eBPF for perfomance analysis and networking

Matthieu De Beule

June 2020

Contents

1 Introduction

The Linux kernel has a pretty clear separation between user-space applications and kernel-space, that is delineated by the system calls. This separation exists for good reasons, such as imposing restrictions on applications regarding access to hardware or the memory of other applications, and more generally protecting integrity and safety.
However, this separation is responsible for some limitations in term of functionality and/or performance, emergent from the guarantees given by the user-space/kernel-space boundary. It is hard to, for example, debug kernel functions without a lot of recompiling (although this has gotten somewhat better), or debug performance issues on a live system with low overhead and without stability issues.
BPF is a generic kernel execution engine in Linux, that executes restricted bytecode in kernel-space. It maintains security guarantees and performance and stability by imposing restrictions to this bytecode.

This semester project had the goal to learn to know BPF: what it can do, the tooling around it, how it can be applied for networking applications.

2 What is (e)BPF?

2.1 Berkeley Packet Filter, or classic BPF

BPF originally refers to "Berkeley Packet Filter", first created in 1992. It allows filtering packets with a kernel agent that can discard packets in the kernel itself, without needing any copies back and forth to and from user space. This is implemented as an interpreter for a machine language that runs on the BPF virtual machine.

For example, tcpdump (in user space) provides filter instructions to the BPF VM (in kernel space): before execution the instructions are verified for safety (since they were to be executed in kernel space). The filter instructions can then filter the wanted packets and no unnecessary packet copying has to happen between kernel and user space.

To be able to run, BPF programs must be safe, deterministic and pass a verifier that proves that. For example: loops, unsafe access to memory, restricted kernel calls are not allowed by the verifier.

More recently, BPF has been extended significantly: these extensions were initially referred to as "eBPF" but are now referred to as "BPF" and the original BPF can be referred to as "cBPF" (for "classical"). Starting here, any time I refer to "BPF" I am talking about eBPF, and I will use "cBPF" to refer to the original version. Of course, naming things is hard and this seems to be no exception.

> "There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-one errors" [8]

2.2 Modern BPF

2.2.1 History and differences with cBPF

Alexei Starovoitov proposed what we now know as BPF in 2013 [1]. It was then referred to as "extended" BPF, or eBPF.

The cBPF VM was very limited, it had two 32-bit registers, 16 memory slots for storage. It could be attached to sockets, xtables, or (later on) seccomp.

While cBPF was a rather obscure technology with its applications limited to packet filtering, BPF is now a general-purpose virtual machine that can run programs on a wide variety of events, including kernel events such as disk I/O, network I/O, basically everything in the kernel using tracepoints and kprobes.

The BPF VM is greatly improved, with 10 64-bit registers, infinite storage thanks to BPF maps (see the **BPF Maps** section), and allows some restricted kernel calls. Using 64-bit registers allows for one-to-one mapping between registers from the 64-bit hosts to the BPF VM, which makes for minimal performance overhead.

2.2.2 BPF virtual machine: limitations and benefits

The BPF verifier guarantees safety by rejecting any unsafe code.
Examples of things it rejects: unbounded loops (since Linux 5.3 bounded loops are now allowed), programs that are too large (previously limited to 4K instructions, now 1 million and if anyone ever gets over this it will be increased again according to the maintainer), out of bounds accesses, malformed jmps, unreachable instructions.

This has the interesting outcome that BPF is not Turing-complete, the program must be a DAG (directed acyclic graph).

This verifier allows BPF programs to have very good performance characteristics, since they can run in kernel space without any possible security or stability problems. They are bounded, will always terminate, will never cause a kernel panic.

These properties allow things to be done from user space that could previously only be done by writing a kernel module, and those have several drawbacks that BPF does not have:

- Significant kernel engineering knowledge is required to develop kernel modules
- Kernel modules may introduce bugs leading to kernel panics or security vulnerabilities

However, kernel modules do not have the restrictions on what kernel functions can be called that BPF does, but this again comes with risks of bugs and misuses that cannot happen with BPF.

