Exploring network protocol state access for eBPF

CS 5204 Operating Systems Final Project Report

Sabiha Afroz, Syed Tauhidun Nabi, Kaiyi Kenny Chen, Md Sayeedul Islam, Wesley Woo

COLLEGE OF ENGINEERING
COMPUTER SCIENCE
VIRGINIA TECH

1. Background and Motivation

Large-scale internet service providers always look for safe communication with low latency and high throughput, which in-memory key-value stores can enable. Memcached, a popular kernel caching solution used in data centers, suffers from performance limitations while using high-speed network interfaces. To overcome this problem, kernel bypass could be a solution, but bypassing the kernel causes other limitations, like losing program isolation and inefficient CPU utilization. Considering all these mentioned points, a group of researchers designed and implemented a eBPF cache system for Memcached, also known as BMC.

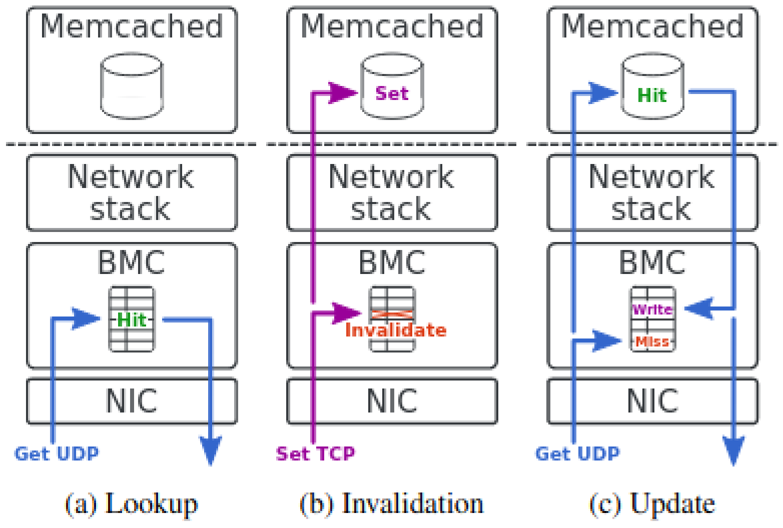

Figure 1: BMC Cache operations

1.1 BMC: Accelerating Memcached with in-kernel caching

Authors Yoann Ghigoff et al., in their BMC paper [1], accelerated the overall performance of Memcached by creating a eBPF caching program. The implemented cache handles GET and SET requests from application clients. GET requests can be sent using UDP or TCP protocol; SET requests are sent via TCP protocol as they require guaranteed delivery. On receiving GET and SET requests, the eBPF program queries the cache and updates/returns the appropriate stored state. This program is attached to the XDP hook, which sits at the NIC driver and is the earliest possible eBPF hook in the network data path. Overall, BMC improves the performance of Memcached through a crash-safe cache extension of the Linux kernel -

- Throughput rises up to 18x compared to vanilla Memcached and 6x compared to MemcachedSR
- Without running BMC, the Memcached operations are about 1 μs faster. So almost negligible latency overhead
- Even though after increasing the workload, similar performance was maintained, as claimed by the authors

1.2 eBPF

eBPF is Linux's extended version of the Berkeley Packet Filter (BPF) which enables user-programmed operation inside the kernel. eBPF offers safety through static function, meets JIT compliance, and attached network driver points or XDP hook. In addition, every eBPF program can operate independently with packet filtering, forwarding, tracing, and network scheduling. However, eBPF static analysis doesn't allow complex application logic to scale. It's necessary to keep loops in a fixed boundary, and no dynamic memory allocation is another limitation of the eBPF program.

1.3 The XDP hook

The eXpress Data Path or XDP hook is essential for attaching eBPF programs to the Linux kernel early on the networking stack. The XDP hook is an excellent resource for achieving high performance in eBPF programs as it even further reduces the overall time spent in the kernel. On top of that, XDP supports eBPF for running independent network operations, as mentioned earlier.

1.4 Our proposed work

Inspired by the BMC paper [1] we proposed an extension of the BMC so that it can process GET requests using any protocol (UDP or TCP). However, our motivation is not to make BMC more efficient in terms of performance but to push BMC to the boundary of what is achievable with eBPF. So in summary, The goal of this project is to extend the BMC paper by exploring the differences between eBPF servers that handle UDP versus TCP.

