Mastering eBPF: creating a high performance ad-blocker

Maxime Chevallier, Alexis Lothoré

Capitole du Libre 2025

Your trainers for today's lab

- Maxime Chevallier and Alexis Lothoré
- Linux engineers and trainers @ *Bootlin* during the day
  - Engineering company specialized in **Embedded Linux** and **Zephyr**
  - 27 people, mostly Toulouse and Lyon
  - Engineering services
  - Training services
  - Very strong open-source focus
  - We are **hiring**, including **interns**
- Hackers at night

About this workshop

- you will learn how to create your own eBPF programs/tools
- you will learn about UDP and DNS
- two parts:
  - eBPF 101 for ~40min
  - hands-on labs during ~1h20
- at any moment, feel free to ask any question !
- those slides are available at:

https://bootlin.com/pub/conferences/2025/cdl/ebpf-workshop.pdf

Requirements

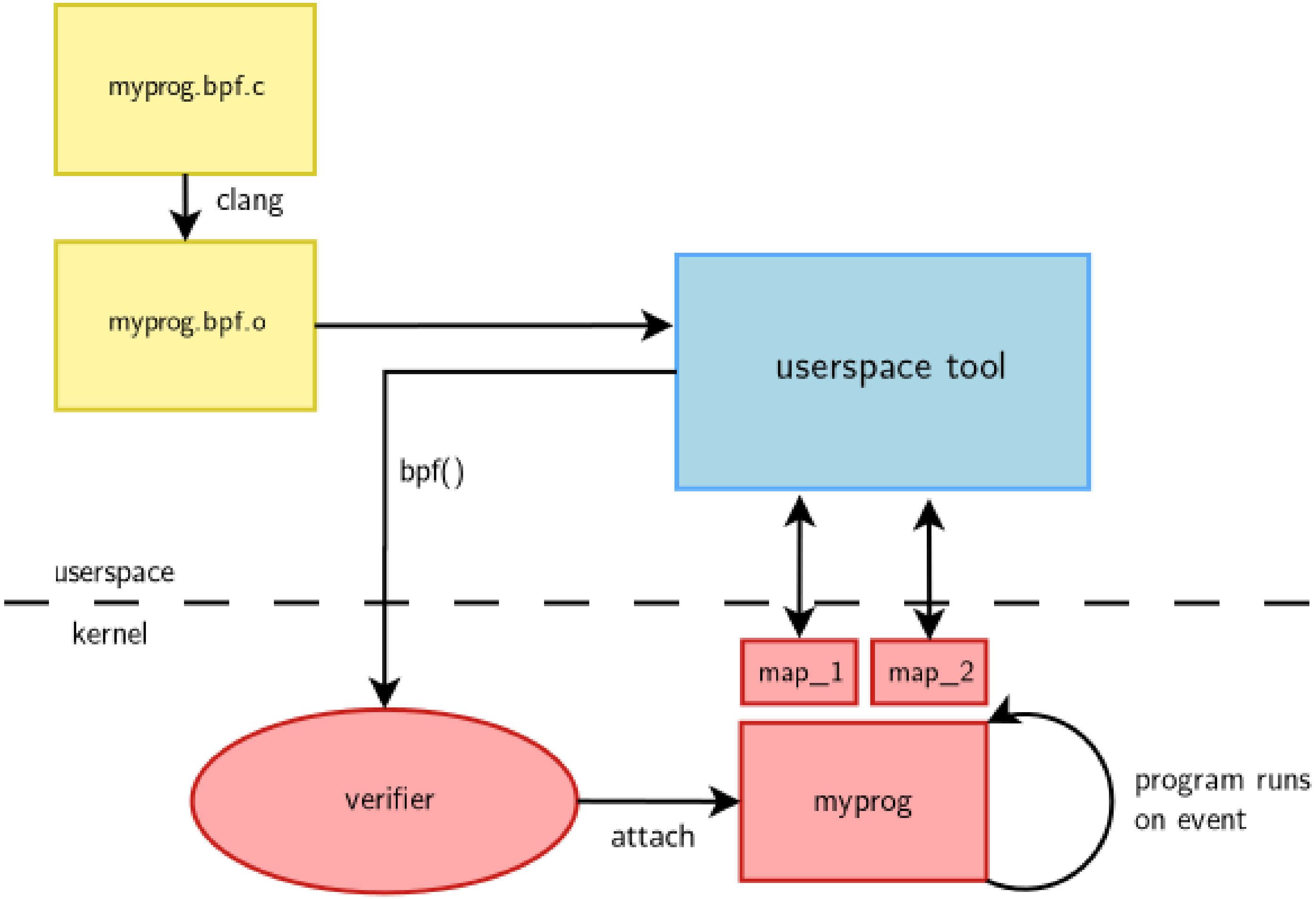
- you need some basic hardware:
  - a laptop
- you need some basic software:
  - an up-to-date Linux distribution (eg: Ubuntu 24.04)
  - you can also use the Virtual Machine image provided by Bootlin
- you need some basic knowledge:
  - general Linux knowledge
  - a bit of C language

