An introduction to Linux kernel programming with eBPF
Datamove seminar

Baptiste Jonglez

1 December 2021
Outline

Goal: understand eBPF basics and give pointers

eBPF: beyond userspace and kernelspace

Application to system and network visibility

Application to network programming

Conclusion
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?

<table>
<thead>
<tr>
<th>Use Cases</th>
<th>Networking</th>
<th>Security</th>
<th>Observability &amp; Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Projects</strong></td>
<td>eBPF</td>
<td>lxc</td>
<td>Katran</td>
</tr>
<tr>
<td><strong>SDKs</strong></td>
<td></td>
<td>cilium, Falco</td>
<td></td>
</tr>
<tr>
<td><strong>Kernel Runtime</strong></td>
<td>eBPF</td>
<td>OS Runtime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Verifier &amp; JIT</td>
<td>Maps</td>
<td>Kernel Helper API</td>
</tr>
</tbody>
</table>

**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

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

![Diagram showing eBPF static verification process](https://ebpf.io)

**Figure**: Image from https://ebpf.io
eBPF kernel helpers

Figure: Image from https://ebpf.io

```c
[...] num = bpf_get_prandom_u32(); [...] 
```
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
| XDP | Demo |
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://lwn.net/Kernel/Index/#Berkeley_Packet_Filter