Server agnostic DNS augmentation using eBPF

Master thesis by Tom Carpay
Security and Network Engineering masters programme

Supervisors: Willem Toorop and Luuk Hendriks
NLnet Labs

August 17, 2020

With ever larger DNS request volumes, handling requests efficiently becomes ever more important. One way to address higher network performance is to bypass the kernel network stack completely, however the software providing the service then has the daunting task to perform all the low-level tasks the kernel would normally handle. The Extended Berkeley Packet Filter (eBPF) and in particular Express Data Path (XDP) kernel hook provides an alternative approach, in which traffic can be considered and acted upon, or passed on to the kernel, from the lowest layer of the network stack.
This research seeks to take this approach one step further and augment DNS software agnostic of the software providing the (basic) DNS service. To this end we explore several aspects of the XDP kernel hook introduced by eBPF. We examine two use cases: QName rewriting and Response Rate Limiting (RRL). We develop a prototype for QName rewriting and develop three RRL prototypes, including an augmentation that is currently unavailable in other DNS software. Each of these prototypes demonstrates one or more different XDP eBPF functionalities to augment DNS. The QName rewriting prototype is used in a continued setup involving the RIPE Atlas network. We perform experiments for each Response Rate Limit prototype by stress testing an authoritative DNS server and introducing the prototypes to the setup. We show that the CPU load drops when using the prototypes over their user space counterparts and we conclude that XDP eBPF is a viable candidate for augmentations to DNS software agnostic of the supplier.

Contents

1 Introduction

As the internet is constantly growing, the amount of DNS requests grows with it. With ever larger request volumes, handling requests more efficiently becomes more and more of a necessity. This increase in volume becomes apparent in the packet processing limitations of network stacks at packet rates higher than 10 Gbps [1].

Currently, there is no Domain Name System (DNS) interaction available low in the Linux kernel network stack. This is desirable for operators of DNS software, for example at high volume authoritative nameservers. A method for handling DNS messages more efficiently is by removing the overhead from the network stack, which could significantly improve performance. A candidate technology this research explores for this is the Berkely Packet Filter (BPF).

The BPF is a Linux based, in-kernel virtual machine that runs a custom 64 bit instruction set that is Just-In-Time (JIT) compiled to machine instructions. BPF allows compiled C BPF bytecode to be attached to a data path via the built in verifier. Once the program passes the verifying process, which is discussed in more detail in Section 3, the program is executed whenever the attached path is traversed.

The extended Berkely Packet Filter (eBPF), the 2014 expansion added to the retroactively called classic Berkely Packet Filter (cBPF), offers a significant number of improvements which enhance its capabilities as packet filter and processor, and shift the focus to a low level tracing tool set. One of the aspects that enables this research is the addition of the eXpress Data Path (XDP). This high performance kernel hook allows BPF programs to be attached to a network interface and execute instructions on a per packet basis. Instructions can be hardware offloaded on selected Network Interface Cards (NICs). Further improvements are discussed in more detail in Section 3. BPF is native to the Linux kernel, so BPF programs are widely available. From Section 2 we learn that BPF programs are more performant to their user space counterparts in many cases.

To explore the capabilities of this technology in relation to DNS, we create a number of prototypes with limited functionalities. These functionalities are chosen based on operator needs and functionalities that are not currently available in DNS software, or are improved by this technology. If successful, these prototypes implement their respective functionalities agnostic of the software supplier, as the logic that is executed is completely invisible to user space. If successful, further developed and be used to augment all current DNS software.

We examine two main use cases: a Query Name (QName) rewriting experimental setup, and Response Rate Limiting, the DNS addition to mitigate DNS amplification attacks. For both use cases we explore prototypes to examine the functionalities of BPF.
The QName rewrite prototype is used in an experimental setup for research being done by Koolhaas and Slokker [2]. For their measurements, a method

is needed to change the DNS query to invoke a predetermined sized response. The authoritative DNS server on the test setup replies different sized responses based on the query name. The probes from which the queries are sent, embed their (variable sized) probe IDs in the query as these are needed to process the measurement results. Besides rewriting the query name to request different sized responses, also the length of the probe ID within the query name needs to be compensated, to guarantee the same (requested) sized responses for all probes. As we show in the Section 4, BPF offers a viable solution to this need, as the measurements can be taken without the need for a DNS parser or modification to a DNS software, facilitating easier reproducibility of the experiment.
The other prototypes revolve around Response Rate Limiting (RRL), a DNS enhancement to mitigate DNS amplification attacks [3]. While RRL is a functionality that is present in a number of the well known DNS software products (BIND[1], NSD[2], KnotDNS[3]) it is not present in others. Software native RRL implementations leave the overhead of the network stack in place for every packet.
The RRL prototypes examine the possibilities for RRL using BPF in three iterations. The first prototype implements a general version of RRL for all incoming DNS messages. The second prototype differentiates the RRL "buckets" for individual IP addresses. The final prototype implements a novel method where a configurable allowlist of "known senders" are exempt from RRL, while all other senders are subject to it per individual source IP address. With each prototype we explore the different functionalities that BPF offers for DNS.

The prototypes for both uses cases are experimentally verified to be functional. In case of the QName rewriting, the XDP program is verified to work on a small scale setup before being used in the large scale experiment by Koolhaas and Slokker.
With the different RRL prototypes, we examine the impact on performance that the XDP programs have to their user space counterpart, where possible. This comparison is drawn by observing the change in CPU load when both cases are being subjected to a stress test. If a comparison is not possible, as with the final RRL prototype iteration, we examine how the prototype scales within the limitation of the resources of this research.

The aim of this research is to augment DNS software agnostic of the software supplier. To this end we pose the main research question: How can XDP eBPF be used to augment and improve DNS software?

To help answer the main research question, we pose the following research sub-questions:

- Which features from XDP BPF could be used to augment DNS software?
- How can DNS augmentations be implemented based upon these XDP

[1] Berkely Internet Name Domain: `https://www.isc.org/bind/`
[2] NLnet Labs' open source Name Server Daemon: `https://www.nlnetlabs.nl/projects/nsd/about/`
[3] Open source authoritative-only name server: `https://www.knot-dns.cz`

eBPF features?

- How do these implementations impact performance?

This paper is structured as follows. In Section 2 we discuss the related work of this research and in Section 3 we discuss the concepts of BPF and XDP in more detail. In Section 4 we examine the research methods, prototypes and algorithms, and in Section 5 the experiments. In Section 6 we examine the results of the experiments and in Section 7 we discuss our findings. Finally, in Section 8 we show our conclusions and in Section 9 we discuss the future work for this research.

2 Related work

The eBPF instruction set itself has been a thoroughly researched topic. In 2017 Vieira et al. [4] gave comprehensive insight into eBPF and its data path hooks. In 2020, Høiland-Jørgenson et al. [5] extended this insight by focusing on the XDP kernel hook. They also compare the packet drop performance of XDP compared to its user space counterparts and show that XDP is more efficient. Their research builds a fundamental understanding of eBPF and the XDP kernel hook, and their conclusion on XDP efficiency build the foundations for this research.

A networking use case for XDP eBPF is for Distributed Denial Of Service (DDOS) protection. This has been studied in scientific research [6] [7] as well as by large organisations such as Cloudflare. The research has shown that an XDP eBPF DDOS program was possible technically and functional [8] and show their product works well [9]. The findings from both the industry and scientific sources show that an XDP protection program offers more efficient DDOS protection than its user space counterparts. Our Response Rate Limiting prototype aims to extend the use case to DNS, to provide protection against the DNS Amplification Reflection type of Denial of Service attacks, independent of the DNS software in use in user space.

3 Background

Originally BPF was designed as a low level packet filter with a customisable set of rules. With the follow up of eBPF in 2014 the functionality of the capabilities of the technology have been expanded to include network performance, firewalls, security, tracing, and device drivers [10].

For clarity, we distinguish classic BPF and extended BPF, which are retroactively named cBPF and BPF respectively, as in the rest of this research we adhere to this naming convention.

The extension on the cBPF improves on several points as shown in Table 1. One of the changes is that the extension of BPF introduces the ability of executing a limited set of kernel functions through "helper functions", which was not allowed in cBPF. An example of a helper function is `bpf_ktime_get_ns`, which calls the kernel function `ktime_get_ns`. Only kernel functions that have a

Property	cBPF	eBPF
Number of registers	2	11
Register size	32 bits	64 bits
Stack size	16 bytes	512 bytes
Instruction execution limit	10.000	1.000.000
Kernel function calls	Not allowed	Allowed
Program offloading to SmartNIC	Not possible	Possible
Dynamic loading and program reloading	Not possible	Possible

Table 1: The changes made extending BPF [4].

helper function counterpart can be used in BPF programs. Specifically, an BPF program cannot call user-space functions, because it is not running in user space.

BPF programs are user-created programs that run in kernel space. Code written in C is compiled to bytecode as a BPF program. Running user-created programs in kernel space could potentially lead to security or stability risks. To mitigate this risk, all BPF programs are inspected by the BPF verifier. If the program passes the verifier, on first execution the bytecode program is JIT compiled to machine instructions to be executed. This life cycle is shown in Figure 1.

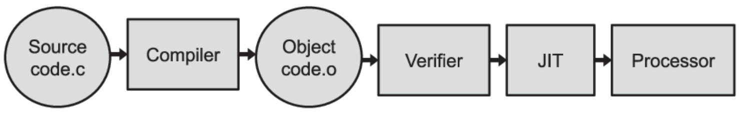

Figure 1: The life cycle of a BPF Program (copied from Vieira et al. [4]).

The BPF verifier performs three main checks [11]. The first check tries to verify that the program does not contain any kernel locking loops and that the program terminates. To this end, the verifier creates a directed acyclic Control Flow Graph (CFG) and does a depth first search of this graph to find the terminating points. If it finds any potentially non terminating path, the verifier will fail the program, as these are not allowed. An example CFG is shown in figure 2.

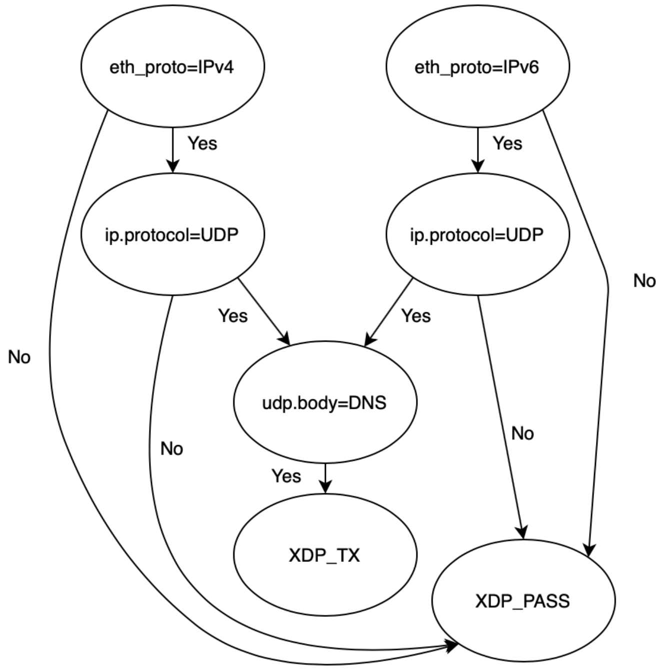

Figure 2: A directed acyclic control flow graph representation for a BPF Program that rejects all incoming DNS queries. This program is follows a different flow based on the received IP version. Note the distinction between the starting points for both IP versions. This distinction is made to ensure that all instructions are reachable in either IP case during verification.

The second verifier check involves simulating the program, executing and checking per single instruction. This simulation verifies that the program stays within bounds and never accesses out of range memory.
For the final verifier check, the verifier restricts kernel functions and data structures available to the program according to the eBPF program type. In this research we use program types corresponding to the XDP kernel hook. The XDP hook is limited to incoming traffic. A more elaborate description of the verifier is described by Vieira et al. [4] and Høiland-Jørgensen et al. [5].