3 Function tracing with little overhead

The most popular use of BPF is performance tracing: instrumenting in various part of the kernel, including the network stack, disk I/O, CPU perfomance analysis along various metrics, memory, file systems.

There are three main ways to do BPF programming: `bpf_trace`, BCC and LLVM.
`bpf_trace` is the highest-level one, and is useful to write one-liners quickly to solve a particular problem (it provides its own higher-level language).
BCC is a compiler allowing you to write C code and compile it into BPF programs (it also has tooling to allow you to write the user-space part of a program in Python).
LLVM IR can also be used to write BPF, but is only really used by developers of BCC or `bpf_trace` (it's LLVM's "Intermediate Representation", which is a portable assembly language)

3.1 BCC and using BPF from python

The most accessible way of working with BPF for tracing is probably with BCC and its python user-space bindings.
You write a BPF program in C, then you load it from within python with `b = BPF(src_file="bpf_program.c")` and then you can access elements declared in C from python, where the `BPF_PERF_OUTPUT(name)` macro is used in C to create an output channel and then accessed like so in python: `b["name"]`.

A useful trick, before loading the BPF program, is to put it inline in the python program and do string subsitutions on it according to flags or arguments given by the user. This means a lot of logic and branches don't have to be done in BPF, which can make for better performance since the BPF program is run every time the hook is triggered, while the substitution is done before-hand and only once.

3.2 kprobes, and other ways of instrumenting kernel functions

3.2.1 What is a k(ret)probe?

> A kprobe (kernel probe) "is a set of handlers placed on a certain instruction address". [2]

For x86, the instruction at the instrumented function's address is replaced by an int3 breakpoint instruction. When this instruction is executed, the breakpoint handler will execute the kprobe handler (our BPF program). Then the original instructions are executed, and normal execution of the program resumes.

The kretprobe is similar, but executes our program after the original function returns, instead of before.

3.2.2 kprobes vs. tracepoints

kprobes are one of the most useful tools to do performance analysis on kernel functions. You can use kprobes to instrument any function in the kernel, without any needed changes in the kernel itself. This is as opposed to tracepoints, who need to be made explicitly available as part of static instrumentation points.

So, if kprobes are available for everything, why do tracepoints exist? The answer is interface stability: kprobes do not offer a stable API, and when using kprobes you should consider them to be instrumenting one particular version of the kernel, and it could stop working for the next. While learning to work with kprobes I found this out after a lot of time lost trying to get an example from BCC to work. This did give me some sense of how to navigate the Linux kernel tree, in order to find the function I want to instrument.

Since the emergence of BPF, there is now a lot of demand for tracepoints, so there should be less and less need to rely on kprobes in the future.

3.2.3 uprobes

I will also quickly mention uprobes, which are analogous to kprobes but for user-space programs. One could for example instrument the `readline()` function in `bash` by attaching to the uprobe `/bin/bash:readline`. This enables BPF to be used for the complete software stack, not only the kernel.

4 BPF maps: the data structures of BPF

A big feature of BPF are the data structures that are available, namely BPF maps.

BPF maps are key/value stores, accessible from different BPF programs or from user-space.

From `man 2 bpf`:

```
  [maps] are a generic  data structure for storage of different data types.
Data types are generally treated as binary blobs, so a user just  specifies  the
size of the key and the size of the value at map-creation time.  In other words,
a key/value for a given map can have an arbitrary structure.

A user process can create multiple maps (with key/value-pairs being opaque bytes
of  data) and access them via file descriptors.  Different eBPF programs can ac-
cess the same maps in parallel.  It’s up to the user process and eBPF program to
decide what they store inside maps.
```

Several types of maps are available for different use-cases:

- Some general-purpose maps:
  - `BPF_HASH`: hash map (associative array)
  - `BPF_(PERCPU_)ARRAY`: array (int-indexed, optimized for fast lookup/update). The PERCPU variant creates one copy per CPU, and are not synchronized (this should be preferred whenever cache coherence problems could introduce significant latency)
- Some maps that can be useful to generate statistics:
  - `BPF_HISTOGRAM` is used to generate histograms, there are helper methods to print these in user space, and to generate them according to a log scale with BPF functions
  - `BPF_PERF_ARRAY` is used to return a hardware-calculated counter of the number of cycles elapsed
- `BPF_STACK_TRACE` is used to store stack traces

One of the big reasons BPF tracing has such a low overhead, is that the only data passed from kernel-space to user-space is the results of the tracing, for example when using the special histogram map.

