Practical Handling of DNS in the Data Plane

Alexander Kaplan, Shir Landau Feibish
The Open University of Israel

ABSTRACT

The Domain Name System (DNS) is a significant component of modern-day internet. Despite this fact, DNS traffic is mostly unencrypted, and as such a likely target for exploitation by malicious actors. The advancement of programmable switches presents researchers with the opportunity to explore DNS traffic from a new vantage point, without sacrificing network bandwidth. In spite of the incentive, DNS research in programmable switches has been scarce, owing to the difficulty in parsing DNS packets.

We present a general solution to DNS packet parsing that can handle the vast majority of DNS packets (97%) using current hardware and can easily be scaled to parse all DNS packets as hardware improves. Our highly configurable solution can be adjusted to fit many distinct use cases. Additionally, we explore the challenges involved in parsing DNS packets and present common pitfalls appearing in previous research attempting to do so.

CCS CONCEPTS

• **Networks** → **Network protocols**.

KEYWORDS

DNS, Programmable Switch, Programmable Networks, Data Plane, Network Measurement

ACM Reference Format:
Alexander Kaplan, Shir Landau Feibish. 2022. Practical Handling of DNS in the Data Plane. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '22), October 19–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563647.3563654

1 INTRODUCTION

DNS is a critical infrastructure of the Internet, used to resolve a human-readable address into an IP address. As an integral part of modern day internet communication, DNS is a prime target for exploitation by malicious actors. Attacks based on the direct or indirect vulnerabilities [1, 4] of the DNS protocol have been rising in recent years, culminating in a 2 Tbps DNS based DDoS attack at the end of 2021 [9].

Programmable switches provide us with a new vantage point for analyzing DNS traffic, thus allowing the creation of new and better mechanisms for securing DNS requests and responses. Programmable switches enable user-defined packet parsing; however, parsing is limited both in depth (i.e. the number of bytes that can be parsed) and in the ability to parse variable length segments. These limitations make parsing most packet payloads challenging, with DNS payload presenting a particularly difficult case.

Not only can a DNS payload be very long, it is also made up of a *varying* number of *variable length* parts. The DNS query received as part of the payload is constructed by splitting the domain into labels, and preceding each label with a byte containing the label length. The end of the query is marked with a 0 byte. As an example the query for **m.example.com** will be constructed as **(1)m(7)example(3)com(0)**. This means that when parsing the packet, the parser is unable to tell the total length of the query until it reaches the last part of the query, that is, if the length of the packet even permits us to do so. To further complicate the issue, the (main) domain part of the query is given only at the end, however, if the query is too long, this part may not be parsed, therefore presenting an issue with identifying and using this crucial bit of information.

Parsing a variable number of variable length items may seem to be an easy problem. Indeed, several solutions have been proposed for parsing DNS queries in a programmable switch. While providing insight into the issues around parsing DNS, the solutions only partially resolve the problem. Meta4 [16] and P4DNS [21] deliver solutions with concrete limits in parsing. These solutions parse DNS packets to a preset limit, without deep analysis of the query content (such as subdomain extraction). Our previous research, WORD [14], provides a more dynamic solution yet is implemented in a software switch. The result turned out to be hard to transfer into a hardware switch without significant changes. P4DDPI [3] solves the problem of parsing depth by using packet recirculation. This approach allows for parsing of the entire DNS query but has a severe disadvantage, since parts of the DNS query may need to be discarded with each recirculation. As a result the packet exiting the switch will not necessarily contain all the information it held when entering the switch, resulting in a damaged packet. We further elaborate on this issue in Section **2.2**.

In this paper, we provide insight into how DNS payload may be parsed. We create a system which can correctly parse as much as 97% of DNS queries using today's hardware, while being easily scalable to further increase this number as programmable switches improve. We summarize our contributions as follows:

- A parsing algorithm for DNS that is non-destructive (**§2.2**), scalable (**§3.2**), practical (**§3.3**), and provides good coverage of DNS requests with hardware available today (**§5.1**). The parser cautiously traverses the query, parsing as much as possible without damaging the contents of the packet.
- A novel approach to processing headers by priority (**§3.3**).
- Sample implementations of three common use cases that may be easily implemented based on our solution, allowing new DNS capabilities in the data plane. We present DNS metrics (**§4.1**), caching (**§4.2**) and firewall (**§4.3**).
- An open source, fully configurable DNS parsing solution [15]. Our parsing solution can be used as the basis of any P4 project targeting DNS.