eBPF basics

eBPF concepts

- eBPF allows to inject user programs into the Linux kernel
  - no need to write kernel code
  - no need to rebuild the kernel
  - no need to restart the kernel
- why do we want to inject programs into the kernel ?
  - to assist debugging/diagnostic on your system
  - to get tracing superpowers: spy on low level details in the kernel
  - to change how packets are handled
  - to run a custom scheduler
  - to set user-based security policies
  - and **many** more !

eBPF lifecycle



Basic eBPF management

eBPF program content

- an eBPF program is generally written in C
  - you define a main C function: this is your eBPF program body
  - you can write standard C in it
  - you can not call into any C library function
  - you can call many well-defined kernel functions (bpf-helpers and kfuncs)
- depending on your **program type**, your program will receive specific arguments
- depending on your **program type**, your program return value may have an effect on the kernel
- your program will be attached to a specific kind of **attach point** in the kernel: a *kprobe*, a *tracepoint*, a *cgroup*, a *network interface*, a *tc filter*, etc

A very simple eBPF program

```
SEC("fentry/__x64_sys_execve")
int BPF_PROG(my_first_program, const char *path,
                char *const _Nullable argv[],
                char *const _Nullable envp[])
{
        char fmt[] = "New program %s executed !";
        // This function comes from bpf-helpers
        bpf_trace_printk(fmt, sizeof(fmt), path);
        return 0;
}
```

A more advanced program

```
SEC("tc")
int my_second_program(struct __skbuff *skb)
{
    // This structure comes from the kernel
    struct iphdr iph;
    int ret;

    // This function comes from bpf-helpers
    ret = bpf_skb_load_bytes(skb, ETH_HLEN, &iph, sizeof(iph));
    if (ret)
        // Returning TC_ACT_OK lets the packet go in/out
        return TC_ACT_OK;

    if (iph.protocol != IPPROTO_UDP)
        // Returning TC_ACT_SHOT drops  the packet
        return TC_ACT_SHOT;

    // Returning TC_ACT_OK lets the packet go in/out
    return TC_ACT_OK;
}
```

bpf-helpers

- eBPF programs can not call any arbitrary kernel function
- But there are tens of functions exposed as `bpf-helpers`:
  - to generate traces into the `ftrace` buffer: `bpf_trace_printk`
  - to manipulate "maps": `bpf_map_lookup_elem`, `bpf_map_update_elem`...
  - to know about the current process being executed: `bpf_get_current_pidf_tgid`, `bpf_get_current_comm`...
  - to read, modify, steer packets: `bpf_skb_load_bytes`, `bpf_skb_change_type`, `bpf_redirect`...
  - etc
- To get the exact list, refer to `man 7 bpf-helpers`