The method that BPF uses to interact with user space is via BPF "maps". These maps are data structures of which the type has to be defined at compile time and are stored in user space. The map types are used in the final verifier check to restrict map usage to its respective type. BPF programs can interact with maps through BPF helper functions such as `bpf_map_lookup_elem()`, `bpf_map_update_elem()`, and `bpf_map_delete_elem()`. Since BPF programs attach to a data path which can be traversed many times per second, to avoid memory locks the map types can be defined as per CPU core or shared between all CPU cores.

A selection of available map types are:

- BPF_ARRAY,

- BPF_PERCPU_ARRAY
- BPF_HASH
- BPF_PERCPU_HASH
- BPF_STACK_TRACE
- BPF_ARRAY_OF_MAPS

The map types used in this research are the `BPF_PERCPU_ARRAY`, `BPF_HASH`, and `BPF_PERCPU_HASH`.
The Array type has a predefined number of elements and is available per CPU or shared between cores. The maximum key size is 32-bit and the size of the array is defined before compiling [12].
The Hash type is a HashMap (or associative array) implementation which is optimised for fast lookup and updates. The Hash type uses the `jhash` library [12], which is a Linux kernel library. The maximum key size is configurable, though the standard is 64-bit.

To summarise, there are three components that enable the agnostic augmentation of DNS software. Firstly, the extension of cBPF allows BPF programs to be used in a wider method. Secondly, the three checks of the BPF verifier ensure that a BPF program is executed safely and that it is finite. Lastly, BPF maps offer a method for BPF programs to interact with user space and need to be configured with a specific type at compile time.

4 Methods

In this section we discuss the approaches for designing the prototypes. Code for all prototypes is found in the Appendix.

As mentioned before, we choose two main use cases to write prototypes for: rewriting query names of DNS messages and response rate limiting large volumes query sources. The selection of prototypes is chosen because each one demonstrates one or more different key functionalities of XDP BPF.

The QName rewrite prototype demonstrates the ability to rewrite packet contents before they are passed on to the network stack and the usability of XDP programs in large scale network tests. The general RRL prototype demonstrates a performance comparison of a native DNS software functionality, and displays the interaction with the map user space storage from within the XDP program. The per IP RRL prototype iterates on the general RRL prototype and attempts to show that increasing the number of source IPs does not influence the workload, which displays that the XDP program scales for more IPs as used as source addresses. The unknown host prototype attempts to demonstrate a functionality that is not available in most, current DNS software and shows interaction and configurability of a running XDP program from user space.
The listed functionalities help to answer the first research sub-question: Which features from XDP BPF could be used to augment DNS software?

4.1 QName rewriting

With the QName rewrite prototype we explore rewriting incoming packets by rewriting the packet before it is passed on to the network stack. This process is completely invisible to programs running in user space. The use case here is DNS message size measurements done with RIPE Atlas[4]. For this use case only the QName and checksum are rewritten, so that the query invokes a different response. The need for rewriting the query comes from the fact that the measurement has been scheduled with predefined properties allowing it to target all resolvers on all current and future probes, in an hourly schedule, and in an ongoing manner without preset end time. However, these predefined properties are bound to a fixed query name. To invoke different response sizes on different moments in time the query name needs to be changed before it is passed on to the network stack. Furthermore, the query structure involves the probe ID, which can be up to 7 digits and changes per probe. Since different probes send queries with different sizes, they invoke different respective responses. To enable all probes to invoke the same response, which is desirable for the use case, the number of digits in the ID is compensated by the XDP program. To guarantee the same sized responses for every probe ID, the query name needs to be further adjusted to reflect the length of the probe ID which is compensated with a tuned response by the NSD daemon serving the request in user space.

To this end, a DNS request is received on the network interface and the XDP program is run by the kernel. The program determines if the packet is either IPv4 or IPv6, and verifies it as a DNS message by looking at the UDP port. When the message is verified to be DNS, the expected size of the query and the size of the probe ID are checked, and a following label is expanded to fill the discrepancy between the expected and the current size. To guarantee the checksum is still correct, it is recalculated and updated. When this succeeds the packet is passed along to the network stack.
Using this method, the DNS label can be modified on the machine running the NSD instance, without the need to modify any zones or the program itself.

4.2 Response Rate Limiting

To explore the RRL topic we present three prototype iterations:

- General RRL
- Per IP RRL
- Unknown host RRL

The first prototype version, general RRL, offers an operator the ability to set a fine-grained limit of received packets that the DNS service can process without dropping packets. This limit can be fine-tuned to the capabilities of the machine and network that this service is running on.

The general version of the RRL prototype functions by counting every DNS message that is received within a predetermined time frame, and stores the

[4]The RIPE Atlas internet measurement network: `https://atlas.ripe.net`

frame starting time and the total counted packets in the frame in an array BPF map with a single entry to be accessible at every program execution. The time frame determines the granularity of the algorithm and is configured before compiling. The starting time in the frame is updated every instance it has reached or surpassed the predetermined time frame size value. This check is performed by subtracting the current time of the packet from the starting time, both in nanoseconds, and examining this against the time frame threshold. If the total number of packets exceeds the threshold within the time frame, which are both configured before compiling, all other packets in the same time frame are either dropped or replied to the sender with the truncated flag set (bounced), depending on the desired behavior.

Per IP RRL
The per IP RRL prototype functions much the same as the general version. The main difference is that the time frame and number of packets are calculated and stored per IP. The values are stored in a hash type BPF map where the key is the source IP address of the incoming query. The size of the map can be configured before compilation.

The per IP RRL prototype provides the option to differentiate between low volume query sources and high volume, repetitive query sources. This differentiation allows the algorithm to restrict high volume senders, while serving the low volumes senders unimpeded.

Unknown host RRL
The unknown host RRL prototype is similar to the per IP RRL prototype in that IPs can be rate limited, but differs in how this is accomplished.

The unknown host RRL program functions by matching the IP of the current DNS message to an entry in the known hosts HashMap, and subjects the message to RRL if the source IP is not in the allowlist. If an entry exists in the known hosts HashMap for an IP, the program can be configured to bounce or pass the message based on the entry. The entry, like in the per IP RRL prototype, stores the number of packets seen from this sender and the start of the current time frame. If the entry does not exist, the algorithm creates an entry in a second HashMap with the number of packets set to 1 and the start time set to the current time. This second HashMap is identical in use to the one in the per IP RRL program.

The method of adding entries to the HashMap is manually adding them from user space. This method allows the user to create a list of trusted senders which are exempt from rate limiting without recompiling and reloading the BPF program.

5 Experiments

In this Section we describe the experimental setups for the two use cases, QName rewriting and RRL, and their respective prototypes.

5.1 QName rewrite

The QName rewrite prototype is used in research by Koolhaas and Slokker [2]. The goal of their experiment is to find the optimal size for User Datagram Protocol (UDP) packets traversing the network without fragmenting using the RIPE ATLAS measurement platform by measuring DNS responses of a known size.

Their experimental setup involves long running RIPE Atlas measurements for which the parameters cannot be changed once the measurements are started. The query name (QName) of DNS requests is changed by an XDP program to invoke requests for different response sizes. Rewriting by the XDP program is done without the need to change the query parameters of the RIPE Atlas measurement, such as individual prove IDs. The setup used by Koolhaas and Slokker can be seen in Figure 3

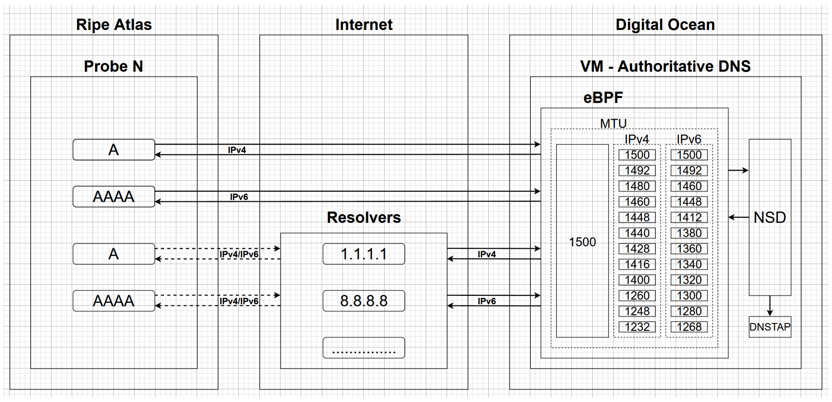

Figure 3: The experimental setup used by Koolhaas and Slokker. Note that the MTU size is determined by the query and is corrected in the eBPF program. (copied from [2]).
An example query from probe N could request a response of 1480 bytes directly from the authoritative server. In this example case, the ID of probe N, which is included in the query, is smaller than the maximum probe length and the response the query receives would be 1472 bytes. To ensure each measurement receives the same response size regardless of the ID length, the eBPF program adds padding to the query to ensure the response size is correct.

The QName prototype is used extensively during measurements of the Koolhaas and Slokker research and shown to function for all used queries which are also shown in Figure 3.

5.2 General RRL

The experimental setup for the RRL experiment consists of two servers. One running an instance of NSD[5], as DNS server, and one running an instance of the Flamethrower[6] tool. Flamethrower is a community-based tool that can be

[5]NSD, an open source authoratative DNS name server: `https://www.nlnetlabs.nl/projects/nsd/about/`

[6]The open source Flamethrower source code: `https://github.com/DNS-OARC/flamethrower`

used as a stress test method for DNS software by sending many queries in rapid succession. The tool has the capability of concurrent requests with configurable Queries per second (QPS). Flamethrower is used in all RRL experiments.

At the time of writing, an instance of NSD comes preconfigured with the RRL feature enabled with a default of 200 QPS per IP address. Enabling the NSD RRL implementation for this experiment offers a method of verifying the RRL prototype. Most packets will be dropped by the NSD RRL when querying in large volumes. Counting the number of dropped packets, as timeouts, allows us to measure the effectiveness of the XDP program as we configure it to return the queries that are response rate limited with the truncated flag set. Using increasingly more aggressive RRL thresholds, i.e. lowering the threshold so that more messages are Response Rate Limited, in the XDP program, we expect the number of dropped packets to decrease. If successful, we can conclude that the program is functional.
Increasing the RRL threshold while measuring the CPU load will also give us insight in the efficiency relation between the two. We expect the CPU load to drop when increasing the aggressiveness of the RRL threshold on the XDP program, by lowering the RRL threshold.
The configuration choice of using the NSD RRL functionality is made, as logging the packet dropped by the XDP program would be a relatively computationally expensive operation. An expensive operation could influence the CPU load measurements, and the current solution allows for easier measuring, as Flamethrower creates a report for every executed run.

5.3 Per IP RRL

The setup for the per IP RRL experiment is similar to the previous experiment. The difference is that the RRL functionality of NSD is switched off and that instead of receiving the same query from one source IP, here we send traffic from multiple source IP addresses. Since the functionality of the program is established by the previous experiment, the goal of this experiment is assessing how traffic originating from multiple senders affects the CPU load of our XDP implementation. The NSD RRL functionality is not needed for this goal. A visual representation of the setup can be seen in Figure 4.

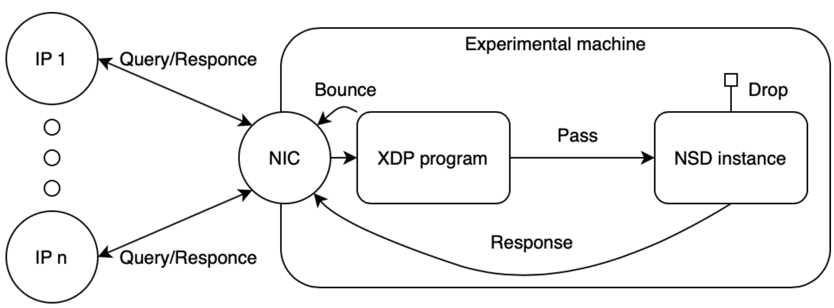

Figure 4: Representation of the setup of the per IP experiment. Note that all senders have their own assigned IP address from which they send queries to the same machine.

