XDP-based DNS hot cache

Jannik Samuel Peters
University of Amsterdam
jannik.peters@os3.nl

Willem Toorop
Supervisor
NLnet Labs
willem@nlnetlabs.nl

Luuk Hendriks
Supervisor
NLnet Labs
luuk@nlnetlabs.nl

***Abstract*—Nameservers are the basis of the Domain Name System (DNS). Some of them, like public resolvers, the root servers or many top-level domain servers, are queried billions of times each day, requiring an enormous number of server instances worldwide to accommodate such a high query volume. The aim of this project is to investigate the feasibility of using eBPF, specifically XDP, to answer certain queries directly from the kernel, reducing the resources, and thus the cost, needed to operate certain domains or resolvers. A benefit of using XDP is that it can be applied to any existing deployment running Linux, regardless of the nameserver software used. For this project, we assess the current limitations of growing and modifying the frame in XDP to accommodate the desired DNS responses by building a proof of concept. We show that the limitations vary per device driver, and that the current state of eBPF and XDP severely limits the extent to which DNS responses can be sent this way.**

***Index Terms*—eBPF, XDP, DNS, Cache, Rust**

I. Introduction

eBPF and specifically the eXpress Data Path (XDP) allow for processing of packets in the Linux kernel without touching the full network stack or user space, potentially reducing the system load significantly. It is used, amongst others, for different applications like DDoS protection, load balancing, or traffic inspection.

Nameservers for the root and top-level domains are subject to a very high number of requests; more than 50% of these are non-existent domains [1]. This high load requires operating an increased number of instances to fulfill this request volume.

XDP seems promising to reduce the burden on the hardware of the nameservers by answering queries at the earliest point possible in the software, potentially reducing the number of instances necessary to serve the queries for highly queried domains and therefore reducing the cost to operate them.

II. Research Questions

To guide the project in investigating the feasibility of an XDP-based DNS hot cache, we define the following research questions:

- To what extent is it possible to answer DNS queries from XDP?
- What type of responses can be returned from XDP, and to what extent?
- Can those responses be learned and cached from (user space) DNS software, and to what extent?

III. Related Work

Previous work, specifically on responding to DNS queries from XDP, has been done by my supervisors, Willem Toorop and Luuk Hendriks from NLnet Labs, and Tom Carpay. They explored the possibility of augmenting DNS servers using XDP by implementing programs that respond with a REFUSED status code [2] or perform Request Rate Limiting [3], [4], [5]. Other projects used XDP to modify incoming frame sizes [6], to perform layer 4 load balancing [7], or to implement DDoS protection [8], [9]. These projects did not use XDP to reply to DNS queries with data responses or cache responses sent by user space DNS software. However, there have been attempts at implementing a simple DNS server in XDP that is supposed to be able to serve A records [10] and at implementing a caching layer using XDP (just like our project) that wasn't completed [11]. Both projects do not work, either because the code is out-dated or because development ended before completion. Either way, XDP has changed a lot since the mentioned attempts, so that the state of the art of using XDP for DNS caching can be explored further and, more importantly, documented.

IV. Approach

We will evaluate the possibilities of augmenting a DNS server with an XDP-based hot cache by incrementally exploring the current state of XDP and its constraints on the types of responses and sizes that can be cached and sent this way and implementing a proof of concept.

We start by exploring the different functionalities of XDP that might support the development of the prototype, documenting its capabilities and limitations. Subsequently, we implement parsing the query name of incoming queries from XDP and answering with a fixed response for a single domain delegation. We also evaluate storage opportunities for cached responses, such as BPF maps or data compiled into the program binary.

V. Introduction to XDP

To better understand the context of this endeavor and the planned architecture it is helpful to understand the integration of XDP into the Linux kernel and its network stack. XDP programs are executed on every frame reaching the system. In native and offloaded execution they are executed at the very first point in software, directly after the network driver has received the frame (see Figure 1). This allows inspecting and modifying the frames before other operations, such as allocating the kernel internal representation of network frames (`sk_buff`), are performed. The XDP program can then forward the potentially modified frame to the network stack, redirect it to a different interface, send it out the interface it came in, or drop it. However, the program can only act within the context of the current frame, meaning it cannot create and send out a completely new one. XDP programs have direct access to the frame's data in the driver, providing the need for it to be laid out linearly in a ”single DMA'ed page” [12]. This early and direct access to the frame's data allows for very high speed handling of network data.

Before attaching an eBPF program to its designated kernel hook, an in-kernel verifier checks the loaded program, to ensure the safety and stability of the kernel. The verifier checks for unreachable program paths, invalid memory access, or other undesired behavior. This has implications on the code we can write, e.g. we need to carefully check that each pointer access to the frame's data is within the bounds of the current frame.

For a more in-depth introduction to eBPF and XDP, please refer to the Cilium BPF and XDP Reference Guide [13] and a paper showing the benefit of using XDP for high performance network applications [14].

VI. Planned Architecture

An XDP program will be used to inspect incoming frames and parse DNS queries to UDP port 53. According to the parsed query name, the frame will need to grow to fit and add the appropriate response, as we can only alter the already existing, incoming frame. eBPF provides helper functions that allow growing or shrinking a frame at its head or tail. These functions can fail, depending on the space requested and available. It's unclear how much space is available and will be evaluated in section VIII.

As XDP programs are only executed on received frames, we need a different program type to read outgoing responses to DNS queries. eBPF provides a program type that is executed in the traffic control (tc) subsystem of the network stack, which allows running tc programs on outgoing frames. We will use such a program to fill the cache used by the XDP programs to answer incoming queries. This cache could either be a (shared) BPF map (which might have performance drawbacks because of locking) or static data in the XDP program code.

VII. Development

The eBPF programs will be written in Rust using the aya-rs eBPF library [15]. During development, the code will need to be tested frequently, to ensure a valid eBPF program, by repeatedly compiling and loading the program, and inspecting the eBPF verifier's error messages.