5 Modifying packets using BPF: XDP and tc

Everything discussed previously (tracepoints, kprobes, etc.) is read-only on the values passed to and returned from the function, which is useful for tracing and performance analysis but does not allow to modify packets in the TCP stack for example.

For instance, data access when using kprobes is read-only and transparently wrapped using `bpf_probe_read()` calls by BCC.

There are various ways to modify packets with BPF. Two notable ways are XDP (eXpress Data Path) and tc (**t**raffic **c**ontrol).

5.1 XDP (eXpress Data Path)

XDP [3] is pretty low-level, its hooks happen before the kernel networking stack (which has good and bad consequences: it allows to make early decisions and improve perfomance by not allocating skb structs, it reduces attack surface in contexts like DoS defence, but also it makes for added complexity to work with the packets since you have very raw, DMA, pointers to work with).
You can drop packets, modify them, send them out again on the same network interface (and with some high perfomance hardware all of this can be offloaded onto the NIC!).
XDP integrates with the existing stack (as opposed to something like DPDK where the stack is implemented in userspace).

Working with XDP

Since there was little to no information regarding how to use XDP with BCC, it seems it is mostly used directly using libbpf, which is part of the kernel tree under `tools/lib/bpf`. The XDP project maintains extensive documentation and tutorials [4]

- XDP works with pointers to the beginning and end of the raw packet data. You literally have just pointers to the beginning and the end in the `xdp_md` struct. The packet is accessed via DMA (i.e. no copies are made for this).
- there is a concern about endianness since the network order can be different than the endianness of the architecture the program is running on. There are helper functions to convert from network order to host endianness and back.
- Most rewriting operations require recalculating checksums, which has to be done explicitly (there are helpers for with `skb`s, but as discussed previously XDP only has raw pointers)
- to work with packets, you have to unwrap layer by layer until you get to the one that interests you (so ethernet, IPv4/IPv6, ICMP/UDP/TCP/...)
- the BPF verifier maintains strong guarantees about what addresses are accessed. This means you have to do some checks that might be superfluous

just to make the program pass the verifier, but this is how BPF can maintain security and stability while not giving up performance.

- Different actions can be taken on the packet: it can be dropped (`XDP_DROP`), passed (`XDP_PASS`), retransmitted out on the same interface (`XDP_TX`), transmitted out of another interface (`XDP_REDIRECT`). `XDP_ABORTED` also drops the packet but emits an event to the `xdp:xdp_exception` tracepoint (without overhead if there is nothing listening).

5.2 tc (traffic control)

tc is higher up in the stack and you work with skb structs instead of the xdp pointers to begin and end. It can be used for traffic shaping, scheduling, policing, dropping on both ingress and egress (whereas XDP is only on ingress). Compared to XDP, it is better at packet mangling, since the skb struct is available at the stage of tc's hook.

6 Using BPF: practical uses

6.1 Tracing with BCC and Python

As an example, I have made a python program [7] using a kprobe hook to instrument calls to `tcp_connect`, which is responsible to "build a SYN and sends it off" (prototype [5], ipv4 implementation [6]). It collects data in a struct `data_t` and accesses the struct members in python to print them.