To keep the measurements comparable while varying the number of senders, a maximum QPS limit needs to be found. The machine running the flamethrower instances and the machine running the NSD instance are not on the same network. We empirically determine a maximum QPS for our specific measurement setup, ensuring the load per core never exceeds 100 percent usage, as this could influence the results. When found, this limit will be used in the rest of the experiments.

For this experiment we configure the RRL aggressively, to ensure the majority of the responses are sent by the XDP program, as opposed to NSD. Since the total number of responses does not vary significantly, we do not expect the CPU load of the NSD instance to change as it is highly optimised for large number of queries from different sources.
To measure the difference in CPU load from the XDP program, the number of flamethrower instances with their own IPs is increased incrementally. We expect the total CPU load not to change significantly as the number of source IPs increases.

5.4 Unknown host RRL

The experimental setup for the unknown host RRL experiment is the same as for the per IP RRL experiment. It consist of multiple flamethrower instances with their own respective IP addresses, which send queries to a single machine running an NSD instance, such as can be seen in Figure 4.
The difference compared to the previous experiment is that instead of varying the number of flamethrower instances, the number of source IPs that are included in the allowlist is incremented between runs. This incremental change allows us to verify that the XDP program is functional, as we expect to see CPU load increase as more IPs are included in the allowlist as the workload for NSD grows. IPs are added to the map of known hosts with the `bpftool` toolset.

Querying non-existing domains allows us to differentiate the responses received from the XDP program and NSD. The responses that the Flamethrower tools receive contain the DNS response code `REFUSED`, while responses that are above the RRL threshold within the XDP program contain the DNS return code `NOERROR`. This difference in response code allows us to verify that the number of queries that reach the NSD instance is increasing according to the respective run.

6 Results

In this Section we present our results from developing and assessing the prototypes as described in Section 5.

6.1 QName rewrite

With the research of Koolhaas and Slokker, a RIPE Atlas measurement was scheduled with the special properties of reaching all resolvers running on all RIPE Atlas probes, and going on continuously and persistently. These special properties came at the expense of the ability to modify the query. Our query rewrite XDP program resolved this limitation independent of, and without modifying DNS software. Maintaining standard components greatly increases reproducibility of the experiment. Furthermore, we were able to fine tune rewritten queries on a per packet basis to fine tune the variations in response sizes caused by the length of the Probe ID embedded in the query name.

We found that rewriting the query name of incoming DNS packets, and restoring the original query name on response, is possible as DNS wire format imposes hard limits on the length and the number of the labels in a query name, which makes the number of control flow paths and boundaries within the program, that the verifier has to check, finite. Overall, we find that an entire DNS packet can be rewritten as long as the size of the packet does not change. A change of the size of a packet is limited to 256 bytes. These 256 bytes are located at the head of the packet to enable encapsulation and are less suitable for extension of the DNS packet [13].

The functionality of the QName rewrite program is emergent from the results of [2], as they verify that the program is effective. All queries within specifications are handled correctly by the XDP program, including the layer 3 and layer 4 checksums. Queries outside of the specification are not tested, and would not likely result in correct execution. Koolhaas and Slokker have reported no loss in performance using the XDP program. From their report we can conclude that it is functional in large scale networking experiments, and thus realises the goal set for this experiment in Section 4.

6.2 General RRL

As mentioned in Section 5, we perform multiple iterations of the experiment increasing the aggressiveness of the RRL threshold to verify that the general RRL program works correctly. To this end, we choose 8 RRL thresholds which give insight into the performance of the XDP program. This performance is measured by comparing timeouts, i.e. packets that are dropped by NSD, against truncated packets. The range of these thresholds is found by experimentally finding a low threshold number that results in a number of timeouts that is close to zero. From there, the threshold is increased in intermediate steps to a number that is close to the number of packets that are dropped by the NSD resolver itself. To visualise the working of the program, we configure all packets that go above the threshold to be returned with the truncation flag instead of being dropped.

In total, roughly 600.000 DNS queries are sent per run. We find that with this configuration NSD uses roughly 80% of a single CPU core for replies. Since the assigned CPU core changes per run, we measure the total CPU load. The machine running the Flamethrower instance has access to 3 CPU cores.

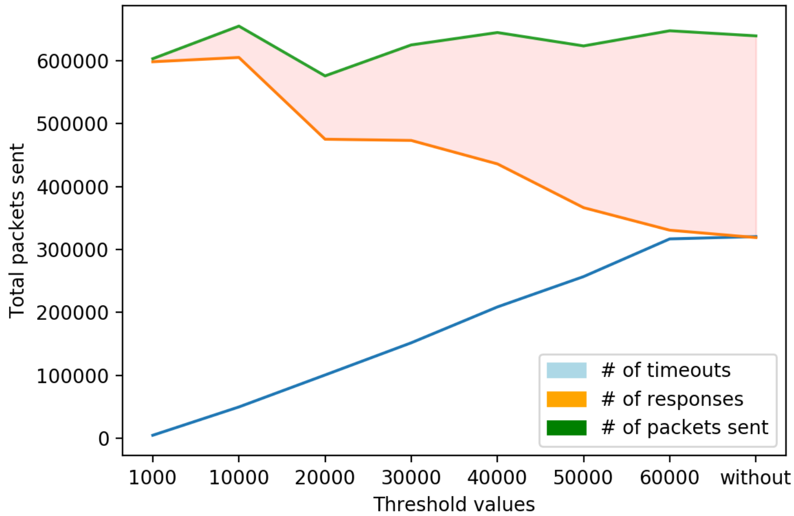

Figure 5: Visualisation of the general RRL experiment. Note that the red shaded area shows all the packets that timeout at the Flamethrower tool as they are dropped by the built-in NSD RRL, and the final measurement is taken without an XDP program attached. We discuss the variability of the number of packets sent in Section 7.

Figure 5 shows incrementally increases in the configured RRL threshold until it roughly corresponds with the NSD RRL threshold. From the figure we notice that increasing the RRL threshold, decreases the number of responses that are received by the Flamethrower tool. This is due to more queries reaching the NSD instance, which starts dropping them accordingly. From this observation we can deduce that packets are processed correctly by the XDP program and conclude that the program is functional.

Simultaneously to the measurements, the total CPU load on the resolver machine was measured as well.

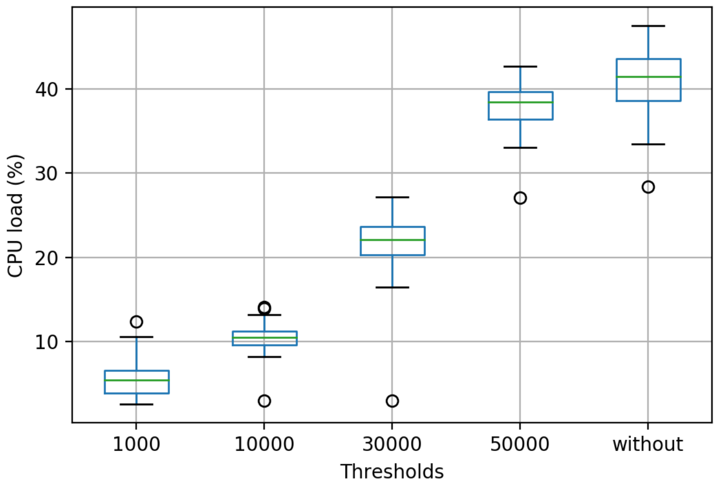

Figure 6: Visualisation of the measured total CPU load per configured RRL threshold. Note that this change is due to a difference in workload for NSD.

In Figure 6 we visualise the CPU load of different RRL thresholds. We can observe that the CPU load drops significantly when lowering the RRL threshold. When we compare the XDP program to the NSD RRL functionality, we can observe that the CPU load is lower in the XDP program while handling the same number of packets. We can observe that this behavior increases as the XDP RRL is configured more aggressively.
From this observation we can conclude that using the XDP program is more CPU efficient than using the NSD RRL functionality. This conclusion is in line with our expectations from Section 5.2.

6.3 Per IP RRL

The maximum QPS described in Section 5.3 to keep the total CPU load around 80% without dropping any packets is found to be roughly 45.000 QPS. All individual Flamethrower processes are assigned one IP address and limit their QPS per source IP so that the total QPS never rises above 45.000 QPS. For example, in the configuration with 2 source IPs, both flamethrower instances are configured to limit at 22.500 QPS. The configured RRL threshold in the XDP program used for all tests is 1 QPS, which is the minimum in the program. This low threshold ensures that we only measure the change in the XDP workload and not in a significantly changing NSD workload. The total CPU load is measured every second for 30 seconds per run.

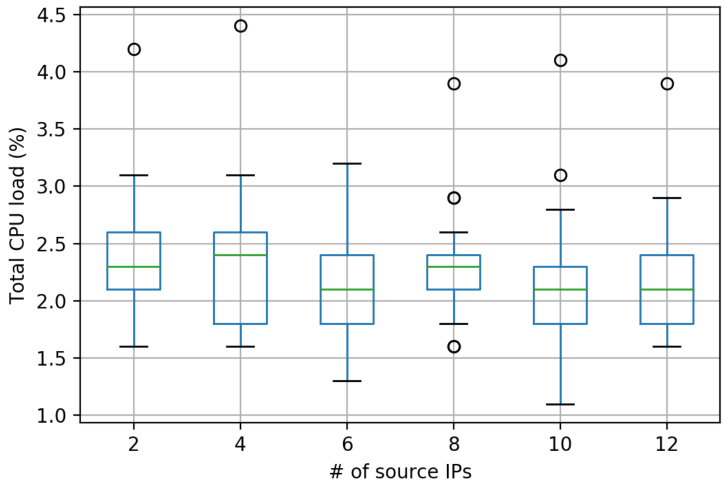

Figure 7: Visualisation of the measured total CPU load per number of source IPs. Note that the QPS is constant over the experiment, and the RRL threshold for the XDP program is set to 1 QPS per source IP.

In Figure 7 we can observe the measured total CPU load for the different number of source IP addresses. We notice the medians of all measurements to be within 0.5% of each other, so we can conclude that our predictions made in Section 5.3 are correct and the total CPU load does not increase significantly as the number of source IPs increases.

6.4 Unknown host RRL

The source IPs are added to the allowlist in increments of 2 IPs per run. This increment gives us a gradual overview of the change in CPU load as more messages reach the NSD instance. Each run involves 12 instances of Flamethrower which are configured to a limit of 3750 QPS. The RRL threshold of the XDP program is configured at 1000 QPS to make a change in CPU load per run visible.

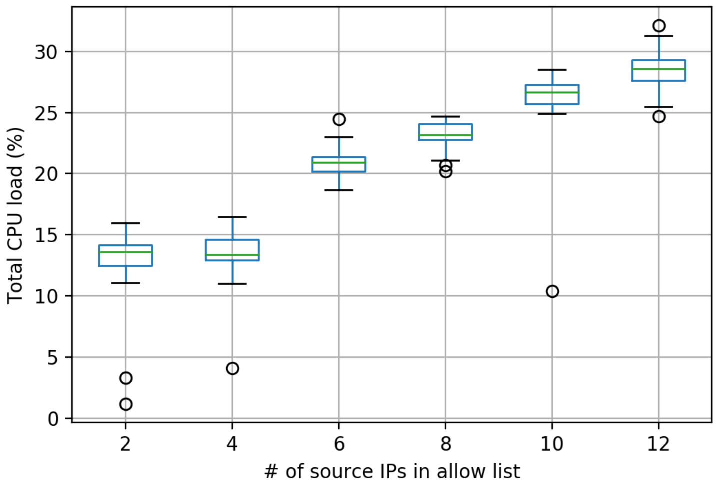

Figure 8: A visualisation of the measured total CPU load per number of source IPs in the allowlist. Note that number of IPs used as source address and the QPS is constant throughout the experiment and only the IPs added to the allowlist changes per run.

From Figure 8 we can observe the visualisation of the CPU load per number of IPs used as source address in the allowlist. The visualisation shows that adding IPs to the allowlist increases the CPU load. Because the number of source IPs and the QPS limit is constant throughout the experiment, we can conclude that the increase in CPU load is due to NSD handling the queries, and the XDP program is functional.