2. What we built

For this project, we built ICMP servers at the XDP and TC (traffic control) hook points, as well as a DNS server at the XDP hook point.

2.1 ICMP echo servers

An ICMP Echo server is a server that uses the ICMP protocol to send a lightweight ICMP response to a client's basic ICMP ping message. ICMP (Internet Control Message Protocol) is a protocol used by network devices such as routers to send error messages and operational information. As the IP protocol does not have any inbuilt mechanism for sending error and control messages, ICMP is used as a supporting protocol in the Internet protocol suite. The ping utility of ICMP protocol is used for network diagnosis. A ping will test the reachability and connection speed between two devices. Therefore, a user can measure the latency between two devices. So, ICMP echo request and echo response messages are used to perform a ping.

2.2 XDP ICMP echo server

Our XDP ICMP echo server is attached to the XDP hook of the kernel network driver to reply back with echo messages. When a packet is received by the network driver it triggers the ICMP echo server program attached to the XDP hook. The program then checks the ethernet and IP headers. Before checking each header information, it first ensures the memory boundary is consistent. If it finds something it passes the packet through the network stack. In the IP header, the program checks for ICMP packet type. If it is not an ICMP packet, the packet just passes through the network stack without any further manipulation. On the other hand, if it is an ICMP echo request, then the program changes the type to 0 (0 - echo response) of the ICMP header and recalculates the checksum. Later the header source and destination addresses are both swapped for IP address and port. After all the modification and reconstruction, the packet is sent through the same interface.

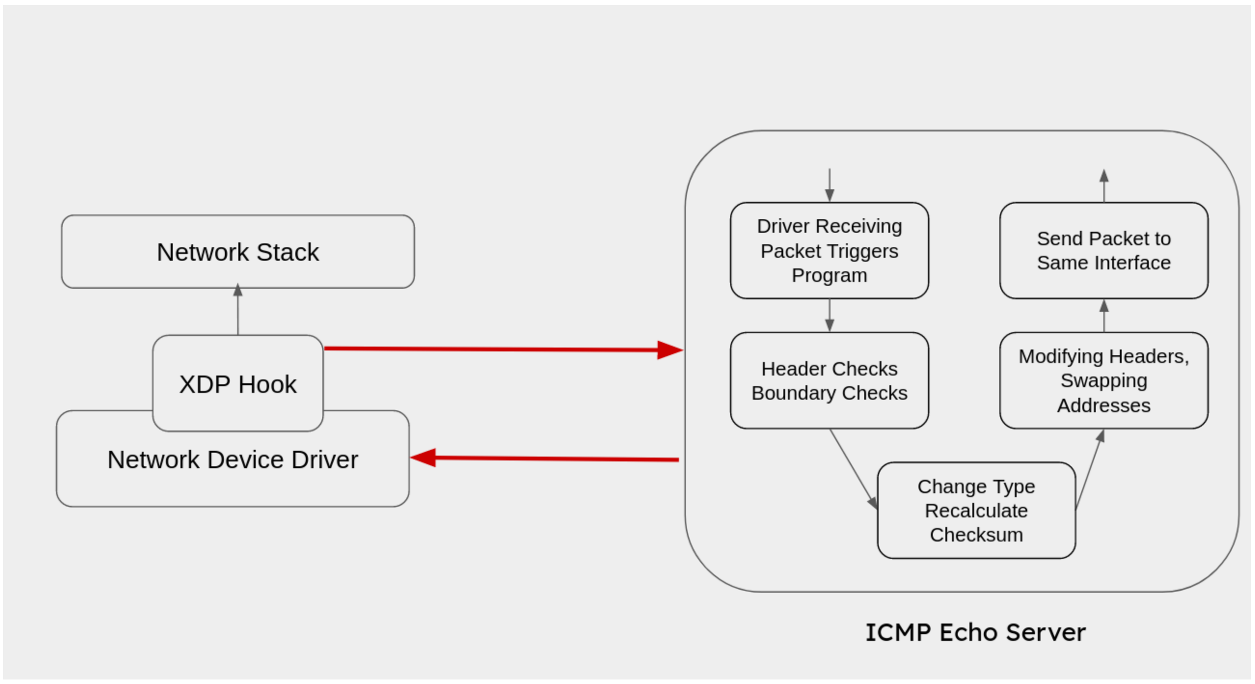
Network Stack
XDP Hook
Network Device Driver
Driver Receiving Packet Triggers Program
Header Checks Boundary Checks
Change Type Recalculate Checksum
Modifying Headers, Swapping Addresses
Send Packet to Same Interface
ICMP Echo Server