Transfer data from/to eBPF programs

- You often need to exchange data between an eBPF program and a userspace program (or between two eBPF programs)
- The kernel exposes a specific kind of memory for this: **maps**
- Maps come in a wide variety:
  - basic arrays: `BPF_MAP_TYPE_ARRAY`
  - hashmaps: `BPF_MAP_TYPE_HASH`
  - queues: `BPF_MAP_TYPE_QUEUES`
  - ring buffers: `BPF_MAP_TYPE_RINGBUF`
  - and many more
- For this lab, you only need to know to use basic array maps

A program manipulating a map

```c
// We declare a single-cell array
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
    __type(key, int);
    __type(value, int);
} packet_count SEC(".maps");

SEC("tc")
int my_third_program(struct __skbuff *skb)
{
    int key = 0;
    int *value;

    // Try to fetch the first element in the array
    value = bpf_map_lookup_elem(&packet_count, &key);
    // If we manage to fetch the value...
    if (value) {
        //... increment it...
        *value++;
        //... and store it again.
        bpf_map_update_elem(&packet_count, &key, value, BPF_ANY);
    }
    return TC_ACT_OK;
}
```

Building eBPF programs

- We either need `clang` or `bpf-unknown-none-gcc`

```
$ clang -g -O2 -target bpf -c dns-blocker.c -o dns-blocker-bpf.o
```

Loading and attaching a program

- The way of attaching a program is tightly coupled to its type
- For any program type, we can:
  1. use existing command line tools like `bpftool`, `tc`, `xdp_tools`, etc (super quick, good for prototyping)
  2. write our own tool based on `libbpf` (easy, quick, customizable)
  3. use bare system calls (complex, but highly customizable)
- You will practice the first method during the workshop. If you manage to get it done quickly, you will be able to experiment the second method.

Managing programs with bpftool

- Listing loaded programs:

```
$ bpftool prog
```

- Loading and attaching a program:

```
$ bpftool prog loadall my_prog.bpf.o /sys/fs/bpf/foo autoattach
```

- Showing BPF programs logs

```
$ bpftool prog tracelog
```

Managing maps with bpftool

- Listing created maps:

```
$ bpftool map
```

- Showing a map content:

```
$ bpftool map dump name <map_name>
```

- Update the content of a map

```
$ bpftool map update name <map_name> key <key> value <value>
```

- `key` and `value` are space-separated hex bytes
- the number of bytes must match the key/value length

Attaching a packet classifier/action program with tc

- Some program types are too specific too be handled generically by `bpftool`
- For packet classifiers/filters programs, we can use `tc`:

```
$ tc qdisc add dev wlan0 clsact
$ tc filter add dev wlan0 egress bpf direct-action object-file \
    dns-blocker.bpf.o sec tc
```