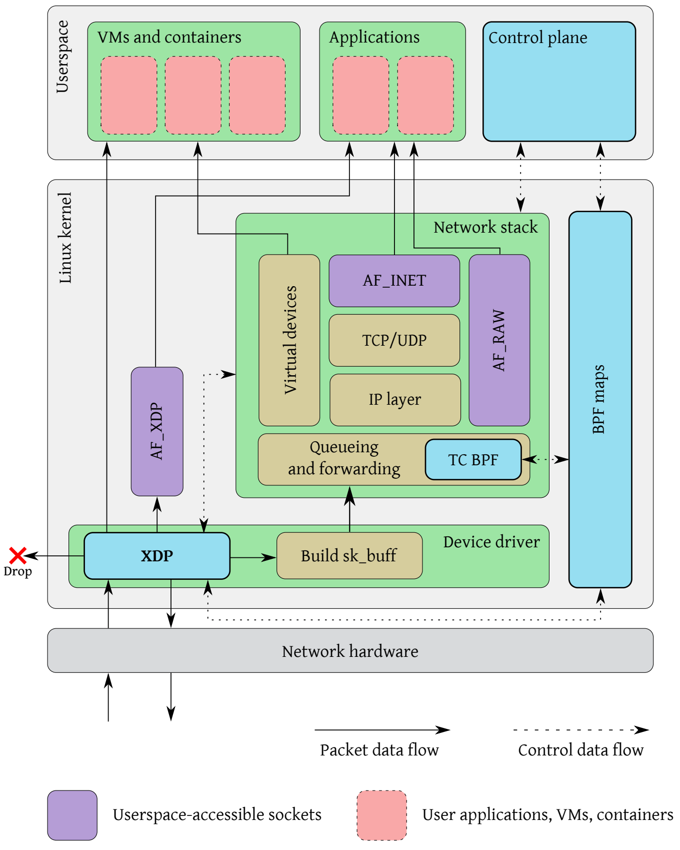

Fig. 1: ”XDP's integration with the Linux network stack. On packet arrival, before touching the packet data, the device driver executes an eBPF program in the main XDP hook. This program can choose to drop packets; to send them back out the same interface it was received on; to redirect them, either to another interface [...] or to userspace through special AF_XDP sockets; or to allow them to proceed to the regular networking stack, where a separate TC BPF hook can perform further processing before packets are queued for transmission. The different eBPF programs can communicate with each other and with userspace through the use of BPF maps. To simplify the diagram, only the ingress path is shown.” (Figure and caption by Høiland-Jørgensen, Brouer, Borkmann, *et al.* [14] licensed under CC BY-SA 4.0)

VIII. Experiments

The research included multiple experiments. We investigated the available tailroom and headroom for two different drivers and experimented with multiple ways of implementing the parsing and comparison of the query name.

A. Available Tailroom

To investigate the available tailroom we implemented a basic XDP program that just sends out the incoming frame with swapped source and destination addresses and ports (see listing 1 for an example).

```
let src_addr_be = (*ipv4hdr).src_addr;
(*ipv4hdr).src_addr = (*ipv4hdr).dst_addr;
(*ipv4hdr).dst_addr = src_addr_be;
```

Listing 1: Swapping IPv4 source and destination

The XDP variant of eBPF programs provides the function `long bpf_xdp_adjust_tail(struct xdp_buff *xdp_md, int delta)`, which allows growing or shrinking the frame at the tail. In this case we grow the frame for different values for delta. We adjusted the delta repeatedly while sending the same DNS query until discovering the maximum working value documenting the initial frame size, the delta, and the resulting frame size. We tested different initial frame sizes using different queries to investigate if the amount of available space changes. Table I shows the different initial frame sizes, the according maximum amount of space available, as well as the resulting frame sizes. Out of curiosity we also tested if the available space changes when the initial frame size exceeds the previously discovered maximum adjusted frame sizes. These larger initial frame sizes will not occur during normal operation of the DNS but were an interesting observation of the capabilities of XDP.

TABLE I: Available space when running as **generic** XDP attached to the **loopback** device

Initial Frame Size	Tailroom (Resulting Frame Size)	Headroom
82[a]	+364 Bytes (446)	+218 Bytes
84[b]	+362 Bytes (446)	+218 Bytes
146[c]	+300 Bytes (446)	+218 Bytes
488	+982 Bytes (1470)	+218 Bytes
554	+916 Bytes (1470)	+218 Bytes

[a] `Query: . NS`
[b] `Query: a. NS`
[c] `Query: <63-character-long-label>. NS`

Most of the testing and development was conducted against the loopback device of the development machine—a laptop running Manjaro Linux with kernel version 6.6.10-1-MANJARO. Unfortunately, the loopback device doesn't support native XDP, so we repeated the above experiment against a network device of a Virtual Machine using the `virtio_net` driver and discovering a much larger leeway to adjust the frame size (see Table II). This shows that the available tailroom highly depends on the driver in use.

TABLE II: Available space when running as **native** XDP using the **virtio_net** driver

Initial Frame Size	Tailroom (Resulting Frame Size)	Headroom
82[a]	+3426 Bytes (3508)	+224 Bytes

[a] `Query: . NS`

B. *Available Headroom*

Adjacent to the evaluation of the available tailroom we tested for the available headroom. It is supposed to be 256 Bytes [12], but is actually less than that, as the Tables I and II show. Conveniently, the missing headroom of the virtio device is exactly 32 Bytes, which is the maximum size usable for metadata [16]. For unknown reasons, the available headroom for the loopback device is smaller. An inconvenience with using the headroom is that the actual data in the frame needs to be moved forward. The implementation for that can be a bit tricky as the verifier only allows copying chunks of a fixed number of Bytes, not allowing it to be deduced from the actual length of the frame.

C. *Parsing the query name*

There were multiple ideas for how to parse and store the domain name of the incoming query. One was saving the pointer to the start and the length to access the domain name for later comparison. However, this didn't work out that well as accessing the byte array this way implies illegal pointer arithmetic, therefore rejecting many attempts at continuing this path. Attempting to circumvent verifier limitations regarding bounds checks using tail calls didn't help either. In the end we resulted to checking the size of the query name using multiple static limits and copying each byte of the query name into a dedicated byte buffer.

IX. Prototype

The final prototype unfortunately doesn't include all the features we intended to achieve. For example, we did not get to implement the actual caching part in the tc layer that would read outgoing responses and fill the cache appropriately. We also didn't get to respond to queries with answers from a map that could be filled by a user space program. This means our prototype can answer specific hard-coded domains with hard-coded responses. One seems to need to be very precise in the requirements during development, as implementing a somewhat variable program turned out to be very difficult.

X. Evaluation

As described above, this project didn't touch on all the aspects of what we set out to do. We also didn't check which drivers are used in the wild to test their capabilities. Another potential issue during development was the choice of programming language. It appears that the Rust compiler creates very different code from the C compiler, which seems to render some code that worked in C unusable.