In the remainder of this paper we evaluate our solution using an Intel Tofino switch (**§5**), and finally discuss current trends in DNS communications and future work (**§6**).

2 CHALLENGES IN DNS PARSING

In this section we discuss the challenges involved in parsing DNS packets within the confined resources of programmable switches. In addition we explore related prior work, and explain why we believe a practical solution is still necessary.

2.1 DNS Structure

The Domain Name System (DNS) protocol [18] is an application layer protocol used to translate domain names into IP addresses. DNS requests are sent out with queries to DNS servers, that respond with the relevant answer. The DNS packet is made up of a header describing the content of the packet, and a payload which contains at least one query and, optionally, DNS answers. The unique structure of the DNS payload presents multiple challenges, presented below.

Multipart. The "domain" being queried in a DNS request is usually comprised of a domain and subdomain pair, where the domain is the main identifier of the site, and the subdomain is the rest of the query (e.g. in the query **mail.example.com** the domain would be **example.com** and the subdomain would be **mail**). From now on we will use the above terminology. To avoid confusion, the term query will be used to identify the combination of domain with subdomain.

Highly Variable. The difficulty in parsing DNS packets in programmable switches comes from the highly variable structure of the packets. Each label in the query is of a variable length, and the number of labels is also heavily varied. Furthermore, the number of labels is not given as a header in the packet, meaning the only way to know how many labels are contained in the packet is to parse all of them, until finally reaching the predefined ending character. While a variable length bit string type does exist in P4 [10], its functionality is extremely limited. Even if we were to choose to use this type, the variable amount of labels remains an issue.

Long and Complex. In addition to the ability to parse a DNS query, some use cases require the ability to parse the IP that is returned in the response (e.g. P4DDPI [3]). A DNS response packet will contain the original DNS query at the beginning of the packet payload, with the response data concatenated immediately afterwards. In order to extract the resolved IP from the DNS response payload we must be able to parse the DNS query completely, otherwise we risk malforming the packet (this is further discussed in Section **2.2**). We note the structure of the DNS query means the labels describing the domain appear only after the labels describing the subdomain, further increasing our motivation to fully parse the query in order to allow proper attribution.

2.2 Non-destructive DNS Parsing

One of the goals we believe is necessary for any P4 project to be considered practical is non-destructive parsing. We define parsing to be destructive when deparsing omits crucial information that was available before the packet was parsed. Since programmable switches don't allow parsing data of arbitrary length, other solutions need to be used to overcome this limitation. As we will show in this section, existing solutions cause a loss of part of the packet data, and thus other solutions are needed.

We first look at the usage of the 'advance' method. Used in solutions such as Meta4 [16], the 'advance' method allows skipping bits in a packet while parsing, potentially allowing further parsing of the packet. The problem is that skipped bits are removed from the packet entirely, resulting in the loss of the data the bits held. The 'advance' method is meant to be used to skip metadata properties that might be attached to the packet (such as port information). We note the use of this method is not unique to DNS and can also be seen in the OS fingerprinting solution provided by Bai et al. [6].

Another solution, used in P4DDPI [3], requires recirculating the packet multiple times. In the solution, each time the packet is recirculated part of the packet data is removed, allowing the same depth of parsing to reach previously unparsed data (as the data is shorter). The problem is the discarded data cannot be retrieved, as the recirculated packet cannot access data from its previous iterations unless stored in some other form in the switch. Since the data skipped is quite large, only a hash of it can be stored which would not help reconstruct the original packet. In the case of P4DDPI the parsed labels are discarded in each recirculation, to make room for additional labels. As an example, when parsing the query **a.b.c.example.com**, after the first recirculation the labels for **a.b.c.example** are dropped (as the label limit in P4DDPI is 4) and we are left with only one label **com**. The packet exiting the switch will therefore only contain the (invalid) query **com**. We note using this method will

always result in data loss, as in order to "defeat" the parsing limitation we have to free space in some way. Even ignoring the performance penalty (as rightfully pointed out in the paper, the performance gained compared to existing solutions heavily negates the performance lost by recirculation), this approach means the packet exiting the switch loses part of the query in the DNS packet.