7 Discussion

In the results from Figure 5 we can observe that the number of packets sent is not the same for every data point. While these measurements are taken over 10 second intervals, as mentioned before in Section 5.3 the information and control of the network falls outside of the resources of this research. While the network variability does influence the results, we argue that this is not significant for the conclusions of the results, as the variability is never more than 5 percent.

While this research shows that XDP BPF provides an opportunity to augment DNS services without making changes to the service itself, the augmentation shown is limited to DNS over UDP.
Currently DNS over UDP is the standard and possible agnostic augmentations can therefore be implemented in the foreseeable future. We do note that DNS over TLS or DNS over HTTPS specifications exist and acknowledge that the class of augmentations shown in this work would not be functional, as the decryption of packets happens in higher network layers which negates the usability on lower layers.

In all RRL experiments, every packet is checked if it exceeds the configured RRL threshold in the time frame, and if it arrived within the time frame. While this method of checking is functional, it could be argued that this is not efficient as BPF helper calls, such as the helper call for the current time, are relatively computationally expensive as they make a system call. To optimise this, the time check could be configured based on a predetermined number of packets or a per-packet probability. This change can be useful in high traffic volume environments, as the expensive system call and the check could be superfluous for every packet, as large volumes of packets are received every time frame.
While these checking methods are likely an efficiency improvement over checking every packet, such performance improvements are not within the scope of exploring this research

In the general RRL experiments the XDP program is configured to bounce the received DNS queries with the TC bit set if the RRL threshold is exceeded. This behavior is not strictly RRL, as this requires the packet to be dropped instead of bounced. The XDP program contrasts the behavior of the NSD instance, Which by default drops 50% of the requests when they exceed the configured NSD RRL threshold. While bouncing packets is computationally more expensive as it requires more steps than dropping the packet, the conclusion drawn in Section 6.2 is still valid, as both actions happen at the kernel level before the network stack and the results show the XDP program to be more efficient than the NSD functionality, even while bouncing the packet instead of dropping it.

A possible criticism on the proposed method of augmenting DNS software is that it results in layer violation of the application layer in the OSI model. While we concede this points as application layer data is resolved on the wrong layer, we can also argue that the violation is contained to a single machine running the program and is thus constrained to the machine and not the wider network.

8 Conclusion

In this research, we have proposed a novel method of augmenting DNS services agnostic of the DNS software supplier. This method is enabled by the BPF instruction set and the included XDP kernel hook.
To showcase the different aspects of this technology we have created 4 prototypes. With the QName rewrite prototype we have shown the ability to rewrite packet contents before they are passed on to the network stack. With the general RRL experiments we have drawn a performance comparison between a DNS software native functionality and the general RRL prototype, and has displayed interaction with the map user space storage from within the XDP program. The per IP RRL prototype iterates on the general RRL prototype and shows that increasing the number of source IPs does not influence the workload. The unknown host prototype has demonstrated a functionality that is not available in most current DNS software and has shown configurability of a running XDP program from user space.

We have answered the first research sub-question in Section 3: Which features from XDP eBPF could be used to augment DNS software? We answer this by examining the properties of BPF and XDP.
The implementations described in Section 4 answer the second research sub-question: How can DNS augmentations be implemented based upon these XDP eBPF features? We answer this with the created prototypes.
In Section 6 we have concluded that all experiments show the XDP BPF prototypes lowers the CPU load compared to the user space counterpart, and have therefore answered the third research sub-question: How do these implementations impact performance?
With this research we have shown that XDP BPF is a viable candidate for augmentations to DNS software agnostic of the supplier and answered the main research question: How can XDP BPF be used to augment and improve DNS software?

9 Future work

To further this research, experiments around the per IP RRL prototype could be done on high traffic simulations or high volume name servers. This could provide a more real-world scenario where DNS defense against amplification attacks is required, while legitimate traffic could still be resolved.

Another potential candidate for augmenting a DNS service with BPF, is server-side handling of DNS Cookies. DNS Cookies are an in-protocol security mechanism against off path denial-of-service and amplification, forgery, or cache poisoning attacks. This protection comes from the DNS server recognising a returning client by verifying a cookie carried in the request, generated from a client-provided nonce and a secret known only by the server.
The DNS Cookie interactions are embedded in EDNS0 options and piggy back on, and are completely independent from, regular DNS interactions between a client and a server, which makes them especially suitable for processing by BPF. Furthermore, earlier DNS software implementations were not interoperable making them impractical to deploy on multi-vendor anycast networks. An BPF implementation would resolve this issue by providing a single, consistent and performant implementation of DNS Cookie processing regardless of the DNS software used to serve the DNS requests.

Another interesting topic for future research is to examine the possible security implications of the prototypes presented in this research. While they are bound to the limitations and the verifier of BPF, they could still pose a possible security risk as unwanted instruction executions in kernel space could pose a security risk.

References

[1] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle, “Demystifying the Performance of XDP BPF,” in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 208–212.

[2] A. Koolhaas and T. Slokker, "Defragmenting DNS; Determining the optimal maximum UDP response size for DNS," 2020.

[3] S. Goldlust, "A Quick Introduction to Response Rate Limiting," https://kb.isc.org/docs/aa-01000 (Accessed: 15 July 2020), 2018.

[4] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[5] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 54–66.

[6] H. Wieren, "Signature-Based DDoS Attack Mitigation: Automated Generating Rules for Extended Berkeley Packet Filter and Express Data Path," Master's thesis, University of Twente, 2019.

[7] N. de Bruijn, "eBPF Based Networking," 2017.

[8] G. Bertin, "XDP in practice: integrating XDP into our DDoS mitigation pipeline," in *Technical Conference on Linux Networking, Netdev*, vol. 2, 2017.

[9] A. Fabre, "L4drop: XDP DDOS mitigations," https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/ (Accessed: Jun 2020), 2019.

[10] B. Gregg, "Learn ebpf tracing: Tutorial and examples," http://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html (Accessed: Aug 2020), 1 Jan 2019.

[11] M. Fleming, "A thorough introduction to eBPF," *Linux Weekly News*, 2017.

[12] C. Neira, "bcc reference guide," https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#1-bpf_table (Accessed: Aug 2020), 2020.

[13] Cilium, "BPF and XDP Reference Guide," https://docs.cilium.io/en/v1.8/bpf/#xdp (Accessed: Aug 2020), 2020.

10 Acknowledgements

I would like to thank my supervisors Willem Toorop and Luuk Hendriks for giving me the opportunity to work on NLnet Labs Research on Networks project, the great cooperation, and their invaluable contributions to this research.

I would also like to thank my family and friends for the (continued) support before, during, and after this project.

A QName rewrite prototype code

```
/*
 *  rrl-per-ip
 *  Implements per IP RLL within a time frame for hosts that are not known in hte
 *  Jun 2020 - Tom Carpay
 */

#include <stdint.h>
#include <linux/bpf.h>
#include <linux/if_ether.h> /* for struct ethhdr   */
#include <linux/ip.h>       /* for struct iphdr    */
#include <linux/ipv6.h>     /* for struct ipv6hdr  */
#include <linux/in.h>       /* for IPPROTO_UDP     */
#include <linux/udp.h>      /* for struct udphdr   */
#include <linux/pkt_cls.h>
#include <bpf_helpers.h>

#define DNS_PORT 53
#define MAX_LABELS 50

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
# define __inline                         \
   inline __attribute__((always_inline))
#endif

#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
# ifndef  ntohs
#  define ntohs(x) __builtin_bswap16(x)
# endif
# ifndef  htons
#  define htons(x) __builtin_bswap16(x)
# endif
# ifndef  ntohl
#  define ntohl(x) __builtin_bswap32(x)
# endif
# ifndef  htonl
#  define htonl(x) __builtin_bswap32(x)
# endif
#else
# ifndef  ntohs
#  define ntohs(x) (x)
# endif
# ifndef  htons
#  define htons(x) (x)
```

```
# endif
# ifndef  ntohl
#  define ntohl(x) (x)
# endif
# ifndef  htonl
#  define htonl(x) (x)
# endif
#endif

#ifndef memset
# define memset(dest, chr, n)   __builtin_memset((dest), (chr), (n))
#endif

#ifndef memcpy
# define memcpy(dest, src, n)   __builtin_memcpy((dest), (src), (n))
#endif

#ifndef memmove
# define memmove(dest, src, n)  __builtin_memmove((dest), (src), (n))
#endif


struct vlanhdr {
        uint16_t tci;
        uint16_t encap_proto;
};

struct dnshdr {
        uint16_t id;

        uint8_t  rd     : 1;
        uint8_t  tc     : 1;
        uint8_t  aa     : 1;
        uint8_t  opcode : 4;
        uint8_t  qr     : 1;

        uint8_t  rcode  : 4;
        uint8_t  cd     : 1;
        uint8_t  ad     : 1;
        uint8_t  z      : 1;
        uint8_t  ra     : 1;

        uint16_t qdcount;
        uint16_t ancount;
        uint16_t nscount;
        uint16_t arcount;
};

struct cursor {
        void *pos;
```

```
        void *end;
};

static __inline
void cursor_init(struct cursor *c, struct xdp_md *ctx)
{
        c->end = (void *)(long)ctx->data_end;
        c->pos = (void *)(long)ctx->data;
}

static __inline
void cursor_init_skb(struct cursor *c, struct __sk_buff *skb)
{
        c->end = (void *)(long)skb->data_end;
        c->pos = (void *)(long)skb->data;
}

#define PARSE_FUNC_DECLARATION(STRUCT)                          \
static __inline                                                 \
struct STRUCT *parse_ ## STRUCT (struct cursor *c)          \
{                                                             \
        struct STRUCT *ret = c->pos;                            \
        if (c->pos + sizeof(struct STRUCT) > c->end)         \
                return 0;                                     \
        c->pos += sizeof(struct STRUCT);                    \
        return ret;                                             \
}

PARSE_FUNC_DECLARATION(ethhdr)
PARSE_FUNC_DECLARATION(vlanhdr)
PARSE_FUNC_DECLARATION(iphdr)
PARSE_FUNC_DECLARATION(ipv6hdr)
PARSE_FUNC_DECLARATION(udphdr)
PARSE_FUNC_DECLARATION(dnshdr)

static __inline
struct ethhdr *parse_eth(struct cursor *c, uint16_t *eth_proto)
{
        struct ethhdr  *eth;

        if (!(eth = parse_ethhdr(c)))
                return 0;

        *eth_proto = eth->h_proto;
        if (*eth_proto == htons(ETH_P_8021Q)
        ||  *eth_proto == htons(ETH_P_8021AD)) {
                struct vlanhdr *vlan;

                if (!(vlan = parse_vlanhdr(c)))
                        return 0;
```

```

                *eth_proto = vlan->encap_proto;
                if (*eth_proto == htons(ETH_P_8021Q)
                ||  *eth_proto == htons(ETH_P_8021AD)) {
                        if (!(vlan = parse_vlanhdr(c)))
                                return 0;

                        *eth_proto = vlan->encap_proto;
                }
        }
        return eth;
}

static __inline
void update_checksum(uint16_t *csum, uint16_t old_val, uint16_t new_val)
{
        uint32_t new_csum_value;
        uint32_t new_csum_comp;
        uint32_t undo;

        undo = ~((uint32_t)*csum) + ~((uint32_t)old_val);
        new_csum_value = undo + (undo < ~((uint32_t)old_val)) + (uint32_t)new_val;
        new_csum_comp = new_csum_value + (new_csum_value < ((uint32_t)new_val));
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        *csum = (uint16_t)~new_csum_comp;
}

static __inline
void rewrite_qname4(struct cursor *c, uint8_t *pkt, struct udphdr *udp)
{
        uint8_t *labels[MAX_LABELS];
        uint8_t  i;

        for (i = 0; i < MAX_LABELS; i++) { /* Maximum 128 labels */
                uint8_t o;

                if (c->pos + 1 > c->end)
                        return;

                       o = *(uint8_t *)c->pos;
                if ((o & 0xC0) == 0xC0) {
                        return;

                } else if (o & 0xC0)
                        /* Unknown label type */
                        return;

                labels[i] = c->pos;
                c->pos += o + 1;
```

```
                if (!o)
                        break;
        }
        if (i >= MAX_LABELS || i < 5
        || *labels[i-4] != 10
        ||  labels[i-4] + *labels[i-4] + 2 > (uint8_t *)c->end
        ||  labels[i-4][ 1] <  '0' || labels[i-4][1] >  '9'
        ||  labels[i-4][ 2] <  '0' || labels[i-4][2] >  '9'
        ||  labels[i-4][ 3] <  '0' || labels[i-4][3] >  '9'
        ||  labels[i-4][ 4] <  '0' || labels[i-4][4] >  '9'
        ||  labels[i-4][ 5] != '-'
        || (labels[i-4][ 6] != 'p' && labels[i-4][6] != 'P')
        || (labels[i-4][ 7] != 'l' && labels[i-4][7] != 'L')
        || (labels[i-4][ 8] != 'u' && labels[i-4][8] != 'U')
        || (labels[i-4][ 9] != 's' && labels[i-4][9] != 'S')
        ||  labels[i-4][10] != '0' )
                return;

        /* Change aligned on 16 bits for checksum recalculaction */
        uint16_t *pls_pos = (labels[i-4] + 10 - (uint8_t *)udp) % 2
                          ? (uint16_t *)&labels[i-4][9]
                          : (uint16_t *)&labels[i-4][10];
        uint16_t  old_pls = *pls_pos;

        switch (*labels[i-5]) {
        case 39: labels[i-4][10] = '2'; break;
        case 38: labels[i-4][10] = '4'; break;
        case 37: labels[i-4][10] = '6'; break;
        case 36: labels[i-4][10] = '8'; break;
        case 35: labels[i-4][10] = 'a'; break;
        case 34: labels[i-4][10] = 'c'; break;
        default: break;
        }
        update_checksum(&udp->check, old_pls, *pls_pos);
#if MTU4 != 1500
        if (labels[i-4][1] != '1' || labels[i-4][2] != '5'
        ||  labels[i-4][3] != '0' || labels[i-4][4] != '0')
                return;

        if ((labels[i-4] - (uint8_t *)udp) % 2) {
                uint16_t *sh1_pos = (uint16_t*)&labels[i-4][1];
                uint16_t  old_sh1 = *sh1_pos;
                uint16_t *sh2_pos = (uint16_t*)&labels[i-4][3];
                uint16_t  old_sh2 = *sh2_pos;

                labels[i-4][1] =  MTU4_STR[0];
                labels[i-4][2] =  MTU4_STR[1];
                labels[i-4][3] =  MTU4_STR[2];
                labels[i-4][4] =  MTU4_STR[3];