Many of the most-used delegations at the root require a larger amount of tailroom than was available in the environment used during this project [17]. For the experimentation with responding to queries using this prototype, we only tested with a small enough response, namely the delegation for the top-level domain `nl`.

Also, the verifier has strict limits to what code it allows. For instance, it currently only allows 1 000 000 instructions per program. The verifier also doesn't allow loops, meaning they have to be unrolled. This, however, increases the number of instructions generated by the compiler enormously when the loops contain complicated logic or even other loops, so they have to be used with care.

Using a shared BPF map could have performance implications compared to static data embedded in the binary, as there will be a need for locking when accessing map data from multiple sources. Therefore, it could be beneficial to only use static data embedded into the program. However, there could still be a use-case for a program at the tc layer to cache frequently sent responses. A user space program could then transform that cache into an XDP program to replace the existing one, updating the "cache". This could definitely be a valid approach, as it is an atomic operation to load, unload, or replace an XDP program [12], meaning no interruption to network traffic—although, for authoritative nameservers, creating a new program while changing the zone file of the nameserver could be an easier approach.

XI. Conclusions

There have been many obstacles in this endeavor. One of them was the variability of the desired program, trying to parse a variable-length payload meant encountering many complaints of the verifier. We did, however, manage to find some answers to the research questions defined above. Yes, it is possible to answer with actual data responses to certain queries. The extent of the answers is limited by the space the environment (i.e., device driver) allows us to use. Any response that would fit into the original query, i.e., any response that only changes DNS header information like flags or opcodes, are definitely possible to produce using XDP. Responses that need additional data, i.e., responses with Resource Records, need to be small enough to fit into the available space—a Virtual Machine with the virtio_net driver will likely have enough space for most responses. The capabilities of drivers for actual hardware were not explored.

Another difficulty is the current state of the verifier not allowing many variable bounds checks with numbers not hard-coded into the program, but read from the incoming frame instead. This would make the code easier to write and read. However, the Rust macro system could probably be used for writing the many hard-coded bounds checks, and have them still be easy to read and write, as it could generate them from shorter code. The other pressing issue of these constraints is that the program in question cannot be too generic in its functionality. It has to be fairly precise in its application and scope.

XII. Future Work

There is still much work to do in terms of actually caching responses from a user space DNS software, as well as finding out the constraints of different real-world environments (i.e., space constraints of drivers for hardware with native or offloaded XDP support). The proof of concept could also be enhanced to accommodate the most queried domains and to allow for easy addition or modification of response data, as well as supporting partial responses (e.g., by omitting some glue).

There would also be a public benefit in improving the eBPF verifier to allow more valid programs, i.e., supporting more variable bounds checks. Another improvement could be made to the documentation or the API for growing the frame, by adding information about how much space is available, alleviating the need for trial and error.

References

[1] G. Huston. "The root of the DNS revisited." (Feb. 8, 2023), [Online]. Available: https://blog.apnic.net/2023/02/08/the-root-of-the-dns-revisited/ (visited on 01/09/2024).

[2] L. Hendriks and W. Toorop. "Journeying into XDP: Part 0." (Jul. 20, 2020), [Online]. Available: https://labs.ripe.net/author/luuk_hendriks/journeying-into-xdp-part-0/ (visited on 01/10/2024).

[3] T. Carpay, L. Hendriks, and W. Toorop. "Journeying into XDP part 1: Augmenting DNS." (Oct. 23, 2020), [Online]. Available: https://labs.ripe.net/author/tom_carpay/journeying-into-xdp-part-1-augmenting-dns/ (visited on 01/10/2024).

[4] W. Toorop, T. Carpay, and L. Hendriks. "Journeying into XDP: Fully-fledged DNS service augmentation." (Feb. 15, 2022), [Online]. Available: https://blog.apnic.net/2022/02/15/journeying-into-xdp-fully-fledged-dns-service-augmentation/ (visited on 01/10/2024).

[5] T. Carpay, "Server agnostic DNS augmentation using eBPF," University of Amsterdam, thesis, Aug. 17, 2020. [Online]. Available: https://rp.os3.nl/2019-2020/p05/report.pdf (visited on 01/15/2024).

[6] A. Koolhaas and T. Slokkker, "Defragmenting DNS, Determining the optimal maximum UDP response size for DNS," University of Amsterdam, thesis, Jul. 5, 2020. [Online]. Available: https://www.nlnetlabs.nl/downloads/publications/os3-2020-rp2-defragmenting-dns.pdf (visited on 01/15/2024).

[7] N. Shirokov and R. Dasineni. "Open-sourcing Katran, a scalable network load balancer." (May 22, 2018), [Online]. Available: https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/ (visited on 01/15/2024).

[8] "Deep dive into Facebook's BPF edge firewall." (Nov. 2018), [Online]. Available: https://cilium.io/blog/2018/11/20/fb-bpf-firewall/ (visited on 02/07/2024).

[9] A. Fabre. "L4Drop: XDP DDoS mitigations." (Nov. 28, 2018), [Online]. Available: https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/ (visited on 02/07/2024).

[10] Zebaz, *Xpress DNS - experimental XDP DNS server*, Oct. 4, 2021. [Online]. Available: https://github.com/zebaz/xpress-dns (visited on 01/15/2024).

[11] M. MICHISHITA, *BDC - eBPF DNS cache*, May 13, 2022. [Online]. Available: https://github.com/aztecher/bdc/ (visited on 01/15/2024).

[12] "BPF and XDP reference guide – program types." (2024), [Online]. Available: https://docs.cilium.io/en/latest/bpf/progtypes/#xdp (visited on 02/07/2024).

[13] "BPF and XDP reference guide." (2024), [Online]. Available: https://docs.cilium.io/en/latest/bpf/ (visited on 02/09/2024).

[14] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, *et al.*, "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*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[15] *Aya, Aya is an eBPF library for the Rust programming language, built with a focus on developer experience and operability.* Version 0.11.0, 2021. [Online]. Available: https://github.com/aya-rs/aya (visited on 02/09/2024).

[16] *Linux kernel source tree*, 2024. [Online]. Available: https://github.com/torvalds/linux (visited on 02/07/2024).

[17] "Root zone file." (2024), [Online]. Available: https://www.internic.net/domain/root.zone (visited on 02/10/2024).

Appendix A
Selection of Code

For brevity the generated scaffolding code is not included, only own files which have changed will be shown. The full project structure can either be obtained from `https://gitlab.os3.nl/jpeters/rp1-code-xdp-dns-cache` or `https://github.com/mozzieongit/xdp-dns-cache`, or generated using `cargo generate --name xdp-dns-cache -d program_type=xdp https://github.com/aya-rs/aya-template`.