We note that most programmable switches support packet cloning, which would allow recirculating the cloned packet, while the original packet continues. However, this solution lacks the ability to take action (i.e. block) on the original packet since the original DNS packet already left the switch.

Both of these approaches for parsing of DNS packets suffer from the same major flaw; the packet leaving the switch is not the same as the one entering it. While acceptable for academic purposes it means these solutions in their current form cannot be used in practice, as network data cannot be sacrificed for additional switch functionality.

3 DNS PACKET PROCESSING

With the above challenges in mind, we present our solution for DNS parsing and handling in the data plane. In this section we explain how our parsing algorithm makes sensible compromises in order to achieve useful and scalable parsing, and further explain how domain and subdomain data can be effectively extracted from the parsed data.

3.1 Overview

The purpose of our algorithm is to overcome the main hurdles associated with DNS parsing, allowing the code following the algorithm to directly address all parts of DNS payload as if they were simple headers.

Our approach to dealing with the dynamic structure of DNS queries is to extract fixed size pieces of the payload in each parsing state. We create numerous parsing states per label that may be called in various combinations to achieve the exact label extraction for any given length. Our focus is on exact label extraction as any other method would prevent us from comparing labels directly. Additionally, we define parameters for the algorithm that provide concrete limits for the parsing process, allowing us to stop early in cases that would exceed our hardware resources. We attempt to achieve this while minimizing the amount of parsing states expended for extraction, all the while taking care not to create parsing loops that might hinder compilation.

Our parsing algorithm is presented in graphical form in Figure **1**. Each rectangle in the figure represents a parsing state, while rhombuses in the figure signify conditional transitions between states, implemented as P4 select statements. The main parsing states used to extract the query labels are highlighted for convenience. We note that while the label

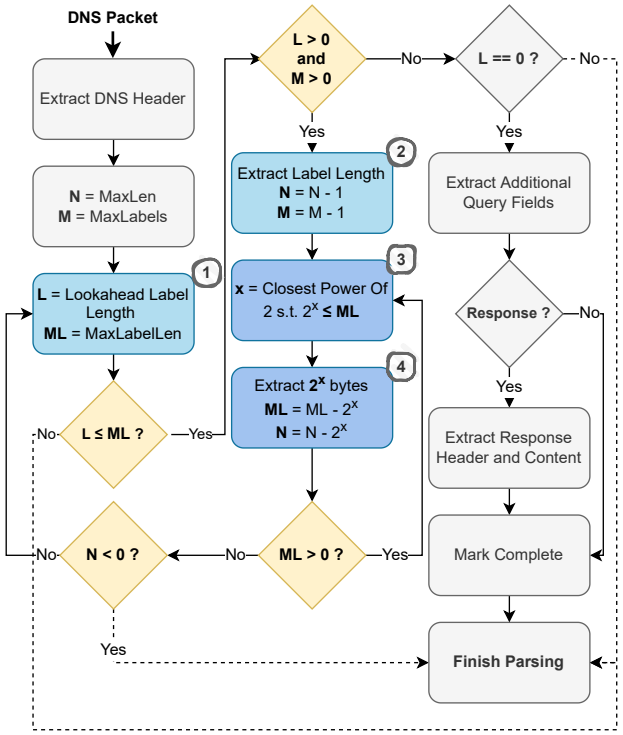

Figure 1: Parsing DNS Packets

extraction is presented as a loop, the underlying states are duplicated for each label in the order they appear. For each label, the algorithm reads the length of the label and based on the value will go through multiple stages of extracting constant sizes of bytes that amount to exactly the label size. The exact operation of the algorithm is explained in more detail in Section **3.2**.

Once we finish parsing the DNS packet we will shift our focus to extracting the domain and subdomain from the DNS query. In addition to having access to each label of the query directly, the domain and subdomain will be hashed to allow easier handling as a constant sized variable. In Section **3.3** we explain our unique approach to this extraction.