When `detect_syn.py` is run, it will print the PID, time and process name every time a SYN is sent. This is being done by instrumenting a function in the kernel, so one could do tracing in an analogous manner for anything you could think of, by just changing what function we attach a kprobe to.

```
b = BPF(text="""
#include <net/sock.h>
#include <linux/sched.h>
// define output data structure in C
struct data_t {
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);
int kprobe__tcp_connect(struct pt_regs *ctx,
                        struct sock *sk) {
    struct data_t data = {};

    data.pid = bpf_get_current_pid_tgid();
    data.ts = bpf_ktime_get_ns();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
""")
def print_event(cpu, data, size):
    event = b["events"].event(data)
    print(f"PID: {event.pid}, Time: {event.ts}, process name: {event.comm.decode()}")


b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
```

6.2 Shaping network traffic with XDP

I wanted to make a basic load-balancer with XDP. The basic idea of a load balancer is to distribute packets among servers:

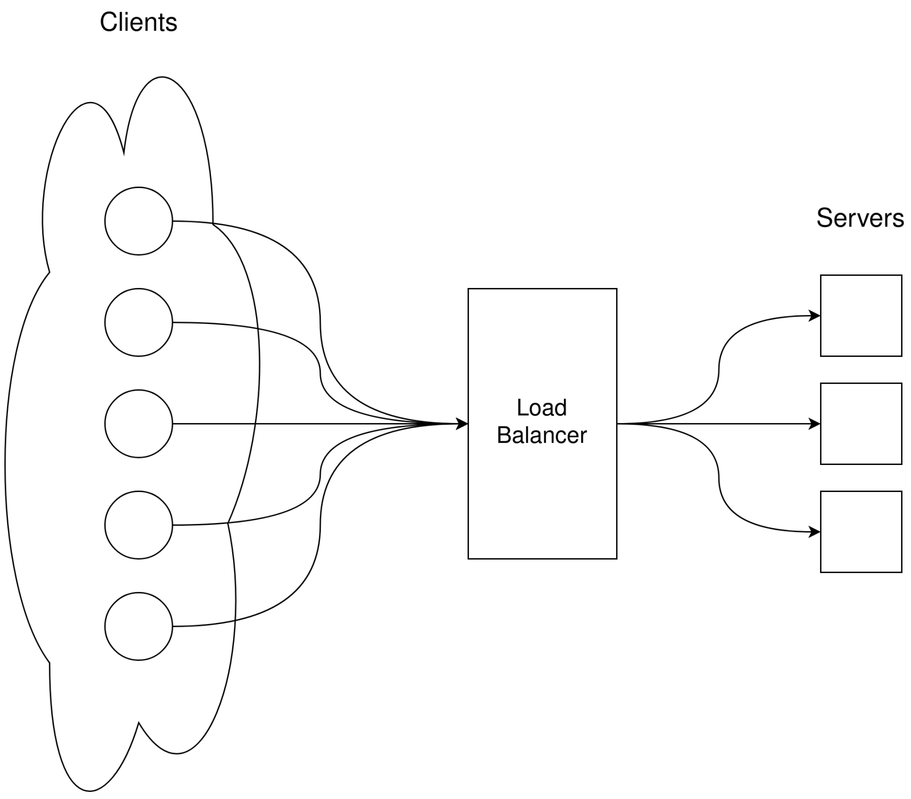

Figure 1: Diagram of general idea of a load balancer

The load balancer needs to redirect incoming packets to one of the servers, splitting them either by applying some heuristic dependent on the IP address of the client, or by remembering the association client/server so that connections can be maintained. An example of a very simple heuristic is a modulo on the IP address (with 3 servers: the result of {IP}%3 chooses the index of the server).

The IP and MAC addresses of the servers are stored in **BPF Maps**, saved there when attaching the BPF program, and retrieved by the BPF program to look up its target during load balancing.

In the XDP BPF program, we can use the `XDP_TX` return value after modifying the packet as needed (modifying the destination IP and MAC address, and adjusting checksums appropriately). This bypasses the Linux network stack, including its `sk_buffer` allocation.