```

```
                update_checksum(&udp->check, old_sh1,*sh1_pos);
                update_checksum(&udp->check, old_sh2,*sh2_pos);
        } else {
                uint16_t *sh1_pos = (uint16_t*)&labels[i-4][0];
                uint16_t  old_sh1 = *sh1_pos;
                uint16_t *sh2_pos = (uint16_t*)&labels[i-4][2];
                uint16_t  old_sh2 = *sh2_pos;
                uint16_t *sh3_pos = (uint16_t*)&labels[i-4][4];
                uint16_t  old_sh3 = *sh3_pos;

                labels[i-4][1] =  MTU4_STR[0];
                labels[i-4][2] =  MTU4_STR[1];
                labels[i-4][3] =  MTU4_STR[2];
                labels[i-4][4] =  MTU4_STR[3];

                update_checksum(&udp->check, old_sh1,*sh1_pos);
                update_checksum(&udp->check, old_sh2,*sh2_pos);
                update_checksum(&udp->check, old_sh3,*sh3_pos);
        }
#endif
}

static __inline
void restore_qname4(struct cursor *c, uint8_t *pkt, struct udphdr *udp)
{
        uint8_t *labels[MAX_LABELS];
        uint8_t  i;

        for (i = 0; i < MAX_LABELS; i++) { /* Maximum 128 labels */
                uint8_t o;

                if (c->pos + 1 > c->end)
                        return;

                       o = *(uint8_t *)c->pos;
                if ((o & 0xC0) == 0xC0) {
                        return;

                } else if (o & 0xC0)
                        /* Unknown label type */
                        return;

                labels[i] = c->pos;
                c->pos += o + 1;
                if (!o)
                        break;
        }
        if (i >= MAX_LABELS || i < 5
        || *labels[i-4] != 10
        ||  labels[i-4] + *labels[i-4] + 2 >  (uint8_t *)c->end
```

```
        ||  labels[i-4][ 1] <  '0' || labels[i-4][1] >  '9'
        ||  labels[i-4][ 2] <  '0' || labels[i-4][2] >  '9'
        ||  labels[i-4][ 3] <  '0' || labels[i-4][3] >  '9'
        ||  labels[i-4][ 4] <  '0' || labels[i-4][4] >  '9'
        ||  labels[i-4][ 5] != '-'
        || (labels[i-4][ 6] != 'p' && labels[i-4][6] != 'P')
        || (labels[i-4][ 7] != 'l' && labels[i-4][7] != 'L')
        || (labels[i-4][ 8] != 'u' && labels[i-4][8] != 'U')
        || (labels[i-4][ 9] != 's' && labels[i-4][9] != 'S'))
                return;

    if (labels[i-4][10] != '0') {
        /* Change aligned on 16 bits for checksum recalculaction
         * Doesn't work on TC/TX!  Maybe we should use bpf_l4_csum_replace()
         * and bpf_csum_diff().
         *
         * uint16_t *pls_pos = (labels[i-4] + 10 - (uint8_t *)udp) % 2
         *                    ? (uint16_t *)&labels[i-4][9]
         *                    : (uint16_t *)&labels[i-4][10];
         * uint16_t  old_pls = *pls_pos;
         */
        labels[i-4][10] = '0';
        // udp->check = 0;
    }

#if MTU4 != 1500
        if (labels[i-4][1] != MTU4_STR[0] || labels[i-4][2] != MTU4_STR[1]
        ||  labels[i-4][3] != MTU4_STR[2] || labels[i-4][4] != MTU4_STR[3])
                return;

        if ((labels[i-4] - (uint8_t *)udp) % 2) {
                /* TODO: 4 bytes checksum recalculating labes[i-4][1-4]
                 *       with bpf_l4_csum_replace() and bpf_csum_diff()
                 */
                labels[i-4][1] = '1';
                labels[i-4][2] = '5';
                labels[i-4][3] = '0';
                labels[i-4][4] = '0';
        // udp->check = 0;
        } else {
                /* TODO: 6 bytes checksum recalculating labes[i-4][0-5]
                 *       with bpf_l4_csum_replace() and bpf_csum_diff()
                 */
                labels[i-4][1] = '1';
                labels[i-4][2] = '5';
                labels[i-4][3] = '0';
                labels[i-4][4] = '0';
        // udp->check = 0;
        }
#endif
```

```
        return;
}

static __inline
void rewrite_qname6(struct cursor *c, uint8_t *pkt, struct udphdr *udp)
{
        uint8_t *labels[MAX_LABELS];
        uint8_t  i;

        for (i = 0; i < MAX_LABELS; i++) { /* Maximum 128 labels */
                uint8_t o;

                if (c->pos + 1 > c->end)
                        return;

                       o = *(uint8_t *)c->pos;
                if ((o & 0xC0) == 0xC0) {
                        return;

                } else if (o & 0xC0)
                        /* Unknown label type */
                        return;

                labels[i] = c->pos;
                c->pos += o + 1;
                if (!o)
                        break;
        }
        if (i >= MAX_LABELS || i < 5
        || *labels[i-4] != 10
        ||  labels[i-4] + *labels[i-4] + 2 > (uint8_t *)c->end
        ||  labels[i-4][ 1] <  '0' || labels[i-4][1] >  '9'
        ||  labels[i-4][ 2] <  '0' || labels[i-4][2] >  '9'
        ||  labels[i-4][ 3] <  '0' || labels[i-4][3] >  '9'
        ||  labels[i-4][ 4] <  '0' || labels[i-4][4] >  '9'
        ||  labels[i-4][ 5] != '-'
        || (labels[i-4][ 6] != 'p' && labels[i-4][6] != 'P')
        || (labels[i-4][ 7] != 'l' && labels[i-4][7] != 'L')
        || (labels[i-4][ 8] != 'u' && labels[i-4][8] != 'U')
        || (labels[i-4][ 9] != 's' && labels[i-4][9] != 'S')
        ||  labels[i-4][10] != '0' )
                return;

        /* Change aligned on 16 bits for checksum recalculaction */
        uint16_t *pls_pos = (labels[i-4] + 10 - (uint8_t *)udp) % 2
                          ? (uint16_t *)&labels[i-4][9]
                          : (uint16_t *)&labels[i-4][10];
        uint16_t  old_pls = *pls_pos;

        switch (*labels[i-5]) {
```

```
        case 39: labels[i-4][10] = '2'; break;
        case 38: labels[i-4][10] = '4'; break;
        case 37: labels[i-4][10] = '6'; break;
        case 36: labels[i-4][10] = '8'; break;
        case 35: labels[i-4][10] = 'a'; break;
        case 34: labels[i-4][10] = 'c'; break;
        default: break;
        }
        update_checksum(&udp->check, old_pls, *pls_pos);
#if MTU6 != 1500
        if (labels[i-4][1] != '1' || labels[i-4][2] != '5'
        ||  labels[i-4][3] != '0' || labels[i-4][4] != '0')
                return;

        if ((labels[i-4] - (uint8_t *)udp) % 2) {
                uint16_t *sh1_pos = (uint16_t*)&labels[i-4][1];
                uint16_t  old_sh1 = *sh1_pos;
                uint16_t *sh2_pos = (uint16_t*)&labels[i-4][3];
                uint16_t  old_sh2 = *sh2_pos;

                labels[i-4][1] =  MTU6_STR[0];
                labels[i-4][2] =  MTU6_STR[1];
                labels[i-4][3] =  MTU6_STR[2];
                labels[i-4][4] =  MTU6_STR[3];

                update_checksum(&udp->check, old_sh1,*sh1_pos);
                update_checksum(&udp->check, old_sh2,*sh2_pos);
        } else {
                uint16_t *sh1_pos = (uint16_t*)&labels[i-4][0];
                uint16_t  old_sh1 = *sh1_pos;
                uint16_t *sh2_pos = (uint16_t*)&labels[i-4][2];
                uint16_t  old_sh2 = *sh2_pos;
                uint16_t *sh3_pos = (uint16_t*)&labels[i-4][4];
                uint16_t  old_sh3 = *sh3_pos;

                labels[i-4][1] =  MTU6_STR[0];
                labels[i-4][2] =  MTU6_STR[1];
                labels[i-4][3] =  MTU6_STR[2];
                labels[i-4][4] =  MTU6_STR[3];

                update_checksum(&udp->check, old_sh1,*sh1_pos);
                update_checksum(&udp->check, old_sh2,*sh2_pos);
                update_checksum(&udp->check, old_sh3,*sh3_pos);
        }
#endif
}

static __inline
uint16_t restore_qname6(struct cursor *c, uint8_t *pkt, struct udphdr *udp)
{
```

```
        uint8_t *labels[MAX_LABELS];
        uint8_t  i;

        for (i = 0; i < MAX_LABELS; i++) { /* Maximum 128 labels */
                uint8_t o;

                if (c->pos + 1 > c->end)
                        return 0;

                       o = *(uint8_t *)c->pos;
                if ((o & 0xC0) == 0xC0) {
                        return 0;

                } else if (o & 0xC0)
                        /* Unknown label type */
                        return 0;

                labels[i] = c->pos;
                c->pos += o + 1;
                if (!o)
                        break;
        }
        if (i >= MAX_LABELS || i < 5
        || *labels[i-4] != 10
        ||  labels[i-4] + *labels[i-4] + 2 >  (uint8_t *)c->end
        ||  labels[i-4][ 1] <  '0' || labels[i-4][1] >  '9'
        ||  labels[i-4][ 2] <  '0' || labels[i-4][2] >  '9'
        ||  labels[i-4][ 3] <  '0' || labels[i-4][3] >  '9'
        ||  labels[i-4][ 4] <  '0' || labels[i-4][4] >  '9'
        ||  labels[i-4][ 5] != '-'
        || (labels[i-4][ 6] != 'p' && labels[i-4][6] != 'P')
        || (labels[i-4][ 7] != 'l' && labels[i-4][7] != 'L')
        || (labels[i-4][ 8] != 'u' && labels[i-4][8] != 'U')
        || (labels[i-4][ 9] != 's' && labels[i-4][9] != 'S'))
                return 0;

    if (labels[i-4][10] != '0') {
        if ((labels[i-4] - (uint8_t *)udp) % 2) {
            uint16_t old = ((uint16_t *)&labels[i-4][1])[4];
            labels[i-4][10] = '0';
#if MTU6 > 1280
            update_checksum(&udp->check, old, ((uint16_t *)&labels[i-4][1])[4]);
#endif
        } else {
            uint16_t old = ((uint16_t *)labels[i-4])[5];
            labels[i-4][10] = '0';
#if MTU6 > 1280
            update_checksum(&udp->check, old, ((uint16_t *)labels[i-4])[5]);
#endif
        }
```

```
    }

#if MTU6 != 1500
        if (labels[i-4][1] != MTU6_STR[0] || labels[i-4][2] != MTU6_STR[1]
        ||  labels[i-4][3] != MTU6_STR[2] || labels[i-4][4] != MTU6_STR[3])
                return 0;

        if ((labels[i-4] - (uint8_t *)udp) % 2) {
        uint16_t old0 = ((uint16_t *)&labels[i-4][1])[0];
        uint16_t old1 = ((uint16_t *)&labels[i-4][1])[1];
                labels[i-4][1] = '1';
                labels[i-4][2] = '5';
                labels[i-4][3] = '0';
                labels[i-4][4] = '0';
#if MTU6 > 1280
                update_checksum(&udp->check, old0, ((uint16_t *)&labels[i-4][1])[0]);
                update_checksum(&udp->check, old1, ((uint16_t *)&labels[i-4][1])[1]);
#endif
        } else {
        uint16_t old0 = ((uint16_t *)labels[i-4])[0];
        uint16_t old1 = ((uint16_t *)labels[i-4])[1];
        uint16_t old2 = ((uint16_t *)labels[i-4])[2];
                labels[i-4][1] = '1';
                labels[i-4][2] = '5';
                labels[i-4][3] = '0';
                labels[i-4][4] = '0';
#if MTU6 > 1280
                update_checksum(&udp->check, old0, ((uint16_t *)labels[i-4])[0]);
                update_checksum(&udp->check, old1, ((uint16_t *)labels[i-4])[1]);
                update_checksum(&udp->check, old2, ((uint16_t *)labels[i-4])[2]);
#endif
        }
#endif
        return (labels[i-4] - (uint8_t *)pkt) + 1;
}


__section("xdp-rewrite-qname")
int xdp_rewrite_qname(struct xdp_md *ctx)
{
        struct cursor   c;
        uint16_t        eth_proto;
        struct iphdr   *ipv4;
        struct ipv6hdr *ipv6;
        struct udphdr  *udp;
        struct dnshdr  *dns;

        cursor_init(&c, ctx);
        if (!parse_eth(&c, &eth_proto))
                return XDP_PASS;
```

```c

        if (eth_proto == htons(ETH_P_IP)) {
        if (!(ipv4 = parse_iphdr(&c)) || ipv4->protocol != IPPROTO_UDP
        ||  !(udp = parse_udphdr(&c)) || udp->dest != htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(&c)))
            return XDP_PASS;

        rewrite_qname4(&c, (void *)dns, udp);

    } else if (eth_proto == htons(ETH_P_IPV6)) {
        if (!(ipv6 = parse_ipv6hdr(&c)) || ipv6->nexthdr != IPPROTO_UDP
        ||  !(udp = parse_udphdr(&c))   ||  udp->dest != htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(&c)))
            return XDP_PASS;

        rewrite_qname6(&c, (void *)dns, udp);
    }
        return XDP_PASS;
}

__section("tc-restore-qname")
int tc_restore_qname(struct __sk_buff *skb)
{
        struct cursor   c;
        uint16_t        eth_proto;
        struct iphdr   *ipv4;
        struct ipv6hdr *ipv6;
        struct udphdr  *udp;
        struct dnshdr  *dns;

        cursor_init_skb(&c, skb);
        if (!parse_eth(&c, &eth_proto))
                return TC_ACT_OK;

        if (eth_proto == htons(ETH_P_IP)) {
        if (!(ipv4 = parse_iphdr(&c)) || ipv4->protocol != IPPROTO_UDP
        ||  !(udp = parse_udphdr(&c)) || udp->source != htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(&c)))
            return TC_ACT_OK;

                uint16_t old_val = ipv4->frag_off;
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
                ipv4->frag_off |= 0x0040;
#else
                ipv4->frag_off |= 0x4000;
#endif
                update_checksum(&ipv4->check, old_val, ipv4->frag_off);
        restore_qname4(&c, (void *)dns, udp);

    } else if (eth_proto == htons(ETH_P_IPV6)) {
```

```
        if (!(ipv6 = parse_ipv6hdr(&c)) || ipv6->nexthdr  != IPPROTO_UDP
        ||  !(udp = parse_udphdr(&c))   || udp->source != htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(&c)))
            return TC_ACT_OK;

        restore_qname6(&c, (void *)dns, udp);
    }
        return TC_ACT_OK;
}

char __license[] __section("license") = "GPL";
```