Figure 2: ICMP Echo Server implemented at the XDP hook

2.3 TC (Traffic Control) ICMP echo server

Our TC ICMP echo server is attached to the TC (Traffic Control) hook to compare the performance of the XDP ICMP echo server. TC echo server [2] is taken and modified for our project requirement and environment setup. The implementation of the TC ICMP server is almost similar to the XDP ICMP echo server. The primary difference is this ICMP echo server is attached to the TC hook, and the TC hook is located in the upper position of the network stack in comparison to the XDP hook location.

2.4 DNS Server

DNS (Domain Name System) Server can be thought of as a phonebook for the internet. In simple terms, a DNS server contains the IP addresses of human-readable web addresses. The DNS client, for example, our desktop, laptop etc. contacts the server to get the IP addresses of websites we want to visit. In the real world, the workflow is more complex, as one server cannot possibly contain the addresses of all the websites of the world. If a user enters “example.com” in their browser, a DNS client sends a request containing “example.com” to the DNS resolver. The resolver queries the root server first, the address of which is known to all resolvers. The root server gives the address of the “com” domain server. Upon querying the “com” domain server, the resolver gets the address of “example.com”. Although a real world DNS server is highly robust and can support any internet protocol, UDP is the preferred protocol for DNS resolution, because of its high speed and low overhead [4].

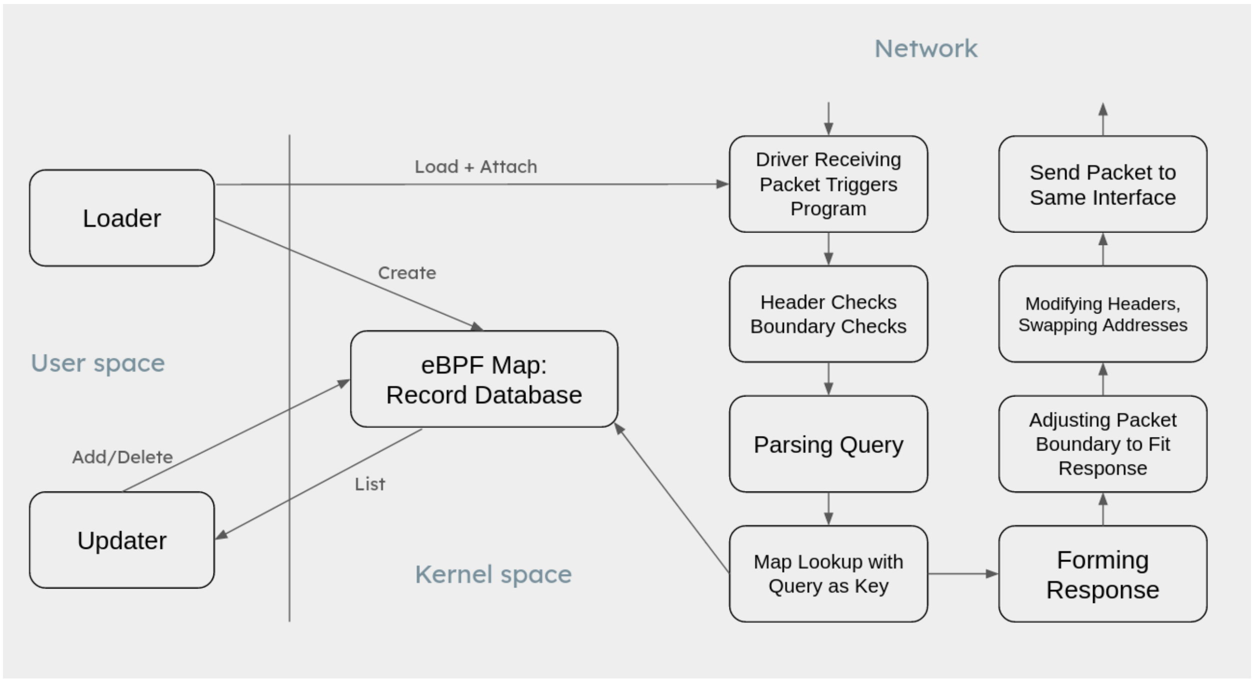
Network
Loader
Load + Attach
Create
User space
eBPF Map: Record Database
Add/Delete
List
Updater
Kernel space
Driver Receiving Packet Triggers Program
Header Checks Boundary Checks
Parsing Query
Map Lookup with Query as Key
Forming Response
Adjusting Packet Boundary to Fit Response
Modifying Headers, Swapping Addresses
Send Packet to Same Interface