3.2 Parsing DNS Payloads

Since we know we are unable to parse every DNS request, our parsing algorithm will focus on providing sensible compromises that can be adjusted to fit most use cases. As hardware implementations improve, the algorithm can be scaled up, eventually reaching full DNS parsing.

In Table **1** we present the notation used throughout this paper, including the algorithm description in Figure **1**. *MaxLen*, *MaxLabelLen* and *MaxLabels* serve as parameters to our algorithm, defining which DNS queries we'll be able to parse; once any one of the parameters is exceeded, parsing will

halt. The user can adjust these parameters to optimize the resources designated to this process.

Parsing Algorithm. We approach parsing by addressing each label in order of appearance. For each label we first check if we are able to read the label entirely by comparing the label length (first byte before the label) to *MaxLabelLen*. We do so before extraction in order to avoid partially parsed labels. If the length is within our parameters we extract it and begin parsing the label. This part of the algorithm is shown as states 1 and 2 in Figure **1**. Once we read the label length we need to begin parsing the label. A naive approach to this problem would be to extract the entire label at once. As we would need to create a header for each size of each label, this approach would require $MaxLabels \cdot MaxLabelLen$ headers. Not only is this approach not scalable, since we don't know which headers are occupied we would need to address all the headers simultaneously or waste resources attempting to locate the single relevant header for each label.

Our approach to parsing the label is shown in states 3 and 4 in Figure **1**. Each state extracts a predefined number of bytes that is an exact power of 2, a process that is repeated until the entire label is extracted. For example, with the label **example** we would parse 4 bytes, then 2 and finally 1. This example is also shown in Figure **2**.

All the highlighted states in Figure **1** are replicated for each of the *MaxLabels* we are able to parse. Additionally, states 3 and 4 are replicated for each power of 2 that is smaller than *MaxLabelLen*. More generally, parsing states 3 and 4 are defined together as LP_i^j for $i \in [1, MaxLabels]$ and $j \in \{2^k | \forall k \in [1, floor(\log_2(MaxLabelLen))]\}$, where each such state extracts j bytes from the i-th label. This means that while our algorithm seems to loop, it actually moves between distinct states, passing through each state at most once, guaranteeing a simple parsing graph. As an example, given the query **example.com** we would use the following states in left-to-right order: $LP_1^4, LP_1^2, LP_1^1, LP_2^2, LP_2^1$, each time extracting a number of bytes that is equal to the largest power of 2 that is smaller than the leftover number of bytes from the label. Our method of parsing guarantees a singular order of parsing for a given label length, despite states appearing in various length extractions. Our approach generalizes the approach taken by Meta4 [16] while resulting in a parsing chain that has $O(MaxLabels \cdot \log(MaxLabelLen))$ states, as opposed to $O(MaxLabels \cdot MaxLabelLen)$ in Meta4. This improvement allows better scaling when increasing both of these parameters.

Using a counter N initialized to *MaxLen* we control the total length of data parsed at any point. Using N we have a more fine-grained control of the total depth of parsing, which is required by some hardware implementations. Some DNS queries may have many labels yet have a small total length (e.g. **a.b.c.d.e.f.example.com** which occupies only 23 bytes). Maintaining a total limit of *MaxLen* allows us to approach these queries, a use case required by some security applications such as WORD [14].

Table 1: Notation used in the paper

Notation	Definition
MaxLen	Maximum amount of characters to be parsed from the entire query
MaxLabelLen	Maximum length of a single label that can be parsed in bytes
MaxLabels	Maximum amount of labels that can be parsed
MaxDomainLabels	Maximum amount of labels matched for domain attribution

Finally, after successfully parsing the DNS query in a packet we may try to parse the response. Parsing is done by relying on DNS compression [18], which means the query is not repeated in the response. DNS compression is widely used, and means we are able to parse the rest of the response as a constant header. Due to constraints from hardware today we chose to focus only on constant size responses (more on this in Section **5**).

Partial Parsing. If we exceed any of our parameters during parsing, our parsing process will be interrupted. The resulting packet will contain some unknown part of the total DNS query; since we could hit our limit as early as the first label parsed (e.g. if the first label is longer than *MaxLabelLen*), this could be a very small fragment. In any case we will not be able to parse response data, as this is provided after the end of the query.

