An introduction to Linux kernel programming with eBPF.

Battlemesh v14, Roma

Baptiste Jonglez

September 19, 2022

Outline

Goal: understand eBPF basics and give pointers

eBPF: beyond userspace and kernelspace

Application to system and network visibility

Application to network programming

Conclusion

Introduction: modern system and network programming

Typical problems

- My complex program has performance issues, how to debug this?
- I need visibility into the kernel behaviour: syscalls, network access, scheduling…
- I need flexible and fast packet processing: filtering, encapsulation, container networking…
- I need to offload some hardware-related tasks in the kernel

Two main needs: **system visibility** and **kernel programmability**

System / network programming models

Userspace

- Good: flexible, safe, easy to program, portable
- Bad: no direct access to hardware or kernel internal

Kernelspace

- Good: fast, direct access to hardware
- Bad: hard to program / debug / maintain, unsafe

Rigid interface between userspace and kernelspace: syscalls, basic statistics (but also perf, kprobe)

eBPF: the best of both worlds?

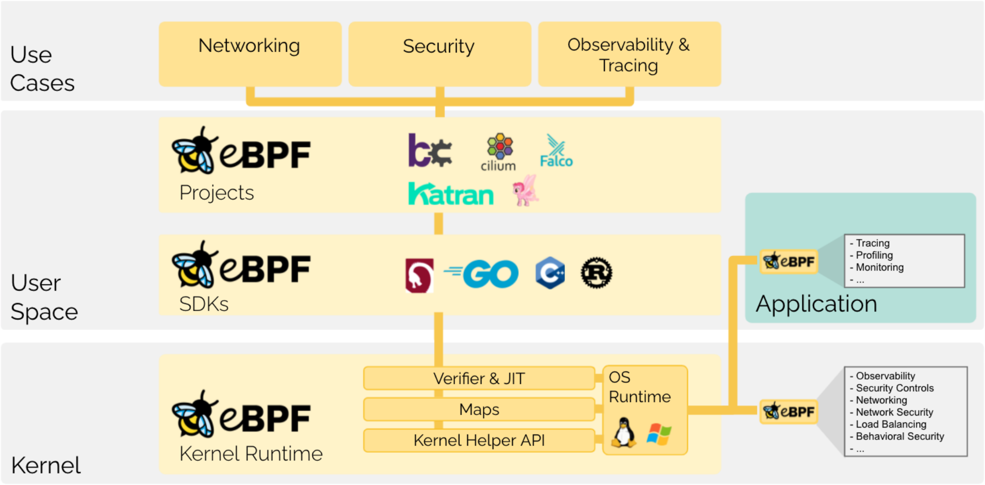


Figure: Image from https://ebpf.io

A simple BPF walkthrough: tcpdump

Capture network packets that match a given filter expression (`man pcap-filter`).

```
tcpdump "host 1.2.3.4 and udp port 53"
```

Work done in libpcap

- `pcap_compile(string)` → returns BPF bytecode
  - classical Flex/Bison lexer, simple code generation
  - bonus: run `tcpdump -d` to see the bytecode
- `pcap_setfilter(bytecode)` → loads BPF bytecode into kernel
  - check bytecode validity
  - `setsockopt(socket, SOL_SOCKET, SO_ATTACH_FILTER, bytecode)` on a raw socket
- **filtering is now done in the kernel! BPF = Berkeley Packet Filter**

A simple BPF walkthrough: tcpdump

Capture network packets that match a given filter expression (`man pcap-filter`).

```
tcpdump "host 1.2.3.4 and udp port 53"
```

Work done in libpcap

- `pcap_compile(string)` → returns BPF bytecode
  - classical Flex/Bison lexer, simple code generation
  - bonus: run `tcpdump -d` to see the bytecode
- `pcap_setfilter(bytecode)` → loads BPF bytecode into kernel
  - check bytecode validity
  - `setsockopt(socket, SOL_SOCKET, SO_ATTACH_FILTER, bytecode)` on a raw socket
- **filtering is now done in the kernel! BPF = Berkeley Packet Filter**

This is classical BPF from around 30 years ago

McCanne, Steven, and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In USENIX winter, vol. 46. 1993.

What’s new with eBPF

New features with eBPF

- Higher **performance** (new instructions, JIT compiling)
- **Many hooks** throughout the kernel that can load eBPF programs
- **Access** to some **kernel data structures** and helper functions
- **Communication with userspace** through “maps”