B General RLL prototype code

```
/*
 *  General_RRL.c
 *  Implements a semi fine grained udp_dns_reply RRL within a time frame
 *  Jun 2020 - Tom Carpay
 */

/*
 *  Includes
 */
#include <stdint.h>
#include <linux/bpf.h>
#include <bpf_helpers.h>    /* for bpf_get_prandom_u32() */
#include <bpf_endian.h>     /* for __bpf_htons()    */
#include <linux/if_ether.h> /* for struct ethhdr    */
#include <linux/ip.h>       /* for struct iphdr     */
#include <linux/ipv6.h>     /* for struct ipv6hdr   */
#include <linux/in.h>       /* for IPPROTO_UDP      */
#include <linux/udp.h>      /* for struct udphdr    */
#include <string.h>         /* for memcpy()         */

/*
 *  Begin defines
 */
#define DNS_PORT     53

#define FRAME_SIZE          1000000000
#define THRESHOLD           50000
/*
 *  End defines
 */

/*
**  Store the time frame
 */
struct bucket {
        uint64_t start_time;
        uint64_t n_packets;
        // uint64_t qps;
};

struct bpf_map_def SEC("maps") state_map = {
        .type = BPF_MAP_TYPE_PERCPU_ARRAY,
        .key_size = sizeof(uint32_t),
        .value_size = sizeof(struct bucket),
        .max_entries = 1
};

/*
```

```
 *  Store the VLAN header
 */
struct vlanhdr {
        uint16_t tci;
        uint16_t encap_proto;
};

/*
 *  Store the DNS header
 */
struct dnshdr {
        uint16_t id;
        union {
                struct {
                        uint8_t  rd     : 1;
                        uint8_t  tc     : 1;
                        uint8_t  aa     : 1;
                        uint8_t  opcode : 4;
                        uint8_t  qr     : 1;

                        uint8_t  rcode  : 4;
                        uint8_t  cd     : 1;
                        uint8_t  ad     : 1;
                        uint8_t  z      : 1;
                        uint8_t  ra     : 1;
                }        as_bits_and_pieces;
                uint16_t as_value;
        } flags;
        uint16_t qdcount;
        uint16_t ancount;
        uint16_t nscount;
        uint16_t arcount;
};

/*
 *  Helper pointer to parse the incoming packets
 */
struct cursor {
        void *pos;
        void *end;
};


/*
 *  Initializer of a cursor pointer
 */
static __always_inline
void cursor_init(struct cursor *c, struct xdp_md *ctx)
{
        c->end = (void *)(long)ctx->data_end;
```

```
	c->pos = (void *)(long)ctx->data;
}

#define PARSE_FUNC_DECLARATION(STRUCT)			\
static __always_inline					\
struct STRUCT *parse_ ## STRUCT (struct cursor *c)	\
{							\
	struct STRUCT *ret = c->pos;			\
	if (c->pos + sizeof(struct STRUCT) > c->end)	\
		return 0;				\
	c->pos += sizeof(struct STRUCT);		\
	return ret;					\
}

PARSE_FUNC_DECLARATION(ethhdr)
PARSE_FUNC_DECLARATION(vlanhdr)
PARSE_FUNC_DECLARATION(iphdr)
PARSE_FUNC_DECLARATION(ipv6hdr)
PARSE_FUNC_DECLARATION(udphdr)
PARSE_FUNC_DECLARATION(dnshdr)

/*
 * Parse ethernet frame and fill the struct
 */
static __always_inline
struct ethhdr *parse_eth(struct cursor *c, uint16_t *eth_proto)
{
	struct ethhdr  *eth;

	if (!(eth = parse_ethhdr(c)))
		return 0;

	*eth_proto = eth->h_proto;
	if (*eth_proto == __bpf_htons(ETH_P_8021Q)
	||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
		struct vlanhdr *vlan;

		if (!(vlan = parse_vlanhdr(c)))
			return 0;

		*eth_proto = vlan->encap_proto;
		if (*eth_proto == __bpf_htons(ETH_P_8021Q)
		||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
			if (!(vlan = parse_vlanhdr(c)))
				return 0;

			*eth_proto = vlan->encap_proto;
		}
	}
	return eth;
```

```
}

/*
 *  Recalculate the checksum
 */
static __always_inline
void update_checksum(uint16_t *csum, uint16_t old_val, uint16_t new_val)
{
        uint32_t new_csum_value;
        uint32_t new_csum_comp;
        uint32_t undo;

        undo = ~((uint32_t)*csum) + ~((uint32_t)old_val);
        new_csum_value = undo + (undo < ~((uint32_t)old_val)) + (uint32_t)new_val;
        new_csum_comp = new_csum_value + (new_csum_value < ((uint32_t)new_val));
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        *csum = (uint16_t)~new_csum_comp;
}

/*
 * Parse DNS mesage.
 * Returns 1 if message needs to go through (i.e. pass)
 *        -1 if something went wrong and the packet needs to be dropped
 *         0 if (modified) message needs to be replied
 */
static __always_inline
int udp_dns_reply(struct cursor *c)
{
        struct udphdr  *udp;
        struct dnshdr  *dns;
        uint32_t        key;

        // check that we have a DNS packet
        if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(c)))
                return 1;

        // get the starting time frame from the map
        key = 0;
        struct bucket *b = bpf_map_lookup_elem(&state_map, &key);

        // the bucket must exist
        if (!b)
        {
                //bpf_printk("!FRAME \n");
                return -1;
        }

        // increment number of packets
```

```
        b->n_packets++;


        // @TODO evaluate this option for frame timing
        // look at the timing every 100 packets
         if (b->n_packets % 100 == 0)

        // look at the timing of the packet a percentage of the time
        //if (bpf_get_prandom_u32() % 100 < 50)
        {
                // get the current and elapsed time
                uint64_t now = bpf_ktime_get_ns();
                uint64_t elapsed = now - b->start_time;

                // make sure the elapsed time is set and not outside of the frame
                if (b->start_time == 0 || elapsed >= FRAME_SIZE)
                {
                        //bpf_printk("New timeframe\n");
                        // start new time frame
                        b->start_time = now;
                        b->n_packets = 0;
                }
        }

        //bpf_printk("n_packets: %llu\n", b->n_packets);

        // @TODO refine bounce rate to fit curve
        if (b->n_packets > THRESHOLD)
        {
                //bpf_printk("bounce\n");
                //save the old header values
                uint16_t old_val = dns->flags.as_value;

                // change the DNS flags
                dns->flags.as_bits_and_pieces.ad = 0;
                dns->flags.as_bits_and_pieces.qr = 1;
                dns->flags.as_bits_and_pieces.tc = 1;

                // change the UDP destination to the source
                udp->dest   = udp->source;
                udp->source = __bpf_htons(DNS_PORT);

                // calculate and write the new checksum
                update_checksum(&udp->check, old_val, dns->flags.as_value);

                // bounce
                return 0;
        }
        else
        {
```

```
                // pass
                return 1;
        }
}

/*
 *  Recieve and parse request
 *  @var struct xdp_md
 */
SEC("xdp-dns-too-many")
int xdp_dns_too_many(struct xdp_md *ctx)
{
        // store variables
        struct cursor   c;
        struct ethhdr  *eth;
        uint16_t        eth_proto;
        struct iphdr   *ipv4;
        struct ipv6hdr *ipv6;
        int            r = 0;

        // initialise the cursor
        cursor_init(&c, ctx);
        if (!(eth = parse_eth(&c, &eth_proto)))
                return XDP_PASS;

        // differentiate the parsing of the IP header based on the version
        if (eth_proto == __bpf_htons(ETH_P_IP)) {
                if (!(ipv4 = parse_iphdr(&c))
                ||    ipv4->protocol != IPPROTO_UDP
                ||   (r = udp_dns_reply(&c))) {
                        return r < 0 ? XDP_ABORTED : XDP_PASS;
                }

                uint32_t swap_ipv4 = ipv4->daddr;
                ipv4->daddr = ipv4->saddr;
                ipv4->saddr = swap_ipv4;

        } else if (eth_proto == __bpf_htons(ETH_P_IPV6)) {
                if (!(ipv6 = parse_ipv6hdr(&c))
                ||    ipv6->nexthdr  != IPPROTO_UDP
                ||   (r = udp_dns_reply(&c)))
                        return r < 0 ? XDP_ABORTED : XDP_PASS;

                struct in6_addr swap_ipv6 = ipv6->daddr;
                ipv6->daddr = ipv6->saddr;
                ipv6->saddr = swap_ipv6;
        } else {
                return XDP_PASS;
        }