A. *Overview of the file structure*

```
./
├── Cargo.toml
├── Makefile
├── xdp-dns-cache/
│   ├── Cargo.toml
│   └── src/
│       └── main.rs
├── xdp-dns-cache-common/
│   ├── Cargo.toml
│   └── src/
│       └── lib.rs
├── xdp-dns-cache-ebpf/
│   ├── Cargo.toml
│   ├── rust-toolchain.toml
│   └── src/
│       ├── csum.rs
│       ├── cursor.rs
│       ├── dns.rs
│       ├── helpers.rs
│       ├── main.rs
│       └── metadata.rs
└── xtask/
    ├── Cargo.toml
    └── src/
        ├── build_ebpf.rs
        ├── main.rs
        └── run.rs
```

Fig. 2: Overview of the project's file structure (excluding lock files)

B. *Makefile*

```
.PHONY: release debug run run-debug

release:
	cargo xtask build-ebpf --release
	cargo build --release

debug:
	RUSTFLAGS="--cfg include_info" cargo xtask build-ebpf
	RUSTFLAGS="--cfg include_info" cargo build

run: release
	sudo RUST_LOG=info target/release/xdp-dns-cache --iface lo

run-debug: debug
	sudo RUST_LOG=info target/debug/xdp-dns-cache --iface lo
```

Listing 2: Makefile

C. *xdp-dns-cache/src/main.rs*

```rust
use anyhow::Context;
use aya::maps::ProgramArray;
use aya::programs::{Xdp, XdpFlags};
use aya::{include_bytes_aligned, Bpf};
use aya_log::BpfLogger;
use clap::Parser;
use log::{info, warn, debug};
use tokio::signal;

#[derive(Debug, Parser)]
struct Opt {
    #[clap(short, long, default_value = "eth0")]
    iface: String,
}

#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    env_logger::init();

    // Bump the memlock rlimit. This is needed for older kernels that don't use the
    // new memcg based accounting, see https://lwn.net/Articles/837122/
    let rlim = libc::rlimit {
        rlim_cur: libc::RLIM_INFINITY,
        rlim_max: libc::RLIM_INFINITY,
    };
    let ret = unsafe { libc::setrlimit(libc::RLIMIT_MEMLOCK, &rlim) };
    if ret != 0 {
        debug!("remove limit on locked memory failed, ret is: {}", ret);
    }

    // This will include your eBPF object file as raw bytes at compile-time and load it at
    // runtime. This approach is recommended for most real-world use cases. If you would
    // like to specify the eBPF program at runtime rather than at compile-time, you can
    // reach for `Bpf::load_file` instead.
    #[cfg(debug_assertions)]
    let mut bpf = Bpf::load(include_bytes_aligned!(
        "../../target/bpfel-unknown-none/debug/xdp-dns-cache"
    ))?;
    #[cfg(not(debug_assertions))]
    let mut bpf = Bpf::load(include_bytes_aligned!(
        "../../target/bpfel-unknown-none/release/xdp-dns-cache"
    ))?;
    if let Err(e) = BpfLogger::init(&mut bpf) {
        // This can happen if you remove all log statements from your eBPF program.
        warn!("failed to initialize eBPF logger: {}", e);
    }

    let flags = 0;
    let mut prog_array = ProgramArray::try_from(bpf.take_map("JUMP_TABLE").unwrap())?;
    let prog_1: &mut Xdp = bpf.program_mut("xdp_parse_dname").unwrap().try_into()?;
    prog_1.load()?;
    prog_array.set(1, prog_1.fd().unwrap(), flags)?;

    let prog_2: &mut Xdp = bpf.program_mut("xdp_check_cache").unwrap().try_into()?;
    prog_2.load()?;
    prog_array.set(2, prog_2.fd().unwrap(), flags)?;

    let program: &mut Xdp = bpf.program_mut("xdp_dns_cache").unwrap().try_into()?;
    program.load()?;
    prog_array.set(0, program.fd().unwrap(), flags)?;

    program.attach(&opt.iface, XdpFlags::default())
        .context("failed to attach the XDP program with default flags - try changing XdpFlags::default() to
        ↪  XdpFlags::SKB_MODE")?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;
    info!("Exiting...");

    Ok(())
}
```

Listing 3: xdp-dns-cache/src/main.rs

D. xdp-dns-cache-ebpf/Cargo.toml

```toml
[package]
name = "xdp-dns-cache-ebpf"
version = "0.1.0"
edition = "2021"

[dependencies]
aya-bpf = { git = "https://github.com/aya-rs/aya" }
aya-log-ebpf = { git = "https://github.com/aya-rs/aya" }
xdp-dns-cache-common = { path = "../xdp-dns-cache-common" }
network-types = "0.0.5"
c2rust-bitfields = { version = "0.18.0", features = ["no_std"] }

[[bin]]
name = "xdp-dns-cache"
path = "src/main.rs"

[profile.dev]
opt-level = 3
debug = false
debug-assertions = false
overflow-checks = false
lto = true
panic = "abort"
incremental = false
codegen-units = 1
rpath = false

[profile.release]
lto = true
panic = "abort"
codegen-units = 1

[workspace]
members = []
```