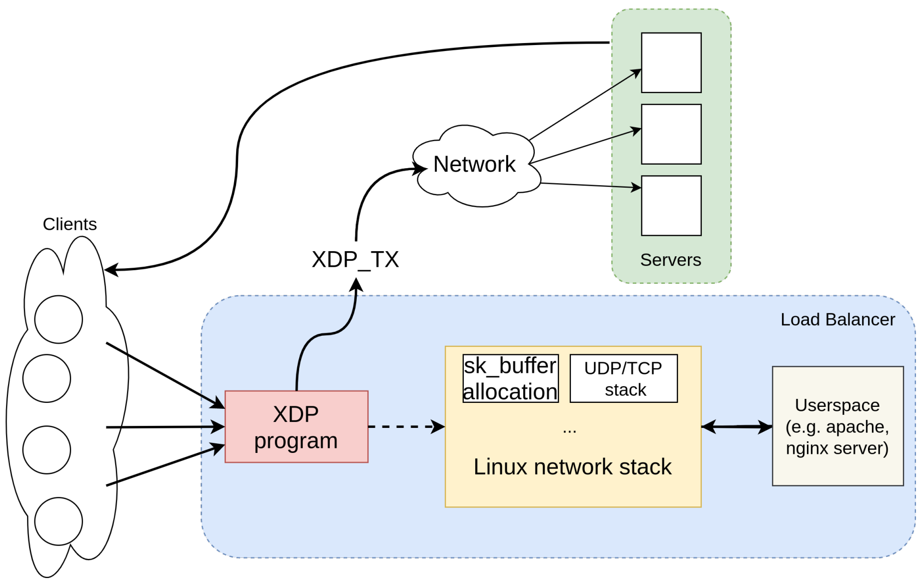

Figure 2: XDP_TX schema, showing that retransmitted packets bypass the kernel network stack

To enable packets to be sent back to the clients from the server, one would also need to rewrite the packets on the egress of the servers, so that from the client's perspective, the connection is established with the load-balancer. This can also be done using BPF, with `tc`.
To accomodate this, the network must be set up in a manner that allows the multiple servers and the load balancer to all send packets pretending to send packets with the same IP and MAC address filled in.

7 Conclusion

BPF is a very powerful technology, with great potential.

As we have seen, it can be used for tracing, debugging the kernel, implementing every network shaping application imaginable, with high performance and with security guarantees.

It already has a very important role, with industry leaders adopting it at massive scales, in such a way that now most of the internet's traffic runs through a BPF program of some kind [9]. It is now also used in Android phones (every phone with Android 9 and up [10]), which demonstrates that its uses are not limited to server applications.

In the future, BPF could also be useful to modularize the kernel more, in the sense that parts of the kernel can be extracted to run in the BPF VM, gaining the safety guarantees provided by that environment.

8 References

- [1] https://lkml.org/lkml/2013/9/30/627 :
Patch from Alexei Starovoitov, Mon, 30 Sep 2013
Subject: `[PATCH net-next] extended BPF`
- [2] https://lwn.net/Articles/132196/ :
An introduction to KProbes
- [3] https://github.com/xdp-project/xdp-paper/blob/master/xdp-the-express-data-path.pdf :
Paper for XDP: "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel"
- [4] https://github.com/xdp-project/xdp-tutorial :
XDP tutorial
- [5] https://elixir.bootlin.com/linux/v5.5.6/source/include/net/tcp.h#L446 :
Prototype for the `tcp_connect` function, in kernel v5.5.6
- [6] https://elixir.bootlin.com/linux/v5.5.6/source/net/ipv4/tcp_output.c#L3565 :
IPv4 implementation for the `tcp_connect` function, in kernel v5.5.6
- [7] https://gitlab.epfl.ch/debeule/bpf/-/blob/master/code/detect_syn.py :
`detect_syn.py`, a python program using BCC and the python bindings to log SYN connections by tracing the `tcp_connect` function.
- [8] https://twitter.com/timbray/status/506146595650699264 :
Common saying, original quote attributed to Phil Karlton: https://skeptics.stackexchange.com/questions/19836/, "off-by-one" bit reportedly added by Leon Bambrick
- [9] https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/ :
Cloudflare blog post about BPF usage
- [10] https://source.android.com/devices/tech/datausage/ebpf-traffic-monitor :
AOSP documentation: eBPF Traffic Monitoring