Figure 3: DNS Server implemented at the XDP hook

2.5 DNS Server using eBPF at XDP hook

As the ICMP server we built was a simple application, implementing a DNS server using eBPF was the next step in exploring the boundaries of eBPF. Our work is an extension of [3], which is a very limited version of the DNS server. The groundwork was to move and modify the eBPF program to use the libbpf library that we used to build the ICMP server and to run in our test environment. Even after our extension, the server has the following limitations: (1) It supports only A (IPv4) and AAAA (IPv6) records, (2) It supports queries over UDP port 53 only, and (3) It supports a single query at a time and doesn’t support recursive lookups.

Our implementation is a simplified DNS server, which can be viewed as a simple lookup table for address translation. The program is attached at the XDP hook which is at the driver level of the network stack. The program is triggered upon the arrival of a packet. The program first checks the packet boundaries and the headers to verify that the packet is intended for a DNS resolution over UDP port 53. Then it parses the query to get the record name, type, and class. According to the type, it looks up in the appropriate eBPF map to get the record. For each type of record, a map is dedicated and those are updated by an updater in the userspace. Upon successful lookups, the program forms the response and appends at the end of the received packet. Finally, it modifies the headers and swaps the addresses to reflect the packet back to where it came from. Thus the program resolves the DNS query, without going through the whole network stack.

3. What we measured

3.1 ICMP Server measurement

We built two ICMP servers – one at the XDP hook and one at the TC hook. Originally, we built two because part of our group did not have the XDP hook on their virtual machine setups. But once we built these two servers, we compared the performance of the two implementations. We sent a ping request using the Linux ping function, and obtained the response time by comparing the statistics that the function returned.

3.1.1 ICMP Server results comparison

Hook Location / Measurement	**XDP hook**	**TC hook**	**Default implementation**
RTT min (ms)	0.175	0.158	0.158
RTT average (ms)	0.378	0.410	0.425
RTT max (ms)	0.552	0.871	0.750
RTT dev (ms)	0.093	0.111	0.113

Our results are not ideal. From an average point of view, XDP hook and TC hook appear to be more time-saving than normal. However, the XDP hook is longer than the default implementation in terms of minimum response time. At the same time, TC hook is longer than the default implementation in terms of maximum response time. The main reason is probably because other functions could be scheduled and ping responses take very little, which results in fluctuation of response time in the data we measured.

3.2 DNS Server measurement

3.2.1 Results

We built two DNS servers. One is an XDP hooked DNS server and the other is a user-space DNS server, which is the normal DNS Server implementation. We use these two DNS Servers for address resolution, and send our address to DNS Server through the dig command while measuring the timing of the results received from the DNS Server. During the experiment, we found that print statements within eBPF drastically increase the response time of the XDP DNS Server, so we first removed all print methods for testing end-to-end DNS responses. We added prints back in to time the performance of solely the eBPF function that is called. Below, the average times required for the DNS Server function call and the median of the end-to-end DNS response time is recorded.

Hook Location / Measurement	**XDP hooked DNS Server**	**User-space DNS Server**
Normal operation	19.267740 ms	20.243687 ms
Only function call	0.204424 ms	0.882204 ms

According to our results, we can clearly find that the average time required for XDP hook DNS server is shorter than the average time required for user-space DNS servers. But when we added a time function to the DNS side and returned the time required for the DNS Server to process the response, we found that the user-space DNS server spent more time processing the DNS address than the XDP hook DNS server. Apart from this, normal operation does not have much time difference.