Listing 4: xdp-dns-cache-ebpf/Cargo.toml

E. xdp-dns-cache-ebpf/src/main.rs

```rust
#![no_std]
#![no_main]

use core::mem;

mod dns;
use dns::*;
mod csum;
mod helpers;
use helpers::*;
mod cursor;
use cursor::Cursor;
mod metadata;
use metadata::*;

use aya_bpf::{
    bindings::xdp_action,
    helpers::*,
    macros::{map, xdp},
    maps::ProgramArray,
    programs::XdpContext,
};
use network_types::{
    eth::{EthHdr, EtherType},
    ip::{IpProto, Ipv4Hdr, Ipv6Hdr},
    udp::UdpHdr,
};

// make a simple wrapper around aya_log_ebpf::info to only include it if the cfg flag
// "include_info" is set: i.e. $ RUSTFLAGS="--cfg include_info" cargo xtask build-ebpf
// This is necessary as different kernels have different limits and including logging adds
// many many instructions to the resulting BPF bytecode. This allows disabling logging at
// compile-time without going through the code to delete every logging statement.
macro_rules! info {
    ($($arg:tt)*) => {
        #[cfg(include_info)]
        {
            aya_log_ebpf::info!($($arg)*);
        }
    };
}

const MAX_SENSIBLE_LABEL_COUNT: u8 = 20;
const CACHED_QNAME_SIZE: usize = 32;

#[map(name = "JUMP_TABLE")]
static mut JUMP_TABLE: ProgramArray = ProgramArray::with_max_entries(8, 0);

#[allow(dead_code)]
const XDP_DNS_CACHE: u32 = 0;
const XDP_PARSE_DNAME: u32 = 1;
const XDP_CHECK_CACHE: u32 = 2;

// answer for nl. NS IN including compression
// const ANSWER_LEN: usize = 58;
// const NSCOUNT: u16 = 3;
// const ARCOUNT: u16 = 0;
// const ANSWER_DATA: [u8; ANSWER_LEN] = [
//     0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00, 0x00, 0x0a, 0x03, 0x6e, 0x73, 0x31,
//     0x03, 0x64, 0x6e, 0x73, 0xc0, 0x0c, 0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00,
//     0x00, 0x06, 0x03, 0x6e, 0x73, 0x33, 0xc0, 0x24, 0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02,
//     0xa3, 0x00, 0x00, 0x06, 0x03, 0x6e, 0x73, 0x34, 0xc0, 0x24,
// ];

// answer for nl. NS IN including compression and glue
const ANSWER_LEN: usize = 201;
const NSCOUNT: u16 = 3;
const ARCOUNT: u16 = 7;
const ANSWER_DATA: [u8; ANSWER_LEN] = [
    0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00, 0x00, 0x0a, 0x03, 0x6e, 0x73, 0x31,
    0x03, 0x64, 0x6e, 0x73, 0xc0, 0x0c, 0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00,
    0x00, 0x06, 0x03, 0x6e, 0x73, 0x33, 0xc0, 0x24, 0xc0, 0x0c, 0x00, 0x02, 0x00, 0x01, 0x00, 0x02,
    0xa3, 0x00, 0x00, 0x06, 0x03, 0x6e, 0x73, 0x34, 0xc0, 0x24, 0xc0, 0x20, 0x00, 0x01, 0x00, 0x01,
    0x00, 0x02, 0xa3, 0x00, 0x00, 0x04, 0xc2, 0x00, 0x1c, 0x35, 0xc0, 0x36, 0x00, 0x01, 0x00, 0x01,
    0x00, 0x02, 0xa3, 0x00, 0x00, 0x04, 0xc2, 0x00, 0x19, 0x18, 0xc0, 0x48, 0x00, 0x01, 0x00, 0x01,
    0x00, 0x02, 0xa3, 0x00, 0x00, 0x04, 0xb9, 0x9f, 0xc7, 0xc8, 0xc0, 0x20, 0x00, 0x1c, 0x00, 0x01,
    0x00, 0x02, 0xa3, 0x00, 0x00, 0x10, 0x20, 0x01, 0x06, 0x78, 0x00, 0x2c, 0x00, 0x00, 0x01, 0x94,
    0x00, 0x00, 0x00, 0x28, 0x00, 0x53, 0xc0, 0x36, 0x00, 0x1c, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00,
    0x00, 0x10, 0x20, 0x01, 0x06, 0x78, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x24, 0xc0, 0x48, 0x00, 0x1c, 0x00, 0x01, 0x00, 0x02, 0xa3, 0x00, 0x00, 0x10, 0x26, 0x20,
    0x01, 0x0a, 0x80, 0xac, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00,
    0x29, 0x04, 0xd0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
];

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    unsafe { core::hint::unreachable_unchecked() }
}

#[xdp]
pub fn xdp_dns_cache(ctx: XdpContext) -> u32 {
    match try_xdp_dns_cache(ctx) {
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}

#[inline(always)]
fn try_xdp_dns_cache(ctx: XdpContext) -> Result<u32, ()> {
    let ethhdr: *mut EthHdr = ptr_at_mut(&ctx, 0)?;

    match unsafe { (*ethhdr).ether_type } {
        EtherType::Ipv4 => do_ipv4(ctx),
        EtherType::Ipv6 => do_ipv6(ctx),
        _ => Ok(xdp_action::XDP_PASS),
    }
}

#[inline(always)]
fn do_ipv6(ctx: XdpContext) -> Result<u32, ()> {
    Ok(xdp_action::XDP_PASS)
}

#[inline(always)]
fn do_ipv4(ctx: XdpContext) -> Result<u32, ()> {
    let ipv4hdr: *mut Ipv4Hdr = ptr_at_mut(&ctx, EthHdr::LEN)?;

    let source_addr = unsafe { u32::from_be((*ipv4hdr).src_addr) };

    match source_addr {
```

```rust
        // source == 127.0.0.2 || 10.1.1.1
        0x7f000002 | 0x0a010101 => {}
        _ => return Ok(xdp_action::XDP_PASS),
    }

    if is_udp_v4(ipv4hdr) {
        return do_udp(ctx, EthHdr::LEN + Ipv4Hdr::LEN);
    };

    Ok(xdp_action::XDP_PASS)
}

#[inline(always)]
fn is_udp_v4(ipv4hdr: *const Ipv4Hdr) -> bool {
    unsafe { (*ipv4hdr).proto == IpProto::Udp }
}

#[inline(always)]
fn is_udp_v6(ipv6hdr: *const Ipv6Hdr) -> bool {
    unsafe { (*ipv6hdr).next_hdr == IpProto::Udp }
}

#[inline(always)]
fn do_udp(ctx: XdpContext, header_offset: usize) -> Result<u32, ()> {
    let udphdr: *mut UdpHdr = ptr_at_mut(&ctx, header_offset)?;
    let dest_port = u16::from_be(unsafe { (*udphdr).dest });

    if dest_port == 53 {
        return do_dns(ctx, header_offset + UdpHdr::LEN);
    } else {
        return Ok(xdp_action::XDP_PASS);
    }
}

#[inline(always)]
fn do_dns(ctx: XdpContext, header_offset: usize) -> Result<u32, ()> {
    info!(&ctx, "do_dns");
    let dnshdr: *mut DnsHdr = ptr_at_mut(&ctx, header_offset)?;
    unsafe {
        info!(
            &ctx,
            "QR:{}, OPCODE:{}, AA:{}, TC:{}, RD:{}, RA:{}, Z:{}, AD:{}, CD:{}, RCODE:{}, QDCOUNT:{}, ANCOUNT:{},
            ↪  NSCOUNT:{}, ARCOUNT:{}",
            (*dnshdr).qr(),
            (*dnshdr).opcode(),
            (*dnshdr).aa(),
            (*dnshdr).tc(),
            (*dnshdr).rd(),
            (*dnshdr).ra(),
            (*dnshdr).z(),
            (*dnshdr).ad(),
            (*dnshdr).cd(),
            (*dnshdr).rcode(),
            (*dnshdr).qdcount(),
            (*dnshdr).ancount(),
            (*dnshdr).nscount(),
            (*dnshdr).arcount(),
        );
    }

    unsafe {
        let dnshdr: &DnsHdr = &*(dnshdr);
        if dnshdr.qr() != 0
            || dnshdr.qdcount() != 1
            || dnshdr.ancount() != 0
            || dnshdr.nscount() != 0
            || dnshdr.arcount() > 1
        {
            info!(&ctx, "Aborting this message, the DNS query is bogus");
            return Ok(xdp_action::XDP_ABORTED);
        }
    }

    unsafe {
        // WARNING: delta for adjust_meta must be <= 32 and a multiple of 4
        if bpf_xdp_adjust_meta(ctx.ctx, -(mem::size_of::<MetaData>() as i32)) != 0 {
            info!(&ctx, "Could not adjust metadata");
            return Ok(xdp_action::XDP_PASS);
        }
    }

    if ctx.metadata() + mem::size_of::<MetaData>() > ctx.metadata_end() {
        info!(&ctx, "Adjust metadata didn't work? The struct doesn't fit");
        return Ok(xdp_action::XDP_PASS);
    }

    let meta: *mut MetaData = ctx.metadata() as *mut MetaData;

    unsafe {
        (*meta).dname_offset = (header_offset + DnsHdr::LEN) as u8;

        if header_offset == EthHdr::LEN + Ipv4Hdr::LEN + UdpHdr::LEN {
            let ipv4hdr: *mut Ipv4Hdr = ptr_at_mut(&ctx, EthHdr::LEN)?;
            let udphdr: *mut UdpHdr = ptr_at_mut(&ctx, EthHdr::LEN + Ipv4Hdr::LEN)?;

            change_len_and_checksums_v4(&ctx, ipv4hdr, udphdr, ANSWER_LEN as u16)?;
        // } else if header_offset == EthHdr::LEN + Ipv6Hdr::LEN + UdpHdr::LEN {
        //     let ipv6hdr: *mut Ipv6Hdr = ptr_at_mut(&ctx, EthHdr::LEN)?;
        //     let udphdr: *mut UdpHdr = ptr_at_mut(&ctx, EthHdr::LEN + Ipv6Hdr::LEN)?;