eBPF hooks

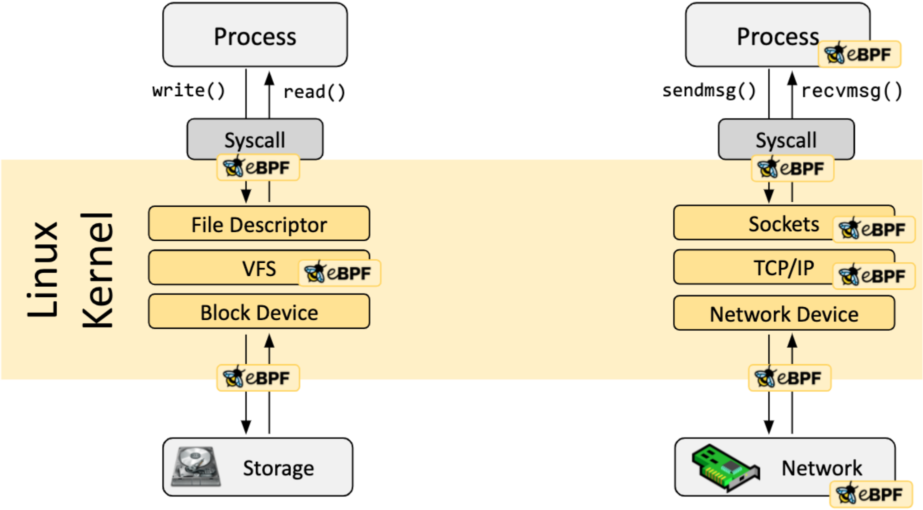


Figure: Image from https://ebpf.io

eBPF static verification

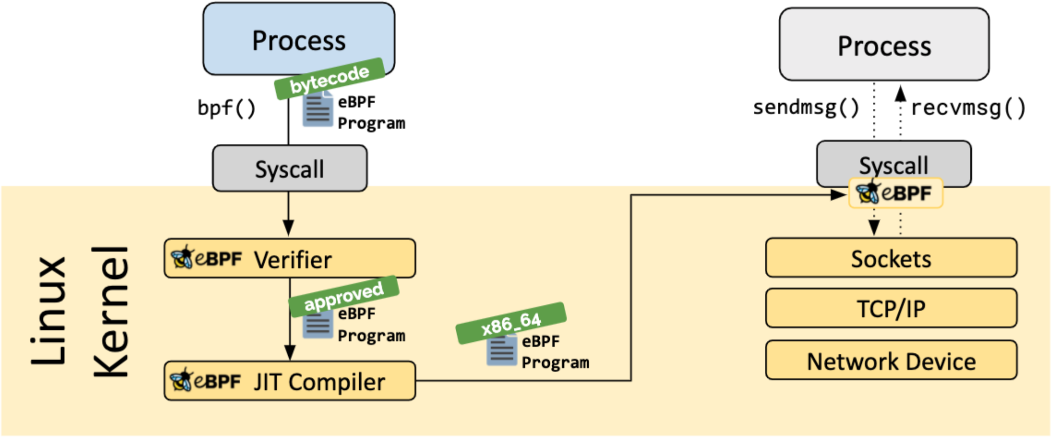
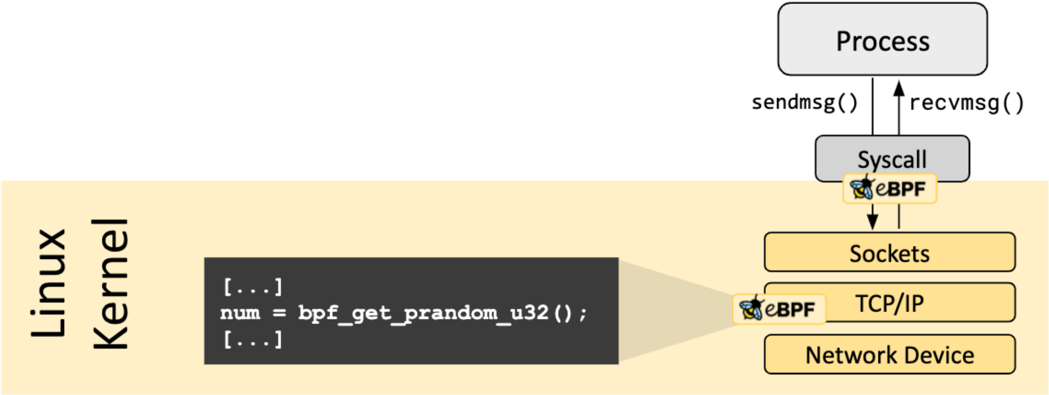


Figure: Image from https://ebpf.io

eBPF kernel helpers



```
[...]
num = bpf_get_prandom_u32();
[...]
```

Figure: Image from https://ebpf.io

eBPF maps: communication with userspace

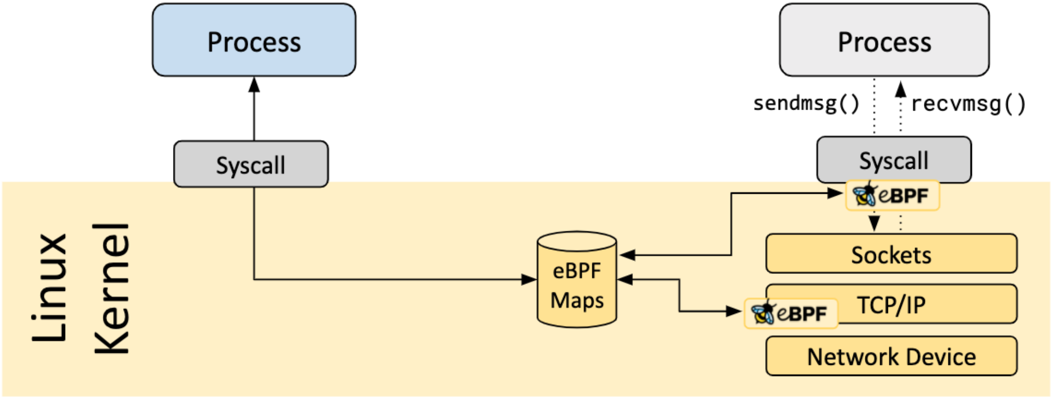


Figure: Image from https://ebpf.io

System and network visibility

Reference

See work of Brendan Gregg: https://www.brendangregg.com + books

Demo time

- bcc
- bpftrace
- ply
- pwru

Network programming with XDP

XDP

Demo

Conclusion

Conclusion

- Very flexible and powerful mechanism to safely run code in the kernel.
- Many different usages in the kernel, and increasing.
- High-level tools are very well documented and accessible
- The low-level infrastructure is complex, may be worth it for specific projects.
- Peak of activity since a few years: many projects, companies, tools…

Pointers

References

- https://ebpf.io
- https://docs.cilium.io/en/latest/bpf/
- https://lwn.net/Kernel/Index/#Berkeley_Packet_Filter
- https://www.brendangregg.com