```

```
        uint8_t swap_eth[ETH_ALEN];
        memcpy(swap_eth, eth->h_dest, ETH_ALEN);
        memcpy(eth->h_dest, eth->h_source, ETH_ALEN);
        memcpy(eth->h_source, swap_eth, ETH_ALEN);

        // bounce the request
        return XDP_TX;
}

char __license[] SEC("license") = "GPL";
```

C Per IP RRL prototype code

```
/*
 *  rrl-per-ip
 *  Implements a semi fine grained udp_dns_reply RRL per ip address within a time fram
 *  Jun 2020 - Tom Carpay
 */

/*
 *  Includes
 */
#include <stdint.h>
#include <linux/bpf.h>
#include <bpf_helpers.h>    /* for bpf_get_prandom_u32() */
#include <bpf_endian.h>     /* for __bpf_htons()    */
#include <linux/if_ether.h> /* for struct ethhdr   */
#include <linux/ip.h>       /* for struct iphdr    */
#include <linux/ipv6.h>     /* for struct ipv6hdr  */
#include <linux/in.h>       /* for IPPROTO_UDP     */
#include <linux/udp.h>      /* for struct udphdr   */
#include <string.h>         /* for memcpy()        */

/*
 *  Begin defines
 */
#define DNS_PORT      53

#define FRAME_SIZE           1000000000
#define THRESHOLD            1000
/*
 *  End defines
 */

/*
 *  Store the time frame
 */
struct bucket {
        uint64_t start_time;
        uint64_t n_packets;
};

struct bpf_map_def SEC("maps") state_map = {
        .type = BPF_MAP_TYPE_PERCPU_HASH,
        .key_size = sizeof(uint32_t),
        .value_size = sizeof(struct bucket),
        .max_entries = 100
};

struct bpf_map_def SEC("maps") state_map_v6 = {
        .type = BPF_MAP_TYPE_PERCPU_HASH,
```

```
        .key_size = sizeof(struct in6_addr),
        .value_size = sizeof(struct bucket),
        .max_entries = 100
};

/*
 *  Store the VLAN header
 */
struct vlanhdr {
        uint16_t tci;
        uint16_t encap_proto;
};

/*
 *  Store the DNS header
 */
struct dnshdr {
        uint16_t id;
        union {
                struct {
                        uint8_t  rd     : 1;
                        uint8_t  tc     : 1;
                        uint8_t  aa     : 1;
                        uint8_t  opcode : 4;
                        uint8_t  qr     : 1;

                        uint8_t  rcode  : 4;
                        uint8_t  cd     : 1;
                        uint8_t  ad     : 1;
                        uint8_t  z      : 1;
                        uint8_t  ra     : 1;
                }        as_bits_and_pieces;
                uint16_t as_value;
        } flags;
        uint16_t qdcount;
        uint16_t ancount;
        uint16_t nscount;
        uint16_t arcount;
};

/*
 *  Helper pointer to parse the incoming packets
 */
struct cursor {
        void *pos;
        void *end;
};


/*
```

```
 *  Initializer of a cursor pointer
 */
static __always_inline
void cursor_init(struct cursor *c, struct xdp_md *ctx)
{
        c->end = (void *)(long)ctx->data_end;
        c->pos = (void *)(long)ctx->data;
}

#define PARSE_FUNC_DECLARATION(STRUCT)                          \
static __always_inline                                                  \
struct STRUCT *parse_ ## STRUCT (struct cursor *c)          \
{                                                           \
        struct STRUCT *ret = c->pos;                            \
        if (c->pos + sizeof(struct STRUCT) > c->end)            \
                return 0;                                   \
        c->pos += sizeof(struct STRUCT);                    \
        return ret;                                             \
}

PARSE_FUNC_DECLARATION(ethhdr)
PARSE_FUNC_DECLARATION(vlanhdr)
PARSE_FUNC_DECLARATION(iphdr)
PARSE_FUNC_DECLARATION(ipv6hdr)
PARSE_FUNC_DECLARATION(udphdr)
PARSE_FUNC_DECLARATION(dnshdr)

/*
 * Parse ethernet frame and fill the struct
 */
static __always_inline
struct ethhdr *parse_eth(struct cursor *c, uint16_t *eth_proto)
{
        struct ethhdr  *eth;

        if (!(eth = parse_ethhdr(c)))
                return 0;

        *eth_proto = eth->h_proto;
        if (*eth_proto == __bpf_htons(ETH_P_8021Q)
        ||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
                struct vlanhdr *vlan;

                if (!(vlan = parse_vlanhdr(c)))
                        return 0;

                *eth_proto = vlan->encap_proto;
                if (*eth_proto == __bpf_htons(ETH_P_8021Q)
                ||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
                        if (!(vlan = parse_vlanhdr(c)))
```

```
                                return 0;

                        *eth_proto = vlan->encap_proto;
                }
        }
        return eth;
}

/*
 *  Recalculate the checksum
 */
static __always_inline
void update_checksum(uint16_t *csum, uint16_t old_val, uint16_t new_val)
{
        uint32_t new_csum_value;
        uint32_t new_csum_comp;
        uint32_t undo;

        undo = ~((uint32_t)*csum) + ~((uint32_t)old_val);
        new_csum_value = undo + (undo < ~((uint32_t)old_val)) + (uint32_t)new_val;
        new_csum_comp = new_csum_value + (new_csum_value < ((uint32_t)new_val));
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        *csum = (uint16_t)~new_csum_comp;
}

static __always_inline
int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
{
        // increment number of packets
        b->n_packets++;

        // get the current and elapsed time
        uint64_t now = bpf_ktime_get_ns();
        uint64_t elapsed = now - b->start_time;

        // make sure the elapsed time is set and not outside of the frame
        if (b->start_time == 0 || elapsed >= FRAME_SIZE)
        {
                //bpf_printk("New timeframe\n");
                // start new time frame
                b->start_time = now;
                b->n_packets = 0;
        }

        // @TODO refine bounce rate to fit curve
        if (b->n_packets < THRESHOLD)
                return 1;

        //bpf_printk("bounce\n");
```

```
        //save the old header values
        uint16_t old_val = dns->flags.as_value;

        // change the DNS flags
        dns->flags.as_bits_and_pieces.ad = 0;
        dns->flags.as_bits_and_pieces.qr = 1;
        dns->flags.as_bits_and_pieces.tc = 1;

        // change the UDP destination to the source
        udp->dest   = udp->source;
        udp->source = __bpf_htons(DNS_PORT);

        // calculate and write the new checksum
        update_checksum(&udp->check, old_val, dns->flags.as_value);

        // bounce
        return 0;
}

/*
 * Parse DNS ipv4 message
 * Returns 1 if message needs to go through (i.e. pass)
 *        -1 if something went wrong and the packet needs to be dropped
 *         0 if (modified) message needs to be replied
 */
static __always_inline
int udp_dns_reply_v4(struct cursor *c, uint32_t key)
{
        struct udphdr  *udp;
        struct dnshdr  *dns;

        // check that we have a DNS packet
        if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(c)))
                return 1;

        // get the starting time frame from the map
        struct bucket *b = bpf_map_lookup_elem(&state_map, &key);

        // the bucket must exist
        if (b)
                return do_rate_limit(udp, dns, b);

        // create new starting bucket for this key
        struct bucket new_bucket;
        new_bucket.start_time = bpf_ktime_get_ns();
        new_bucket.n_packets = 0;

        // store the bucket and pass the packet
        bpf_map_update_elem(&state_map, &key, &new_bucket, BPF_ANY);
```

```
        return 1;
}

/*
 * Parse DNS mesage.
 * Returns 1 if message needs to go through (i.e. pass)
 *        -1 if something went wrong and the packet needs to be dropped
 *         0 if (modified) message needs to be replied
 */
static __always_inline
int udp_dns_reply_v6(struct cursor *c, struct in6_addr *key)
{
         struct udphdr  *udp;
         struct dnshdr  *dns;

         // check that we have a DNS packet
         if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
         ||  !(dns = parse_dnshdr(c)))
                 return 1;

         // get the starting time frame from the map
         struct bucket *b = bpf_map_lookup_elem(&state_map_v6, key);

         // the bucket must exist
        if (b)
                return do_rate_limit(udp, dns, b);

        // create new starting bucket for this key
        struct bucket new_bucket;
        new_bucket.start_time = bpf_ktime_get_ns();
        new_bucket.n_packets = 0;

        // store the bucket and pass the packet
        bpf_map_update_elem(&state_map_v6, key, &new_bucket, BPF_ANY);
        return 1;
}


/*
 *  Recieve and parse request
 *  @var struct xdp_md
 */
SEC("xdp-dns-too-many")
int xdp_dns_too_many(struct xdp_md *ctx)
{
        // store variables
        struct cursor   c;
        struct ethhdr  *eth;
        uint16_t        eth_proto;
        struct iphdr   *ipv4;
```

```
	struct ipv6hdr *ipv6;
	int            r = 0;

	// initialise the cursor
	cursor_init(&c, ctx);
	if (!(eth = parse_eth(&c, &eth_proto)))
		return XDP_PASS;

	// differentiate the parsing of the IP header based on the version
	if (eth_proto == __bpf_htons(ETH_P_IP))
	{
		if (!(ipv4 = parse_iphdr(&c))
		||    ipv4->protocol != IPPROTO_UDP
		||   (r = udp_dns_reply_v4(&c, ipv4->saddr))) {

			return r < 0 ? XDP_ABORTED : XDP_PASS;
		}

		uint32_t swap_ipv4 = ipv4->daddr;
		ipv4->daddr = ipv4->saddr;
		ipv4->saddr = swap_ipv4;

	}
	else if (eth_proto == __bpf_htons(ETH_P_IPV6))
	{
		if (!(ipv6 = parse_ipv6hdr(&c))
		||    ipv6->nexthdr  != IPPROTO_UDP
		||   (r = udp_dns_reply_v6(&c, &ipv6->saddr)))
			return r < 0 ? XDP_ABORTED : XDP_PASS;

		struct in6_addr swap_ipv6 = ipv6->daddr;
		ipv6->daddr = ipv6->saddr;
		ipv6->saddr = swap_ipv6;
	}
	else
	{
		return XDP_PASS;
	}

	uint8_t swap_eth[ETH_ALEN];
	memcpy(swap_eth, eth->h_dest, ETH_ALEN);
	memcpy(eth->h_dest, eth->h_source, ETH_ALEN);
	memcpy(eth->h_source, swap_eth, ETH_ALEN);

	// Bounce

	// bounce the request
	return XDP_TX;
}
```

```
char __license[] SEC("license") = "GPL";
```