        //     change_len_and_checksums_v6(&ctx, ipv6hdr, udphdr, ANSWER_LEN as u16)?;
        } else {
            return Err(());
        }

        let _ = JUMP_TABLE.tail_call(&ctx, XDP_PARSE_DNAME);
    }

    Ok(xdp_action::XDP_PASS)
}

#[xdp]
pub fn xdp_parse_dname(ctx: XdpContext) -> u32 {
    info!(&ctx, "Hello tailcall :)");
    if ctx.metadata() + 1 > ctx.data() {
        info!(&ctx, "there is no metadata available in xdp_parse_dname");
        return xdp_action::XDP_PASS;
    }

    let data_end = ctx.data_end();
    let metadata: &mut MetaData = unsafe { &mut *(ctx.metadata() as *mut MetaData) };

    if ctx.metadata() + mem::size_of::<MetaData>() > ctx.data() {
        info!(&ctx, "we goofed with the metadata");
        return xdp_action::XDP_PASS;
    }

    let proto: EtherType;
    let dnshdr: &mut DnsHdr;
    let dnsdata_off: usize;
    let v4_off = Ipv4Hdr::LEN + EthHdr::LEN + UdpHdr::LEN + DnsHdr::LEN;
    let v6_off = Ipv6Hdr::LEN + EthHdr::LEN + UdpHdr::LEN + DnsHdr::LEN;
    if metadata.dname_offset as usize == v4_off {
        proto = EtherType::Ipv4;
        dnsdata_off = v4_off;
        unsafe {
            dnshdr = &mut *((ctx.data() + EthHdr::LEN + Ipv4Hdr::LEN + UdpHdr::LEN) as *mut DnsHdr)
        }
    } else if metadata.dname_offset as usize == v6_off {
        proto = EtherType::Ipv6;
        dnsdata_off = v6_off;
        unsafe {
            dnshdr = &mut *((ctx.data() + EthHdr::LEN + Ipv6Hdr::LEN + UdpHdr::LEN) as *mut DnsHdr)
        }
    } else {
        info!(&ctx, "ether_type doesn't match");
        return xdp_action::XDP_PASS;
    }

    let mut cursor: Cursor = Cursor::new(ctx.data() + dnsdata_off);
    let mut buf: [u8; CACHED_QNAME_SIZE] = [0; CACHED_QNAME_SIZE];

    // if dns query is at least X bytes long
    // let len = 5; // .
    // // let len = 7; // no need to check for single character long TLDs (they don't exist)
    // let len = 8; // nl. / de. / ...
    // let len = 9; // com. / ...
    // let len = 10; // name. / ...
    // NOTE: we might be able to use a for loop (unrolled possibly) to reduce the amount of code
    // TODO: maybe ignore the need for class and type in the bounds/numbers below?
    if ctx.data() + dnsdata_off + 18 < ctx.data_end() {
        // at least a query to nlnetlabs.nl. fits
        if let Err(action) = parse_qname(&ctx, 14, &mut buf, &mut cursor) {
            return action;
        }
    } else if ctx.data() + dnsdata_off + 10 < ctx.data_end() {
        // at least a query to name. fits
        if let Err(action) = parse_qname(&ctx, 6, &mut buf, &mut cursor) {
            return action;
        }
    } else if ctx.data() + dnsdata_off + 9 < ctx.data_end() {
        // at least a query to com. fits
        if let Err(action) = parse_qname(&ctx, 5, &mut buf, &mut cursor) {
            return action;
        }
    } else if ctx.data() + dnsdata_off + 8 < ctx.data_end() {
        // at least a query to nl. fits
        if let Err(action) = parse_qname(&ctx, 4, &mut buf, &mut cursor) {
            return action;
        }
    } else if ctx.data() + dnsdata_off + 5 < ctx.data_end() {
        // at least a query to . fits
        if let Err(action) = parse_qname(&ctx, 1, &mut buf, &mut cursor) {
```

```rust
            return action;
        }
    } else {
        info!(&ctx, "dns query not long enough");
        return xdp_action::XDP_ABORTED;
    }

    unsafe {
        let s = core::str::from_utf8_unchecked(&buf);
        info!(&ctx, "buf (unchecked utf8): {}", s);
    }

    if cursor.pos + 2 > ctx.data_end() {
        info!(&ctx, "dns query not long enough");
        return xdp_action::XDP_ABORTED;
    }

    let q_type: u16 = u16::from_be(unsafe { *(cursor.pos as *const u16) });
    cursor.pos += 2;

    if cursor.pos + 2 > ctx.data_end() {
        info!(&ctx, "dns query not long enough");
        return xdp_action::XDP_ABORTED;
    }

    let q_class: u16 = u16::from_be(unsafe { *(cursor.pos as *const u16) });
    cursor.pos += 2;

    info!(&ctx, "q_type: {}, class: {}", q_type, q_class);

    if buf[0..4] == [0x02, 0x6e, 0x6c, 0x00] {
        info!(&ctx, "yes it's for nl.");

        match proto {
            EtherType::Ipv6 => {
                // if let Ok(ipv6hdr) = ptr_at_mut(&ctx, EthHdr::LEN) {
                //     swap_ipv6_addr(ipv6hdr);
                // }

                // if let Ok(udphdr) = ptr_at_mut(&ctx, EthHdr::LEN + Ipv6Hdr::LEN) {
                //     swap_udp_ports(udphdr);
                // }
            }
            EtherType::Ipv4 => {
                if let Ok(ipv4hdr) = ptr_at_mut(&ctx, EthHdr::LEN) {
                    swap_ipv4_addr(ipv4hdr);
                }

                if let Ok(udphdr) = ptr_at_mut(&ctx, EthHdr::LEN + Ipv4Hdr::LEN) {
                    swap_udp_ports(udphdr);
                }
            }
            _ => return xdp_action::XDP_PASS,
        }

        if let Ok(ethhdr) = ptr_at_mut(&ctx, 0) {
            swap_eth_addr(ethhdr);
        }

        // dnshdr.set_tc(1);
        dnshdr.set_qr(1);
        dnshdr.set_ra(0);
        dnshdr.set_nscount(NSCOUNT);
        dnshdr.set_arcount(ARCOUNT);

        // ignore EDNS0 and just overwrite with answer
        if cursor.pos + ANSWER_LEN < ctx.data_end() {
            for i in 0..ANSWER_LEN {
                unsafe {
                    *(cursor.pos as *mut u8) = ANSWER_DATA[i];
                    cursor.pos += 1;
                }
            }

            // nullify the rest
            for _i in 0..100 {
                if cursor.pos < ctx.data_end() {
                    unsafe {
                        *(cursor.pos as *mut u8) = 0;
                        cursor.pos += 1;
                    }
                }
            }
        }
        return xdp_action::XDP_TX;
    }

    unsafe {
        let _ = JUMP_TABLE.tail_call(&ctx, XDP_CHECK_CACHE);
        info!(&ctx, "tail call failed");
    }
    xdp_action::XDP_PASS
}

#[xdp]
pub fn xdp_check_cache(ctx: XdpContext) -> u32 {
    info!(&ctx, "Hello second tailcall :)");
    xdp_action::XDP_PASS
}

#[inline(always)]
fn parse_qname(
    ctx: &XdpContext,
    max_bytes: usize,
    buf: &mut [u8],
    cursor: &mut Cursor,
) -> Result<(), u32> {
    let mut buf_index = 0;
    let mut label_bytes_left = 0;
    let mut reached_root_label = false;
    for _i in 0..=max_bytes {
        let char: u8 = unsafe { *(cursor.pos as *const u8) };
        info!(ctx, "{}", char);
        cursor.pos += 1;
        if char == 0 {
            info!(ctx, "reached root label");
            reached_root_label = true;
            break;
        }

        if label_bytes_left == 0 {
            if (char & 0xC0) == 0xC0 {
                info!(ctx, "complabel");
                // compression label; not checking validity of reference
                // compression label would be the last label of dname
                break;
            } else if (char & 0xC0) != 0 {
                info!(ctx, "unknown label");
                return Err(xdp_action::XDP_PASS);
            } else {
                info!(ctx, "label len: {}", char);
                label_bytes_left = char + 1; // +1 because of length itself
            }
        }

        buf[buf_index] = char;
        buf_index += 1;
        label_bytes_left -= 1;
    }

    info!(ctx, "chars left: {}", label_bytes_left);
    if !reached_root_label {
        info!(ctx, "qname was not read appropriately");
        return Err(xdp_action::XDP_PASS);
    }

    Ok(())
}
```

