accumulating a long-term record of the domains/ports that certain client IPs communicate to, we can build a profile that may eventually lead to a positive detection of an IoT device. Due to the convoluted nature of rules

associating domain/ports to IoT devices (many IoT devices have multiple or overlapping detection rules), we leave the detection decision making logic up to the control plane.

To evaluate our IoT fingerprinting use case, we ran our program with 10 hours of traffic from a IoT traffic dataset collected by IMPACT (Information Marketplace for Policy and Analysis of Cyber-risk and Trust) [7] known to contain traffic from seven IoT devices from the device detection list from Saidi et al. In Table 7, we show how many unique rules we were able to detect from each of the seven devices where a "rule" is defined as a domain-port number pairing that indicates a positive detection of a particular device. For each of the seven devices, we were able to get at least one positive detection for each device without any false device detections, confirming the effectiveness of our fingerprinting application.

Device Name	Number of Unique Rules Detected
Alexa Fire TV	18
Philips Hub	5
Alexa Echo Dot	8
TPlink smart bulb	1
TPlink smart plug	2
Amcrest Camera	1
Wansview Camera	2

Table 7: Number of unique IoT rules detected for each of the devices included in the 10-hour evaluation trace.

6.4 Other Use Cases

In this section, we provide examples of other potential use cases that could be implemented with the Meta4 framework. We did not create or test hardware implementations for these use cases.

Web Fingerprinting By creating detection rules defined by the amount of traffic in terms of bytes/packets sent from certain domains, we can also create a form of web fingerprinting. We can create a known list of domains from all the domains associated with a webpage of interest. In many ways, this application would work very similarly to the standard traffic volume measurement case. For each of the domains in the known list, we keep track of how much traffic (in terms of packets/bytes) that we see for each client. The basic concept is that for certain webpages, when a client downloads that webpage, a relatively consistent set of domains are requested by the client. Furthermore, a relatively consistent amount of bytes are downloaded from each of those domains. Together, a set of domains and an amount of bytes for each of those domains could potentially be a strong fingerprint for a webpage.

IDS Bypass by Domain If a network operator notices that there is a lot of traffic associated with a few, trusted domains, they could potentially improve the performance of the network by allowing traffic from those domains to bypass intrusion detection (IDS) processing. For example, if a network operator notices a large increase in Zoom traffic that is hurting overall performance for an IDS system, they could use Meta4 to identify traffic associated with the *.zoom.us domain name and allow those packets to bypass IDS processing.

Traffic Inspection/Blacklisting by Domain Similar to the IDS bypass use case, a network operator could also seek to blacklist certain domains. Meta4 could be used to identify traffic associated with certain problematic domains and either re-route that traffic off the switch for further analysis or rate-limit that traffic.

7 Related Work

Intentional Network Monitoring The basic design and functionality of Meta4 was inspired by the ideas behind intentional network monitoring, pioneered by the NetAssay system [6]. 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 sought to 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 [1] 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. The concept of altering how traffic is handled within a network based on its domain name also played a key part in informing how Meta4 was developed. Meta4 differs from NetAssay in that it was implemented entirely in the data plane and focuses primarily on use cases (such as IoT fingerprinting, web fingerprinting, etc) that take advantage of the <cIP, sIP, D> tuple that can be extracted from DNS response packets.

Parsing Domain Names Previous work in developing solutions for parsing variable length domain names in PISA-based switches greatly helped to inform the solution that we implemented for Meta4. P4DNS [17], an implementation of an in-network DNS server in P4, offered the solution of parsing different domain names in different, pre-determined, fix-length parser states. The P4 Authoritative DNS Server [11], another attempt at implementing a local DNS server in P4, expands on the ideas of P4DNS, and parses the separate labels of a domain name into separate fixed-length parser states. We expanded on these ideas in designing our domain name parser for Meta4 to optimize the efficiency and performance of our parser. In particular, as discussed in section 5.1, we stored domain name labels in header fields of size 1, 2, 4, and 8 bytes so that we could minimize the amount of metadata we needed to store while also allowing us to maximize the length of domain labels we could parse. In addition, we standardized the number of bytes (15) that could be parsed for each label of a domain name in order to allow for more flexibility in the domains that we could parse (as opposed to different pre-set

maximum lengths for different labels that could be parsed as in the P4 Authoritative DNS Server).