D Unknown host RRL prototype code

```c
/*
 *  rrl-per-ip
 *  Implements per IP RLL within a time frame for hosts that are not known in hte
 *  Jun 2020 - Tom Carpay
 */

/*
 *  Includes
 */
#include <stdint.h>
#include <linux/bpf.h>
#include <bpf_helpers.h>    /* for bpf_get_prandom_u32() */
#include <bpf_endian.h>     /* for __bpf_htons()    */
#include <linux/if_ether.h> /* for struct ethhdr   */
#include <linux/ip.h>       /* for struct iphdr    */
#include <linux/ipv6.h>     /* for struct ipv6hdr  */
#include <linux/in.h>       /* for IPPROTO_UDP     */
#include <linux/udp.h>      /* for struct udphdr   */
#include <string.h>         /* for memcpy()        */

/*
 *  Begin defines
 */
#define DNS_PORT      53

#define FRAME_SIZE           1000000000
#define THRESHOLD            1000
/*
 *  End defines
 */

/*
 *  Store the time frame
 */
struct bucket {
        uint64_t start_time;
        uint64_t n_packets;
};

struct bpf_map_def SEC("maps") state_map = {
        .type = BPF_MAP_TYPE_PERCPU_HASH,
        .key_size = sizeof(uint32_t),
        .value_size = sizeof(struct bucket),
        .max_entries = 100 // Enough for testing purposes
};

struct bpf_map_def SEC("maps") state_map_v6 = {
        .type = BPF_MAP_TYPE_PERCPU_HASH,
```

```
        .key_size = sizeof(struct in6_addr),
        .value_size = sizeof(struct bucket),
        .max_entries = 100 // Enough for testing purposes
};

/*
 *  Smallest storage space possible
 */
struct data {
        uint8_t unused;
};

struct bpf_map_def SEC("maps") known_hosts = {
        .type = BPF_MAP_TYPE_HASH,
        .key_size = sizeof(uint32_t),
        .value_size = sizeof(struct data),
        .max_entries = 100 // Enough for testing purposes
};

struct bpf_map_def SEC("maps") known_hosts_v6 = {
        .type = BPF_MAP_TYPE_HASH,
        .key_size = sizeof(struct in6_addr),
        .value_size = sizeof(struct data),
        .max_entries = 100 // Enough for testing purposes
};

/*
 *  Store the VLAN header
 */
struct vlanhdr {
        uint16_t tci;
        uint16_t encap_proto;
};

/*
 *  Store the DNS header
 */
struct dnshdr {
        uint16_t id;
        union {
                struct {
                        uint8_t  rd     : 1;
                        uint8_t  tc     : 1;
                        uint8_t  aa     : 1;
                        uint8_t  opcode : 4;
                        uint8_t  qr     : 1;

                        uint8_t  rcode  : 4;
                        uint8_t  cd     : 1;
                        uint8_t  ad     : 1;
```

```
                        uint8_t  z      : 1;
                        uint8_t  ra     : 1;
                } as_bits_and_pieces;
                uint16_t as_value;
        } flags;
        uint16_t qdcount;
        uint16_t ancount;
        uint16_t nscount;
        uint16_t arcount;
};

/*
 *  Helper pointer to parse the incoming packets
 */
struct cursor {
        void *pos;
        void *end;
};


/*
 *  Initializer of a cursor pointer
 */
static __always_inline
void cursor_init(struct cursor *c, struct xdp_md *ctx)
{
        c->end = (void *)(long)ctx->data_end;
        c->pos = (void *)(long)ctx->data;
}

#define PARSE_FUNC_DECLARATION(STRUCT)                          \
static __always_inline                                                  \
struct STRUCT *parse_ ## STRUCT (struct cursor *c)          \
{                                                               \
        struct STRUCT *ret = c->pos;                            \
        if (c->pos + sizeof(struct STRUCT) > c->end)           \
                return 0;                                       \
        c->pos += sizeof(struct STRUCT);                      \
        return ret;                                             \
}

PARSE_FUNC_DECLARATION(ethhdr)
PARSE_FUNC_DECLARATION(vlanhdr)
PARSE_FUNC_DECLARATION(iphdr)
PARSE_FUNC_DECLARATION(ipv6hdr)
PARSE_FUNC_DECLARATION(udphdr)
PARSE_FUNC_DECLARATION(dnshdr)

/*
 * Parse ethernet frame and fill the struct
```

```
 */
static __always_inline
struct ethhdr *parse_eth(struct cursor *c, uint16_t *eth_proto)
{
        struct ethhdr  *eth;

        if (!(eth = parse_ethhdr(c)))
                return 0;

        *eth_proto = eth->h_proto;
        if (*eth_proto == __bpf_htons(ETH_P_8021Q)
        ||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
                struct vlanhdr *vlan;

                if (!(vlan = parse_vlanhdr(c)))
                        return 0;

                *eth_proto = vlan->encap_proto;
                if (*eth_proto == __bpf_htons(ETH_P_8021Q)
                ||  *eth_proto == __bpf_htons(ETH_P_8021AD)) {
                        if (!(vlan = parse_vlanhdr(c)))
                                return 0;

                        *eth_proto = vlan->encap_proto;
                }
        }
        return eth;
}

/*
 *  Recalculate the checksum
 */
static __always_inline
void update_checksum(uint16_t *csum, uint16_t old_val, uint16_t new_val)
{
        uint32_t new_csum_value;
        uint32_t new_csum_comp;
        uint32_t undo;

        undo = ~((uint32_t)*csum) + ~((uint32_t)old_val);
        new_csum_value = undo + (undo < ~((uint32_t)old_val)) + (uint32_t)new_val;
        new_csum_comp = new_csum_value + (new_csum_value < ((uint32_t)new_val));
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        new_csum_comp = (new_csum_comp & 0xFFFF) + (new_csum_comp >> 16);
        *csum = (uint16_t)~new_csum_comp;
}

static __always_inline
int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
{
```

```c
        // increment number of packets
        b->n_packets++;

        // get the current and elapsed time
        uint64_t now = bpf_ktime_get_ns();
        uint64_t elapsed = now - b->start_time;

        // make sure the elapsed time is set and not outside of the frame
        if (b->start_time == 0 || elapsed >= FRAME_SIZE)
        {
                //bpf_printk("New timeframe\n");
                // start new time frame
                b->start_time = now;
                b->n_packets = 0;
        }

        // @TODO refine bounce rate to fit curve
        if (b->n_packets < THRESHOLD)
                return 1;

        //bpf_printk("bounce\n");
        //save the old header values
        uint16_t old_val = dns->flags.as_value;

        // change the DNS flags
        dns->flags.as_bits_and_pieces.ad = 0;
        dns->flags.as_bits_and_pieces.qr = 1;
        dns->flags.as_bits_and_pieces.tc = 1;

        // change the UDP destination to the source
        udp->dest   = udp->source;
        udp->source = __bpf_htons(DNS_PORT);

        // calculate and write the new checksum
        update_checksum(&udp->check, old_val, dns->flags.as_value);

        // bounce
        return 0;
}

/*
 * Parse DNS ipv4 message
 * Returns 1 if message needs to go through (i.e. pass)
 *        -1 if something went wrong and the packet needs to be dropped
 *         0 if (modified) message needs to be replied
 */
static __always_inline
int udp_dns_reply_v4(struct cursor *c, uint32_t key)
{
        struct udphdr  *udp;
```

```
        struct dnshdr  *dns;

        // check that we have a DNS packet
        if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(c)))
                return 1;

        // get the host from the list
        struct data *host = bpf_map_lookup_elem(&known_hosts, &key);

        // if the host is known we do not rate limit it
        if (host)
        {
                // pass
                return 1;
        }

        // get the starting time frame from the map
        struct bucket *b = bpf_map_lookup_elem(&state_map, &key);


        //bpf_printk("ip: %u\n", key);

        // the bucket must exist
        if (b)
                return do_rate_limit(udp, dns, b);

        // create new starting bucket for this key
        struct bucket new_bucket;
        new_bucket.start_time = bpf_ktime_get_ns();
        new_bucket.n_packets = 0;

        // store the bucket and pass the packet
        bpf_map_update_elem(&state_map, &key, &new_bucket, BPF_ANY);
        return 1;
}

/*
 * Parse DNS mesage.
 * Returns 1 if message needs to go through (i.e. pass)
 *         -1 if something went wrong and the packet needs to be dropped
 *          0 if (modified) message needs to be replied
 */
static __always_inline
int udp_dns_reply_v6(struct cursor *c, struct in6_addr *key)
{
         struct udphdr  *udp;
         struct dnshdr  *dns;

         // check that we have a DNS packet
```

```
	if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
	||  !(dns = parse_dnshdr(c)))
		return 1;

	// get the host from the list
	struct data *host = bpf_map_lookup_elem(&known_hosts_v6, key);

	// if the host is known we do not rate limit it
	if (host)
	{
		// pass
		return 1;
	}

	// get the starting time frame from the map
	struct bucket *b = bpf_map_lookup_elem(&state_map_v6, key);

	// the bucket must exist
	if (b)
		return do_rate_limit(udp, dns, b);

	// create new starting bucket for this key
	struct bucket new_bucket;
	new_bucket.start_time = bpf_ktime_get_ns();
	new_bucket.n_packets = 0;

	// store the bucket and pass the packet
	bpf_map_update_elem(&state_map_v6, key, &new_bucket, BPF_ANY);
	return 1;
}


/*
 *  Recieve and parse request
 *  @var struct xdp_md
 */
SEC("xdp-dns-too-many")
int xdp_dns_too_many(struct xdp_md *ctx)
{
	// store variables
	struct cursor   c;
	struct ethhdr  *eth;
	uint16_t        eth_proto;
	struct iphdr   *ipv4;
	struct ipv6hdr *ipv6;
	int             r = 0;

	// initialise the cursor
	cursor_init(&c, ctx);
	if (!(eth = parse_eth(&c, &eth_proto)))
```

```c
			return XDP_PASS;

	// differentiate the parsing of the IP header based on the version
	if (eth_proto == __bpf_htons(ETH_P_IP))
	{
		if (!(ipv4 = parse_iphdr(&c))
		||    ipv4->protocol != IPPROTO_UDP
		||   (r = udp_dns_reply_v4(&c, ipv4->saddr))) {

			return r < 0 ? XDP_ABORTED : XDP_PASS;
		}

		uint32_t swap_ipv4 = ipv4->daddr;
		ipv4->daddr = ipv4->saddr;
		ipv4->saddr = swap_ipv4;

	}
	else if (eth_proto == __bpf_htons(ETH_P_IPV6))
	{
		if (!(ipv6 = parse_ipv6hdr(&c))
		||    ipv6->nexthdr  != IPPROTO_UDP
		||   (r = udp_dns_reply_v6(&c, &ipv6->saddr)))
			return r < 0 ? XDP_ABORTED : XDP_PASS;

		struct in6_addr swap_ipv6 = ipv6->daddr;
		ipv6->daddr = ipv6->saddr;
		ipv6->saddr = swap_ipv6;
	}
	else
	{
		return XDP_PASS;
	}

	uint8_t swap_eth[ETH_ALEN];
	memcpy(swap_eth, eth->h_dest, ETH_ALEN);
	memcpy(eth->h_dest, eth->h_source, ETH_ALEN);
	memcpy(eth->h_source, swap_eth, ETH_ALEN);

	// Bounce

	// bounce the request
	return XDP_TX;
}

char __license[] SEC("license") = "GPL";
```