- To remove the program:

```
$ tc qdisc del dev wlan0 clsact
```

The lab

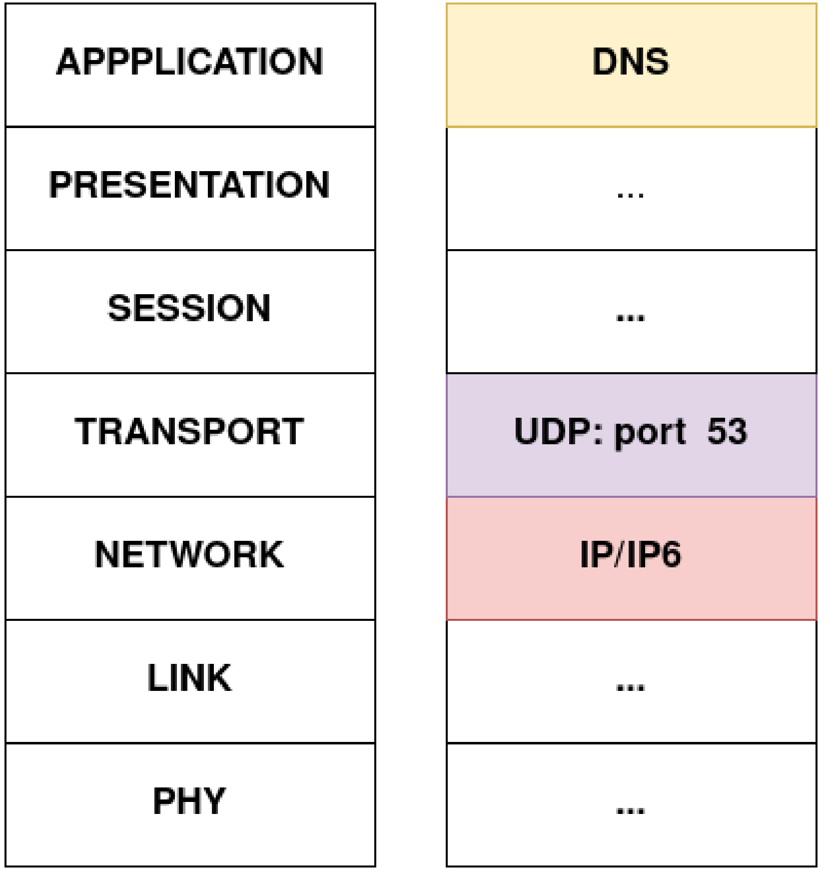
The mission

- Implement an ad-blocker, preventing ads from being fetched in our web browser
- For the sake of ~~overengineering~~ fun, we'll do it in **eBPF**
- There are various ways of blocking ads
- One common way is to prevent the corresponding DNS query to go out of your computer

The DNS protocol

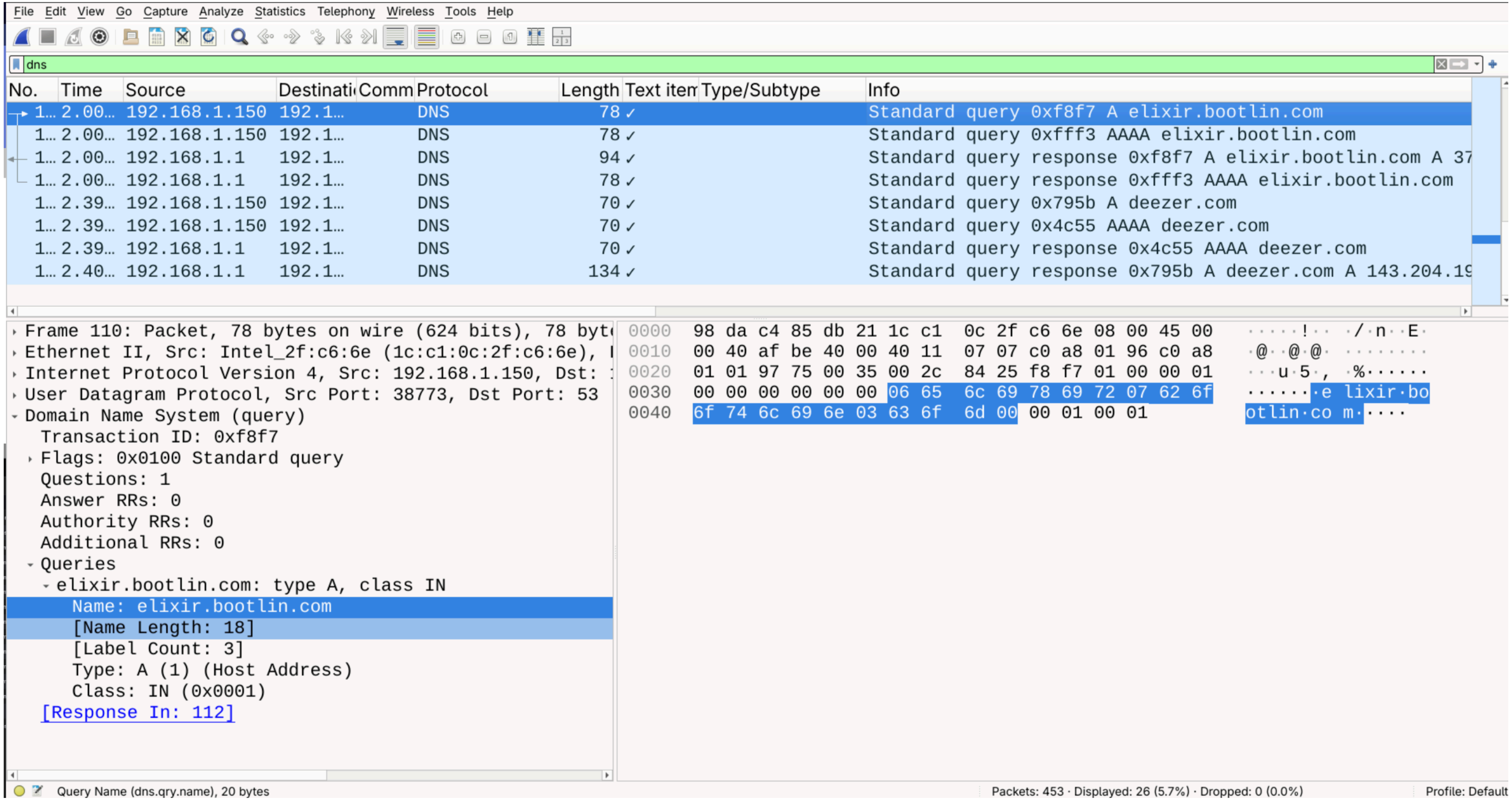
- each time we want to access a resource on the web, we ask for it through a specific URL
  - `"please fetch https://elixir.bootlin.com/static/style.css"`
- the system needs the corresponding IP address for the host part
- this is done by emitting a DNS **query** to a DNS server
  - `"what is the IP address for elixir.bootlin.com ?"`
- the server replies with a DNS **answer**
  - `"the address for elixir.bootlin.com is 37.27.174.60"`
- the web browser can now send the relevant request
  - `"37.27.174.60: HTTP GET static/style.css"`