Listing 5: xdp-dns-cache-ebpf/src/main.rs

F. xdp-dns-cache-ebpf/src/dns.rs

```rust
#![allow(dead_code)]
use c2rust_bitfields::BitfieldStruct;

pub const DNS_PORT: u16 = 53;
pub const RR_TYPE_OPT: u16 = 41;
pub const RCODE_REFUSED: u8 = 5;

#[repr(C, align(1))]
#[derive(BitfieldStruct)]
pub struct DnsHdr {
    pub id: u16,
    #[bitfield(name = "qr", ty = "u8", bits = "7..=7")]
    #[bitfield(name = "opcode", ty = "u8", bits = "3..=6")]
    #[bitfield(name = "aa", ty = "u8", bits = "2..=2")]
    #[bitfield(name = "tc", ty = "u8", bits = "1..=1")]
    #[bitfield(name = "rd", ty = "u8", bits = "0..=0")]
    #[bitfield(name = "ra", ty = "u8", bits = "15..=15")]
    #[bitfield(name = "z", ty = "u8", bits = "14..=14")]
    #[bitfield(name = "ad", ty = "u8", bits = "13..=13")]
    #[bitfield(name = "cd", ty = "u8", bits = "12..=12")]
    #[bitfield(name = "rcode", ty = "u8", bits = "8..=11")]
    codes_and_flags: [u8; 2],
    pub qdcount: u16,
    pub ancount: u16,
    pub nscount: u16,
    pub arcount: u16,
}

impl DnsHdr {
    pub const LEN: usize = core::mem::size_of::<DnsHdr>();

    pub fn id(&self) -> u16 {
        u16::from_be(self.id)
    }

    pub fn qdcount(&self) -> u16 {
        u16::from_be(self.qdcount)
    }

    pub fn ancount(&self) -> u16 {
        u16::from_be(self.ancount)
    }

    pub fn nscount(&self) -> u16 {
        u16::from_be(self.nscount)
    }

    pub fn arcount(&self) -> u16 {
        u16::from_be(self.arcount)
    }

    pub fn set_id(&mut self, id: u16) {
        self.id = u16::to_be(id)
    }

    pub fn set_qdcount(&mut self, count: u16) {
        self.qdcount = u16::to_be(count)
    }

    pub fn set_ancount(&mut self, count: u16) {
        self.ancount = u16::to_be(count)
    }

    pub fn set_nscount(&mut self, count: u16) {
        self.nscount = u16::to_be(count)
    }

    pub fn set_arcount(&mut self, count: u16) {
        self.arcount = u16::to_be(count)
    }
}
```