The reason why our DNS server function call time is different is that we use different programming languages and library support. User-space DNS Server function calls use a library called Dnslib in python. We add DNS records to Dnslib in advance. When we send the response we receive to Dnslib, Dnslib will use the DNS records we imported before.The time required by our DNS server may be increased by the use of the library. XDP hooked DNS Server is more direct. It is written in C language. By storing our DNS records in the cache, we search them one by one through the for loop and return the results. This simpler approach does save a lot of time.

We found that there is no significant time gap between using the normal port and XDP hooked, which is similar to the results of our ICMP Server measurement. Ebpf didn't help us save too much time. It is very possible that this time includes the time required to send and receive with the server segment, but we have not found an effective way to measure it. This is where we can improve in the future. All in all, XDP hooked takes less time than the normal port, but we're not sure by how much.

4. Key results

4.1 eBPF programming challenges

Writing eBPF programs can be more difficult than other forms of kernel programming because the verifier must approve all code attached to a hook. We had to resolve the following verifier issues as we wrote our program:

1. Extra Boundary Checks

Before each memory access, the validity of the access must be ensured. For example, we had to check packet lengths to confirm that a location we try to access is guaranteed to be in the packet.

2. Bounded Loops

We need to include extra static conditions for loops in order to guarantee termination. For example, to iterate through every item in the list, we could not iterate until the len(list) – instead, we had to pre-define some maximum list size.

3. Mandatory Initialization

Some variables like map keys cannot be uninitialized. Additionally, if context is changed, all memory pointers need to be re-initialized.

4. eBPF Stack Size is only 512 Bytes

Because of limited stack size, there is a limit on how many variables can be declared. To get around this, we worked mainly with pointers. Similarly, too many nested function calls also cause problems.

5. Debugging

Verifier rejection messages are not straightforward to debug. Furthermore, the eBPF community is still relatively new and growing, so it can be difficult to find help online. Also, too many print statements in an eBPF program can overflow the eBPF stack.

4.2 Tradeoffs between XDP and TC hooks of the network stack

As these two hooks are located in different places of the kernel network stack, there are some benefits and drawbacks when we attach eBPF programs in these two hooks. Specifically, we see a tradeoff between performance and flexibility between the XDP and TCP hook points.

XDP hook is located near the network driver. Therefore, eBPF programs connected to this hook get triggered whenever a packet reaches the network driver. So, theoretically the performance is faster. We know that eBPF programs communicate with the kernel using helper functions. So, helper functions act as an interface for these programs. Many helper function facilities are absent from the XDP hook such as `bpf_l4_csum_replace` helper function. Hence, all of these cases are handled manually in programming which is cumbersome.

TC hook is located in the upper position compared to XDP hook. So, eBPF programs attached to this hook give slower performance. In addition to that, this hook is rich with varieties of helper functions. Therefore, programming eBPF applications is much easier and flexible.

4.3 eBPF Maps: passing state in eBPF programming

In addition to exploring the state available at the XDP hook point, we looked at how state can be passed in between eBPF programs and user space through eBPF maps. eBPF maps operate with the lookup, update, and delete primitives. Both user-space and kernel-space (eBPF) programs can change the state of these shared maps and identified through file descriptors.

While defining a map in eBPF, the programmer must specify a map type from a given enumeration, shown below:

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PROG_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    BPF_MAP_TYPE_STACK_TRACE,
    BPF_MAP_TYPE_CGROUP_ARRAY,
    BPF_MAP_TYPE_LRU_HASH,
    BPF_MAP_TYPE_LRU_PERCPU_HASH,
};
```

In order, these types are: hash maps (programmer can define the key/value types), arrays (key is a 32 bit integer; value is flexible), program arrays (array with program file descriptors as values), perf event arrays (map that can be used to receive events from eBPF programs using the linux perf API), per CPU hash maps and arrays (maps and arrays that are initialized per CPU, and thus don’t use locks), stack trace maps (used for storing stack traces), cgroup arrays (used for providing local storage at the cgroup the program is attached to), and LRU hashes/per CPU hashes (used for caches – these maps will remove the least recently used items when full).

In addition to map type, eBPF programmers must specify a maximum number of map entries and a set of map flags. Key and value sizes must be specified if not already set by map type.

These maps can be accessed through user-space and eBPF hook points via API. During this project, we did not find a way to use these maps to pass information about kernel networking stack state (TCP state) to different eBPF programs. However, there are many eBPF hook points, and a direction for future work could be exploring whether there are hooks that can access TCP state, and thus pass this state to different points in the kernel (i.e. to the XDP hook) via maps.

4.4 eBPF and the future of transport protocols

From our work, we find that it may be prohibitively difficult to write networked servers with eBPF that rely on stateful protocols. As discussed in the previous section, maps are the primary way to pass state to and from eBPF programs. Protocols such as TCP require maintaining state during packet processing, which we could not easily achieve in this project.

However, eBPF has promising use cases in the future of the Internet as protocols are moving towards user-space development. The best example of this is the QUIC transport protocol, which is rapidly becoming adopted. QUIC is an end-to-end encrypted protocol built on top of UDP and developed in user space. QUIC was designed to be end-to-end encrypted to avoid “implementation entrenchment”. The designers specifically reference TCP as an example protocol that has fallen victim to implementation entrenchment: because so many OS accelerate TCP processing via kernel programs such as DPDK, any updates to the TCP protocol also require mass upgrades to operating systems that implement such programs. Because QUIC is end-to-end encrypted and developed in user space, it is impossible for the transport protocol functions to be offloaded to the kernel. As such, QUIC applications will definitely benefit from eBPF programming! Any application state that will need to be accessed by a eBPF program will be guaranteed to exist in user space.

Additionally, the HTTP/3 protocol is built around QUIC, and so we expect adoption of QUIC to grow significantly in the coming years. More generally, the return to end-to-end principles that QUIC strives towards looks to be a growing movement that will motivate the design of other future transport protocols.

5. Retrospective

5.1 What we learned & what went well

From this project, we got a lot of hands-on experience working with eBPF tools and in the environment. We also learned a lot about the types of states accessible from eBPF programs, and how states can be passed from the kernel to user space with eBPF.

This project also gave us a hands-on understanding of “implementation entrenchment”, a problem that faces a lot of transport protocol developers. Through this, we explored the future of network transport protocols and the ways that researchers are looking to overcome this entrenchment.

5.2 What didn’t go well

A major barrier to our progress was setting up our group’s development environment. First, we tried to run the eBPF programs from inside our ORACLE virtual machine. However, the virtual machine network interfaces did not have the XDP driver for our project. As a result, we could not hook our ICMP eBPF program inside the kernel network driver. We followed some StackOverflow solutions and tried to install XDP-compatible drivers, but could not figure it out. To resolve this problem, we moved our working direction towards QEMU Linux. For some members of our group, this meant installing the QEMU Linux inside their ORACLE VM.

Additionally, while we looked around for UDP-based protocols to write a server for, we found that a lot of the work has been done already, which actually made sense – a lot of this work just required innovation and not any additional insight into system design. This was unfortunate because the only work we were really qualified to do was more basic implementation, as we were all relatively new to eBPF. This seems like an important lesson about systems research in general.

5.3 Major Challenges

eBPF Compiler

As userspace code is run inside the kernel, the kernel relies on the eBPF verifier to make sure the code is secure and safe for the system. However, this verifier imposes some restrictions on code developers. For example, the verifier sets instruction limits for an eBPF program. If it crosses the threshold, the verifier rejects the program. The verifier also confirms that an eBPF program terminates. A verifier also checks out-of-memory bound access for memory access instructions and eBPF only allows maps for holding shared/persistent data. All of these issues increase the complexity of eBPF programming.

eBPF Documentation

eBPF is an emerging research direction in the kernel research community. So, the eBPF programming documentation is not well structured, and the online resources are scattered and scarce for the researchers. This was part of the reason that it took us so long to figure out our development environment.

6. References

[1] Ghigoff, Yoann, Julien Sopena, Kahina Lazri, Antoine Blin and Gilles Muller. “BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing.” Symposium on Networked Systems Design and Implementation (2021).

[2] ICMP server at TC hook, URL: https://github.com/badboy/ebpf-icmp-ping

[3] Experimental DNS Server, URL: https://github.com/zebaz/xpress-dns

[4] RFC 1035, URL: https://www.rfc-editor.org/rfc/rfc1035#section-4.2