To better illustrate the issue let us look at the case where *MaxLabels* is set to 3 and we receive a packet with the query **mail.google.com.mali.cious.net**. After parsing 3 labels we are left with the query **mail.google.com**. This case could result in attributing behavior wrongly, and has the potential of being exploited by a malicious actor.

Since we aim to provide a practical solution for a general use case our algorithm does not enforce skipping partially parsed packets. Instead in our implementation we choose to enforce this rule directly. We encourage any solutions based on our implementation to adopt this approach, or address the problem in some other way.

3.3 Extracting the Domain

Following the parsing steps, we obtain the extracted labels of the DNS packet. In order for this information to be more useful we would like to be able to split it into two parts, domain and subdomain. For the purposes of this work we define the domain to be TLD (top-level domain) with one additional label, while the subdomain contains all the remaining labels (note that the subdomain might be an empty

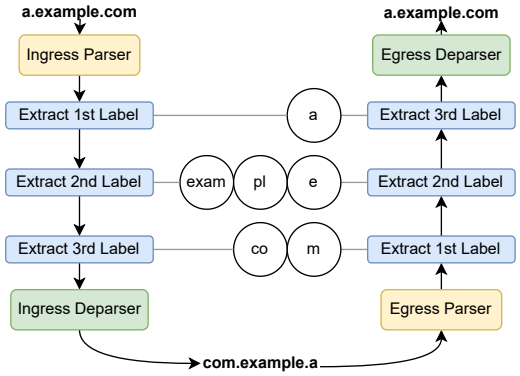

Figure 2: Parsing Process Example

string). To illustrate we look at *a.test.example.com*; our domain is *example.com*, while the subdomain is *a.test*. The split into domain and subdomain is useful for attributing multiple packets to a single owner, whether for measurement [7] or security [14] purposes.

As discussed in Section **2**, the structure of the DNS packet holds the domain information in the last labels of the query. As a result we are unable to easily gauge the starting position of the domain labels. Matching without knowing the starting position of the domain labels would require use of additional TCAM resources to compare for every potential start position, and due to the vast number of bits involved in a DNS query could also require the use of multiple tables for the comparison. In an already constrained environment such waste of resources is problematic.

Our solution for extracting the domain relies on the implementation of separate ingress and egress pipelines available in some switches [10], including the Intel Tofino [13]. In this type of switch the packet will go through parsing and deparsing twice, once for the ingress pipeline and once for the egress pipeline. As shown in the example in Figure **2** we use this two step design to our advantage by reversing the order of our labels **twice**, once after the first pipeline and once after the second one. The result is our packet comes out of the router unaffected, while the second pipeline receives the DNS query in reverse order. This works to our advantage since we receive the domain labels at the start of the parsed query, simplifying the task of extraction to a few TCAMs (match-action tables) that determine at which label the domain ends and the subdomain begins.

In order to split our query into two parts we make use of the Public Suffix List (PSL) published by Mozilla [11]. The PSL aims to be an exhaustive list of functional TLDs (named suffixes), used by multiple browsers for domain highlighting [11]. Using a table populated with these suffixes we can match the (now first) labels of our DNS query to get the number of labels belonging to the domain. With the number of labels in hand we are able to separate the the domain labels from the subdomain labels. We note the use of the PSL as the best solution for common case DNS handling. More specific scenarios may benefit from smaller lists of TLDs which could require less TCAMs as well.

4 APPLICATIONS

We have implemented several applications to exhibit how DNS parsing in the data plane may be utilized. We have open sourced the code of these applications as well [15]. Additionally, we discuss how our solution can be used for more applications, such as security and other protocols.

4.1 Metrics

We can use the domain, subdomain or even specific labels to count how many requests containing certain attributes pass through our switch. In our implementation we store a count of first labels of the subdomain (e.g. **m** in **m.example.com**), in addition to storing a count of subdomain hashes. In our implementation we focused on storing the total count of packets for a metric (e.g. how many DNS queries we've seen with the subdomain **m**, regardless of domain), but it would be possible to use a solution such as Beaucoup [8] to create a unique count per domain. This information could be useful in assessing common patterns of internet access, achieved while the traffic is en-route, with no added latency.