Data Structure Implementation in P4 In addition, many of the ideas used to implement the data structures of Meta4, especially the <cIP, sIP, D> register data structure (DNS Response table), were inspired by previous P4 projects that similarly required fitting large amounts of packet information into a limited amount of register memory. PRECISION [2] and HashPipe [16], 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 provided 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 expanded on these ideas for our implementation of our DNS Responses table to create a two-stage data structure to handle hash collisions that also evicts entries (similar to HashPipe) for having timestamp values that had not been updated for a certain period of time.

8 Conclusion

This paper outlined the design and implementation of Meta4, a framework for network monitoring in the data plane that allows a network operator to associate traffic by domain name. Developed for PISA-based programmable switches, Meta4 parses DNS response packets to store client IP, server IP, and domain ID (<cIP, sIP, D>) tuples on registers in the switch. Using this register data structure, we are able to associate subsequent packets to a domain ID by matching the destination and source IP addresses to the client IP and server IP of a (<cIP, sIP, D>) entry respectively. This allows for a host of potential use cases including traffic volume measurement by domain name, DNS tunneling detection, IoT device fingerprinting, and webpage fingerprinting.

We implemented Meta4 on Tofino hardware using the P4 language and deployed Meta4 on Princeton University's campus. 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.

As indicated in section 6, there are many potential applications that can be developed using the Meta4 framework to run in the data plane. Future work that expands on what we have done here could include creating data-plane implementations of domain-based web fingerprinting, IDS bypassing by domain, or domain blacklisting. Another possible direction is creating additional tools that work off of the Meta4 framework to further help network operators to accomplish their intent without worrying about low-level network details. For example, one application, similar to the Sonata network querying system [8], could allow network operators to make queries for traffic based on domain name or even higher-level abstractions such as queries for video-streaming traffic, or queries for education-related traffic. Finally, Meta4 specifically focuses on using DNS response packets to join IP address information to domain names. This form of intentional network monitoring, however, can be expanded beyond just DNS. For example, one could monitor traffic based on a join of a client IP address and a userid from wifi. In addition, one could again join domain name and IP address information but by using SNI (included in TLS handshakes) instead of using DNS.

Acknowledgments

I would like to thank Professor Jennifer Rexford and Dr. Hyojoon Kim for advising me during this project. Their constant willingness to help, offer advice, and provide feedback was absolutely crucial at every step. I am very grateful for their patient guidance and understanding especially during the challenges presented by the pandemic. I would also like to thank Professor David Walker for providing additional feedback and reviewing this work. Finally, I would like to thank the Princeton Computer Science Department and Office of Information Technology for providing the institutional support and infrastructure to implement and deploy Meta4.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *ACM Symposium on Operating Systems Principles*, pages 186–201, 1999.

[2] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[3] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium*, pages 365–379, 2012.

[4] Ron Bowes. dnscat2. https://github.com/iagox86/dnscat2, 2013-2015.

[5] Chromium. Issue 164026dns TTL not honored. https://bugs.chromium.org/p/chromium/issues/detail?id=164026, 2018.

[6] Sean Donovan and Nick Feamster. Intentional network monitoring: Finding the needle without capturing the haystack. In *ACM SIGCOMM HotNets Workshop*, 2014.

[7] Information Marketplace for Policy, 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.

[8] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371, 2018.

[9] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, pages 90–106, 2020.

[10] Hyojoon Kim, Xiaoqi Chen, Jack Brassil, and Jennifer Rexford. Experience-driven research on programmable networks. *ACM SIGCOMM Computer Communication Review*, 51(1):10–17, 2021.

[11] Xiazhou Li. P4 authoritative DNS server. *Barefoot Networks*, 2017.

[12] mozillaZine. Network.dnscacheexpiration. http://kb.mozillazine.org/Network.dnsCacheExpiration, 2010.

[13] PerfMatrix. Think time: Importance of think time in performance testing. https://www.perfmatrix.com/think-time/, 2019.

[14] The Spamhaus Project. https://www.spamhaus.org, 2021.

[15] Said Jawad Saidi, Anna Maria Mandalari, Roman Kolcun, Hamed Haddadi, Daniel J Dubois, David Choffnes, Georgios Smaragdakis, and Anja Feldmann. A haystack full of needles: Scalable detection of IoT devices in the wild. In *ACM Internet Measurement Conference*, pages 87–100, 2020.

[16] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM Symposium on SDN Research*, pages 164–176, 2017.

[17] Jackson Woodruff, Murali Ramanujam, and Noa Zilberman. P4DNS: In-network DNS. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2019.