DNS queries



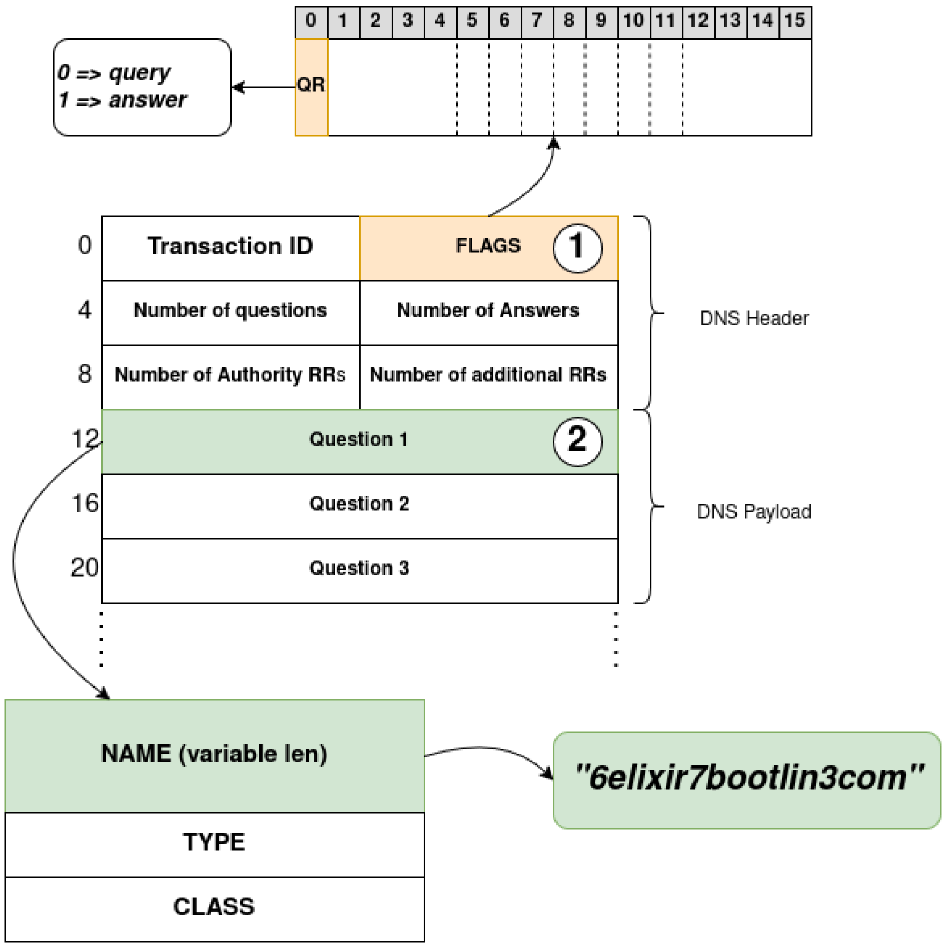
DNS in the OSI model

Tracing and understanding DNS queries with Wireshark



Tracing packets in wireshark

Parsing DNS queries in eBPF



A DNS query

1. parse DNS header
   - get QR flag
   - it must be **0** (meaning: **query**)
2. get first query in the payload
   - get name field
   - parse name field (byte by byte or string by string)
   - rebuild the queried host
     - 6elixir7bootlin3com
     - **elixir.bootlin.com**

Intercepting DNS queries with BPF

- We will catch outgoing DNS queries with an eBPF program attached to our network interface **egress**
  - it will be a **BPF_PROG_TYPE_SCHED_ACT** program, see prog example 2
- it will first contain a hardcoded domain to block
- the program will decapsulate and validate all layers:
  - `ethernet` -> `ip` or `ip6` -> `udp` -> `dns`
- if the packet is anything else, it will let it go (`return TC_ACT_OK`)
- otherwise, we check if it is a query, and if so, if the target host is the one we want to block
  - if so, we drop the packet (`return TC_ACT_SHOT`)

It's now your turn !

- We provide some basic code as well as step-by-step instructions:

```
$ git clone https://github.com/bootlin/ebpf-workshop-cdl
```

- Take your time to **read** and **understand** each instruction. Feel free to ask us for clarifications or more details.
- You can run the whole lab on your machine if you have a recent distribution. Otherwise, run the labs on the dedicated Virtual Box machine:

```
$ wget https://f000.backblazeb2.com/file/bootlin-ebpf-workshop/bootlin-
ebpf-workshop.ova
```

- Feel free to pair up with someone: peer programming makes it even funnier !
- When blocked on an issue, try to get familiar with the basic tools discussed earlier so that you can analyze it !