4.2 Caching

A more ambitious use case we could facilitate is DNS caching. A DNS response contains both the original query and the IP it is resolved into (Looking at A records), which could be used as a cache source.

When receiving a DNS response our implementation will use the domain and subdomain hashes to calculate a numeric index. Using this index we will store the IP, TTL and a signature value composed from the original query. On receiving a DNS request that is mapped to the same index we will check the TTL to make sure our cache is not expired, compare the signature to validate a match, and finally wrap the original DNS packet with the request in a response header. The packet is then sent back to its sender as a DNS response.

Our implementation only caches responses which contain a single result. Even with this caveat we managed to achieve a 10% hit rate when replaying a 24 hour trace [20]. Since our cache is accessed en-route the response can arrive back to the client much faster, resulting in significant performance gain for the client while alleviating the load on DNS servers.

4.3 Firewall

Our breakdown into domain and subdomain is especially useful when trying to implement a DNS firewall. The firewall

will block DNS requests by following rules set in advance. We implement two different types of blocking mechanisms which could be used in different environments. *Domain block-list* support blocking off whole domains, regardless of the subdomain attached to them. *Subdomain allow-list* will only allow certain subdomains to be resolved, blocking off anything not explicitly allowed.

Our implementation of a firewall is relatively bare-bones, meant mostly to showcase the ability. For future research it is possible to use IPs resolved in DNS requests to block IP traffic based on a DNS allow-list/block-list.

4.4 Security

As discussed in the introduction to this paper, the security aspect of DNS communication is an extremely important one. As we are unable to parse all DNS queries we are left with a gap in our implementation that can potentially be abused by a malicious actor. In this section we cover some solutions that may help future implementations overreach this gap.

It is important to note the DNS-based attacks discussed in the context of programmable switches are usually volumetric. We can use our metrics application (**§4.1**) to count all packets we failed to parse, maintaining an up to date count. We can further use this information as an indication of attack (either with a threshold, or a relative change), and set our switch to drop unparsed packets. This method can be used in conjunction with any other security application that works for all packets we did manage to parse correctly.

An additional approach that can be used is forwarding some of the unparsed packets to the control plane for further inspection. Since we expect the attack to contain a large volume of packets, even sampling a small percentage of unparsed packets (which are likely to be related to the attack, once it starts) can allow us to notice a pattern of attack. Once the attack has been noted we can once again switch to drop unparsed packets, significantly reducing the attack surface.

We note even without any special consideration for unparsed packets a security application based on our solution would significantly reduce the type of queries that can be used in a DNS based attack, meaning a packet crafted specifically to circumvent switch detection would likely be easier to detect using traditional methods.

4.5 Other Protocols

Our paper focuses on the DNS protocol, a protocol that is particularly complex to parse in programmable switches. Still, other protocols exist which could benefit from the methods we've developed. One such example is the TCP-based MQTT protocol [19].

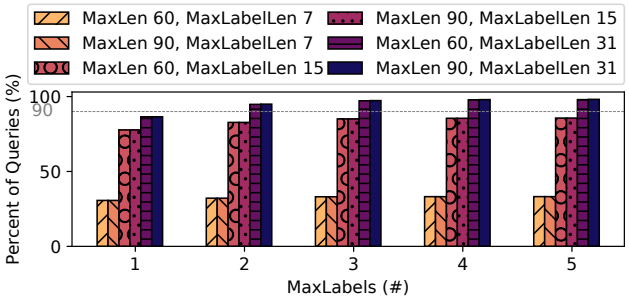

Figure 3: Distribution of DNS Queries

MQTT is an IoT focused messaging protocol, designed to provide lightweight publish/subscribe communication. MQTT is available in encrypted and unencrypted variants, but is commonly used unencrypted due to the costly overhead of encryption in low power devices [2].

MQTT contains various variable length fields both in its header and payload. Some of the fields are strings, such as username and password fields used for authentication. Our methods for parsing and processing DNS queries can be used to address MQTT communication, thus providing a better ability to process the various strings contained in the packet. This improvement could be used to implement an efficient in-network MQTT broker, which could provide a performance benefit over a software based solution when dealing with high message throughput, a result previously seen in MQTT communication implemented using programmable data planes [5].