Listing 6: xdp-dns-cache-ebpf/src/dns.rs

G. xdp-dns-cache-ebpf/src/metadata.rs

```rust
// This struct needs to be a multiple of 4 bytes in size and at max 32 bytes in size
// Check in kernel code in file include/net/xdp.h:xdp_metalen_invalid
#[repr(C)]
pub struct MetaData {
    pub dname_offset: u8,
    pub lbl1_offset: u8,
    pub lbl2_offset: u8,
    pub lbl3_offset: u8,
}
```

Listing 7: xdp-dns-cache-ebpf/src/metadata.rs

H. xdp-dns-cache-ebpf/src/helpers.rs

```rust
use aya_bpf::{bindings::xdp_action, helpers::bpf_xdp_adjust_tail, programs::XdpContext};
use aya_log_ebpf::info;
use core::mem;
use network_types::{eth::EthHdr, ip::Ipv4Hdr, udp::UdpHdr};

use crate::csum::*;

#[inline(always)]
pub fn swap_udp_ports(udphdr: *mut UdpHdr) {
    unsafe {
        let src_port_be = (*udphdr).source;
        (*udphdr).source = (*udphdr).dest;
        (*udphdr).dest = src_port_be;
    }
}

#[inline(always)]
pub fn swap_eth_addr(ethhdr: *mut EthHdr) {
    unsafe {
        let src_addr_be = (*ethhdr).src_addr;
        (*ethhdr).src_addr = (*ethhdr).dst_addr;
        (*ethhdr).dst_addr = src_addr_be;
    }
}

#[inline(always)]
pub fn swap_ipv4_addr(ipv4hdr: *mut Ipv4Hdr) {
    unsafe {
        let src_addr_be = (*ipv4hdr).src_addr;
        (*ipv4hdr).src_addr = (*ipv4hdr).dst_addr;
        (*ipv4hdr).dst_addr = src_addr_be;
    };
}

#[allow(dead_code)]
#[inline(always)]
pub fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = mem::size_of::<T>();

    if start + offset + len > end {
        return Err(());
    }

    Ok((start + offset) as *const T)
}

#[inline(always)]
pub fn ptr_at_mut<T>(ctx: &XdpContext, offset: usize) -> Result<*mut T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = mem::size_of::<T>();

    if start + offset + len > end {
        return Err(());
    }

    Ok((start + offset) as *mut T)
}

#[allow(dead_code)]
#[inline(always)]
pub fn csum_replace_u32(mut check: u16, old: u32, new: u32) -> u16 {
    check = csum_replace(check, (old >> 16) as u16, (new >> 16) as u16);
    check = csum_replace(check, (old & 0xffff) as u16, (new & 0xffff) as u16);
    check
}

#[inline(always)]
pub fn change_len_and_checksums_v4(
    ctx: &XdpContext,
    ipv4hdr: *mut Ipv4Hdr,
    udphdr: *mut UdpHdr,
    delta: u16,
) -> Result<u32, ()> {
    let orig_frame_size = ctx.data_end() - ctx.data();

    let orig_ipv4_len = u16::from_be(unsafe { (*ipv4hdr).tot_len });
    let ipv4_len_new = orig_ipv4_len + delta;
    let mut csum = u16::from_be(unsafe { (*ipv4hdr).check });

    csum = csum_replace(csum, orig_ipv4_len, ipv4_len_new);

    unsafe {
        info!(
            ctx,
            "ctx.len: {} + delta = {} || ipv4 len before: {}, ipv4 len after: {}, delta: {}",
            orig_frame_size,
            orig_frame_size + delta as usize,
            orig_ipv4_len,
            ipv4_len_new,
            delta
        );
        (*ipv4hdr).tot_len = u16::to_be(ipv4_len_new);
        (*ipv4hdr).check = u16::to_be(csum);
    }

    unsafe {
        (*udphdr).len = u16::to_be(u16::from_be((*udphdr).len) + delta);
        (*udphdr).check = 0;
    }

    // using adjust_tail invalidates all boundschecks priviously done, so this
    // has to go below the address swaps
    if unsafe { bpf_xdp_adjust_tail(ctx.ctx, delta.into()) } != 0 {
        info!(ctx, "adjust_tail failed for tail delta: {}", delta);
    }

    Ok(xdp_action::XDP_PASS)
}
```

Listing 8: xdp-dns-cache-ebpf/src/helpers.rs

I. *xdp-dns-cache-ebpf/src/csum.rs*

```rust
/******************************************************************************
* Title: XDP Tutorial
* Author: Eelco Chaudron
* Date: 2019-08-16
* Availability: https://github.com/xdp-project/xdp-tutorial
* ***************************************************************************/
// Inspired by xdp-tutorial:advanced03-AF_XDP/af_xdp_user.c
// The algorithm can also be found in RFC 1624.

#[inline(always)]
pub fn csum16_add(csum: u16, addend: u16) -> u16 {
    let res: u16 = csum;
    let res = res.wrapping_add(addend);
    if res < addend {
        res + 1
    } else {
        res
    }
}

#[inline(always)]
pub fn csum16_sub(csum: u16, addend: u16) -> u16 {
    csum16_add(csum, !addend)
}

#[inline(always)]
pub fn csum_replace(check: u16, old: u16, new: u16) -> u16 {
    !csum16_add(csum16_sub(!check, old), new)
}
```

Listing 9: xdp-dns-cache-ebpf/src/csum.rs

J. *xdp-dns-cache-ebpf/src/cursor.rs*

```rust
pub struct Cursor {
    pub pos: usize,
}

impl Cursor {
    pub fn new(pos: usize) -> Self {
        Self { pos }
    }
}
```

Listing 10: xdp-dns-cache-ebpf/src/cursor.rs