Useful resources

Useful resources

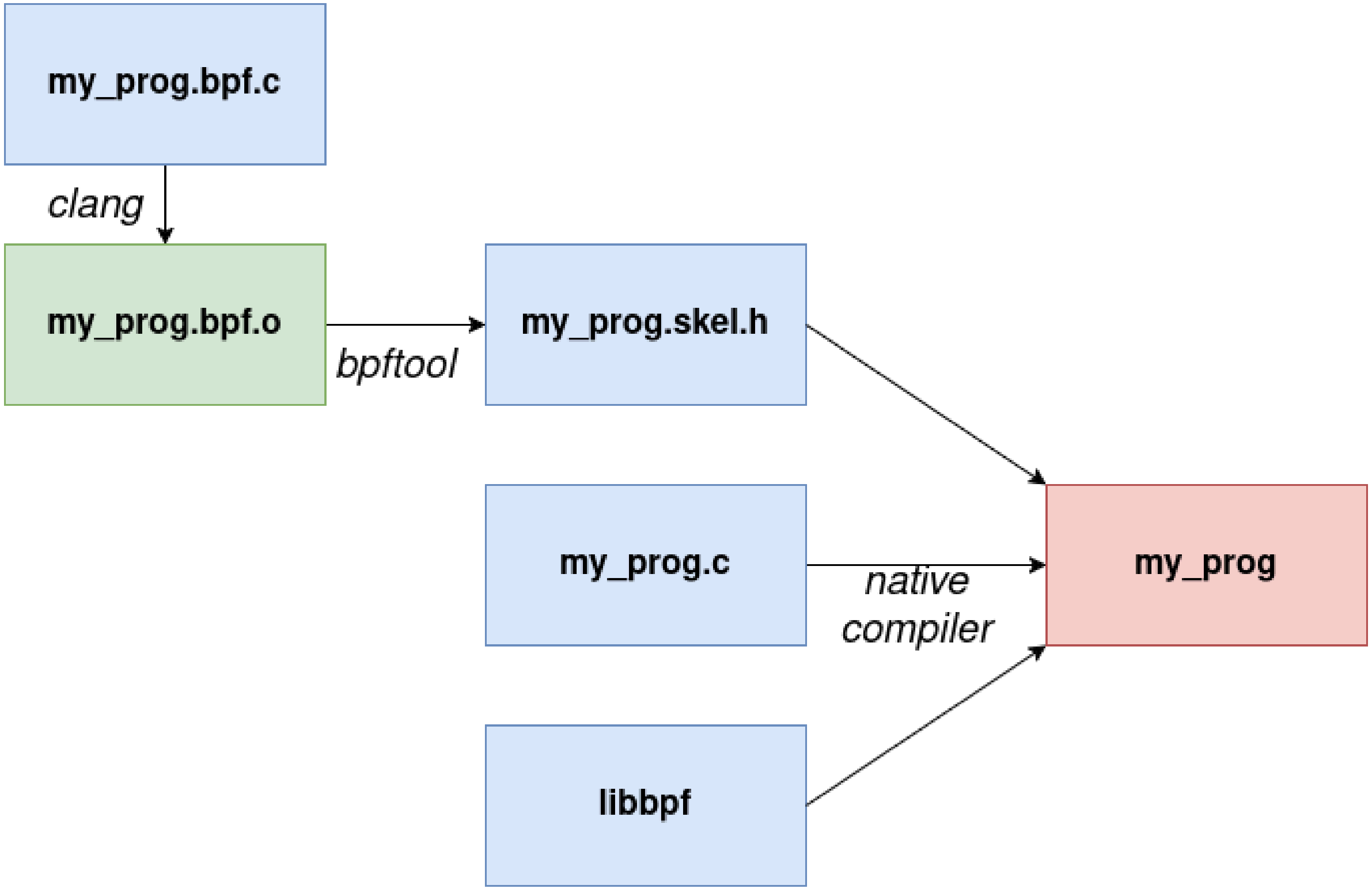
- docs.ebpf.io
- Bootlin "Debugging, Profiling, Tracing and Performance Analysis" training
- Bootlin "Embedded Linux Networking" training
- man 7 bpf-helpers
- Linux kernel selftests (those are good examples)

Going further

Building a standalone userspace tool

- Your current implementation needs some manual steps to work
  - loading the program with `tc`
  - configuring the program through a map with `bpftool`
- What if instead of doing all of this, we could get a single userspace binary (a daemon) doing all of this for us ?
- This can be done thanks to libbpf

libbpf-based userspace programs



Building a libbpf application

Generating a BPF skeleton file

- You can turn your `my_prog.bpf.o` file into a “skeleton file”
  - this is a C header (`my_prog.skel.h`)
  - it contains the whole BPF program as a byte array
    - you don’t need `my_prog.bpf.o` anymore
  - it contains auto-generated API to manipulate your program from userspace

```
$ bpftool gen skeleton my_prog.bpf.o > my_prog.skel.h
```

A very simple libbpf program

```c
#include <unistd.h>
#include "my_prog.skel.h"
int main(int argc, char *argv[])
{
    struct my_prog *skel;
    skel = my_prog__open_and_load();
    if(!skel)
        exit(EXIT_FAILURE);
    if (my_prog__attach(skel)) {
        my_prog__destroy(skel);
        exit(EXIT_FAILURE);
    }
    while(true) {
      sleep(1);
    }
    return 0;
}
```

Writing and compiling our userspace program

- You can use auto-generated functions from your program skeleton
- You can use generic functions from libbpf
  - eg: to manipulate your program maps
- Once done, you can build your program thanks to a native compiler:

```
$ gcc my_prog.c -o my_prog -lbpf
```

- And voila, you now have a standalone eBPF tool !