5 EVALUATION

As our research is meant to provide a basis for general DNS research, we chose not to focus on the performance of our specific applications. Instead, in this evaluation we executed our implementation on an Edgecore 100BF-32X programmable switch, based on the Intel Tofino platform. Using two ports running at 40Gbps we connected two virtual machines to the switch, one for sending messages and a second for receiving.

To validate the correctness of our solution, we ran a 24 hour trace from a university campus [20] with our full implementation loaded, and validated all DNS packets received on the other side were undamaged.

5.1 Parameterization

In order to support every DNS query we can gather our parameters from the DNS specification [18]; each label can be at most 63 bytes (*MaxLabelLen*), total length cannot exceed 255 bytes (*MaxLen*) and at most there can be 127 labels (*MaxLabels*).

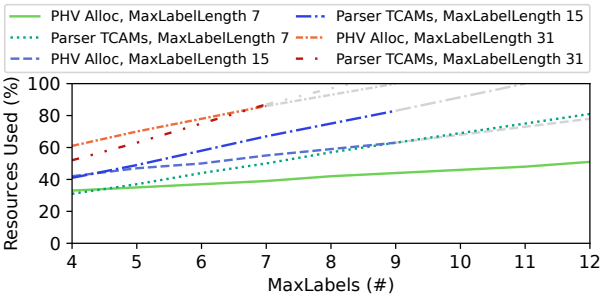

Figure 4: Resource Utilization

Since we cannot achieve such parsing using current hardware, we evaluate real world data to figure out what parameters would serve our purpose optimally. Figure **3** shows the distribution of DNS queries based on a public recording of DNS traffic from a university campus [20]. With these numbers in mind, we set the following parameters:

- *MaxLabelLen* is set to 31, meaning we can only parse labels up to a length of 31. This means each label is split into sections of 1, 2, 4, 8, and 16 bytes. The sections will be used depending on the size, for example a label with 7 bytes will use sections 1, 2 and 4.
- *MaxLabels* is set to 6, meaning queries with more than 6 labels will be cut off after the 6th label.
- *MaxLen* is set to 60. While seemingly counter intuitive our measurements in Figure **3** show the increase in matched queries from raising the limit to 90 is negligible (0.1%).
- *MaxDomainLabels* is set to 4, meaning we can at most refer to a domain made up of 4 labels.

Our chosen parameters should fit the vast majority of DNS related use cases, with more than 97% of DNS traffic being compatible. We note use cases requiring analysis of more labels should be able to reduce *MaxLabelLen* in order to increase *MaxLabels*. [1]

5.2 Resource Utilization

In order to further illustrate the generality of our solution, we attempt to compile our implementation using different variations of the parameters. We focus our attention on two main metrics; PHV allocation and Parser TCAM usage. The resource usage is presented in Figure **4**, highlighting the percentage of each resource spent as a function of our parameters. The gray line in Figure **4** represents parameters we have not managed to compile due to at least one of our resources being exceeded. Our data demonstrates how different parameters can result in different resource utilization, providing a way for our tool to be used for various goals.

[1]Due to current hardware constraints we are unable to parse multiple queries, which would be required to address CNAME records. Instead, our implementation focuses on A-type DNS records (IPv4). It was possible to also include AAAA-type DNS records (IPv6) as they are also constant in size, but for brevity we chose to avoid this less common use case.

Table 2: Additional Resource Utilization

Resource	Minimum Usage	Maximum Usage
Exact Match Xbar	10.1%	62.0%
Hash	18.1%	40.3%
SRAM	7.7%	13.0%
ALU	4.2%	4.2%
TCAM	0.7%	0.7%

Additional resource usage is shown in Table **2**, showing the minimum and maximum values recorded through all compilation parameters. These values show that while our implementation expends some of the switch resources when the parameters are pushed to the limits, other resources are left underutilized; leaving room for additional switch applications.

6 FUTURE WORK AND DISCUSSION

We've highlighted how our solution manages to handle 97% of regular DNS traffic using todays hardware, while being able to scale easily as the hardware improved. With this promising result in mind we would like to draw attention to two current trends in DNS traffic.

Internationalized domain names have been steadily rising in usage since their creation over 10 years ago [17]. While providing a functional purpose their implementation entails translating non-latin script using Punycode; an implementation which is likely to extend the length of the DNS query.

Cloud services usage is on a significant upwards trajectory as more enterprises choose to adopt the technology [12]. In our analysis of DNS traffic [20] we have noticed a significant portion of DNS traffic that exceeds our chosen constraints was related to cloud services.

As these trends continue we expect to see more and more traffic that exceeds the capabilities of existing switches. Even with that in mind, we believe our solution is able to accommodate many DNS related use cases, and we are hoping to see programmable switches improve further in coming years.

In the future we intend to explore security applications related to DNS, such as DNS water torture[14] and subdomain enumeration. We hope our implementation inspires additional researchers to examine DNS in the data plane.

ACKNOWLEDGMENTS

We thank the anonymous SOSR reviewers and our shepherd Rinku Shah for their valuable feedback. This work is supported by the Israel Science Foundation under grant No. 980/21.

REFERENCES

[1] Kamal Alieyan, Mohammed M Kadhum, Mohammed Anbar, Shafiq Ul Rehman, and Naser KA Alajmi. 2016. An overview of DDoS attacks based on DNS. In *2016 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 276–280.

[2] AR Alkhafajee, Abbas M Ali Al-Muqarm, Ali H Alwan, and Zaid Rajih Mohammed. 2021. Security and Performance Analysis of MQTT Protocol with TLS in IoT Networks. In *2021 4th International Iraqi Conference on Engineering Technology and Their Applications (IICETA)*. IEEE, 206–211.

[3] Ali AlSabeh, Elie Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2022. P4DDPI Securing P4-Programmable Data Plane Networks via DNS Deep Packet Inspection. (2022).

[4] Suranjith Ariyapperuma and Chris J Mitchell. 2007. Security vulnerabilities in DNS and DNSSEC. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE, 335–342.

[5] Asier Atutxa, David Franco, Jorge Sasiain, Jasone Astorga, and Eduardo Jacob. 2021. Achieving low latency communications in smart industrial networks with programmable data planes. *Sensors* 21, 15 (2021), 5199.

[6] Sherry Bai, Hyojoon Kim, and Jennifer Rexford. 2022. Passive OS fingerprinting on commodity switches. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE, 264–268.

[7] Kenton Born and David Gustafson. 2010. Detecting dns tunnels using character frequency analysis. *arXiv preprint arXiv:1004.4358* (2010).

[8] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 226–239.

[9] Cloudflare. 2021. Cloudflare blocks an almost 2 Tbps multi-vector DDoS attack. https://blog.cloudflare.com/cloudflare-blocks-an-almost-2-tbps-multi-vector-ddos-attack/

[10] The P4 Language Consortium. 2021. P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html

[11] Mozilla Foundation. 2020. Public Suffix List. https://publicsuffix.org/

[12] Gartner. 2021. Gartner Says Four Trends Are Shaping the Future of Public Cloud. https://www.gartner.com/en/newsroom/press-releases/2021-08-02-gartner-says-four-trends-are-shaping-the-future-of-public-cloud

[13] Intel. 2021. P4 16 Intel Tofino Native Architecture Public Version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf

[14] Alexander Kaplan and Shir Landau Feibish. 2021. DNS water torture detection in the data plane. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 24–26.

[15] Alexander Kaplan and Shir Landau-Feibish. 2022. Practical Handling of DNS in the Data Plane - Code. https://gitlab.com/runs-lab/public/practical-dns

[16] Jason Kim, Hyojoon Kim, and Jennifer Rexford. 2021. Analyzing Traffic by Domain Name in the Data Plane. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research*. 1–12.

[17] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. 2018. A Reexamination of Internationalized Domain Names: The Good, the Bad and the Ugly.. In *DSN*. 654–665.

[18] P. Mockapetris. 1987. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. https://datatracker.ietf.org/doc/html/rfc1035

[19] OASIS. 2019. MQTT Specifications. https://mqtt.org/mqtt-specification/

[20] Manmeet Singh, Maninder Singh, and Sanmeet Kaur. 10. Days DNS Network Traffic from April-May, 2016. *Accessed: Aug* 12 (10), 2021.

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