Alma Mater Studiorum - Università di Bologna

Class of Infrastructures for Cloud Computing and
Big Data M

eBPF Observability
in kernel space vs user space

Luca Cimino

25 Luglio 2024

eBPF

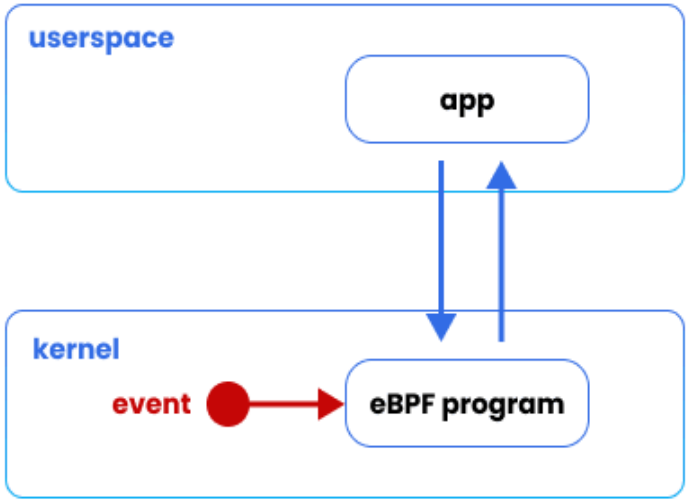
eBPF is a technology of Linux kernel to **run user-defined programs in kernel space**.

- Extend the capabilities of the OS kernel without modifying kernel's source code or adding additional modules

Networking
Packets processing and analysis

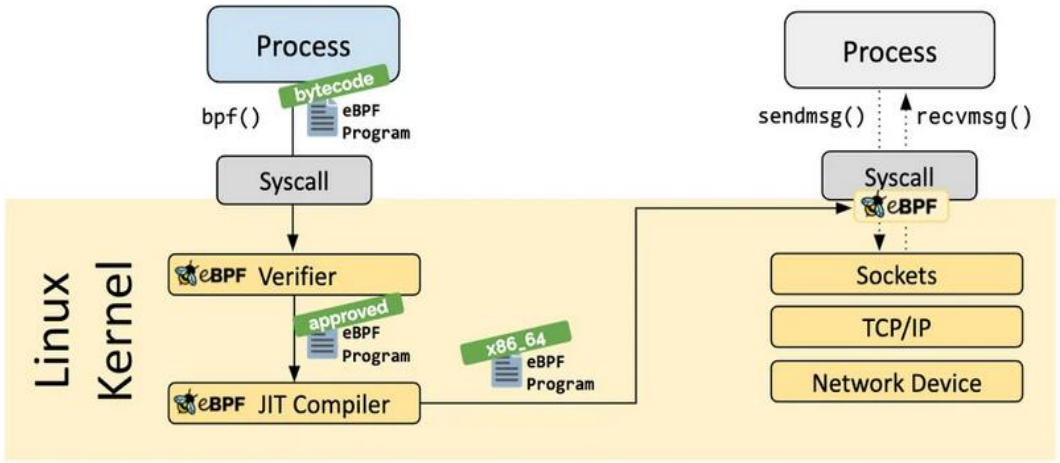
Observability
System calls, socket events, I/O operations, resources usage

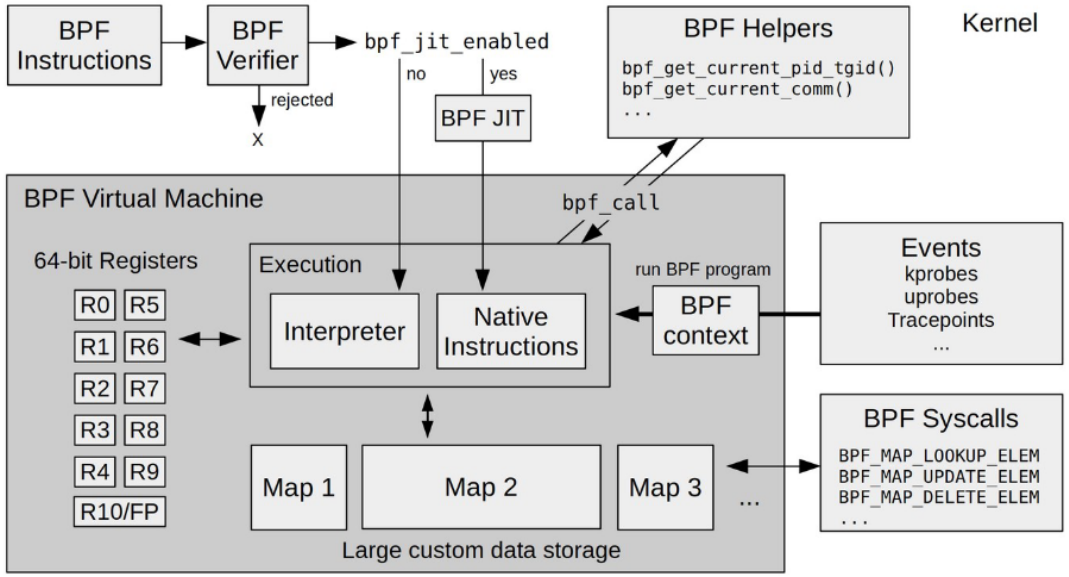
Security
Firewalls, intrusion detection, network monitoring



eBPF: Loading and Execution

- **Loading**: eBPF bytecode is approved by a verifier and compiled to native machine code by a JIT compiler.
- **Execution**: eBPF programs are event-driven





- **Sandboxed environment**: eBPF defines a virtual machine with its own istruction set to run isolated programs in kernel space

Hook Point

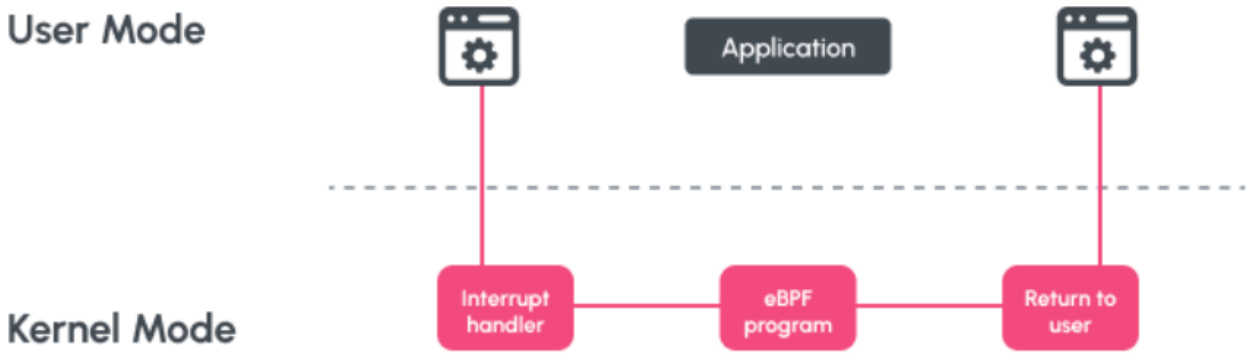
Event-driven: eBPF programs run when the kernel or an application passes a certain **hook point**.

- **Tracepoint**: static kernel hook point (system calls, network events, devices events, …)
- **Watchpoint**: access to a memory address

It is possible to create custom hook point:

- **kprobe**: attached to a kernel function
- **uprobe**: attached to a user application function

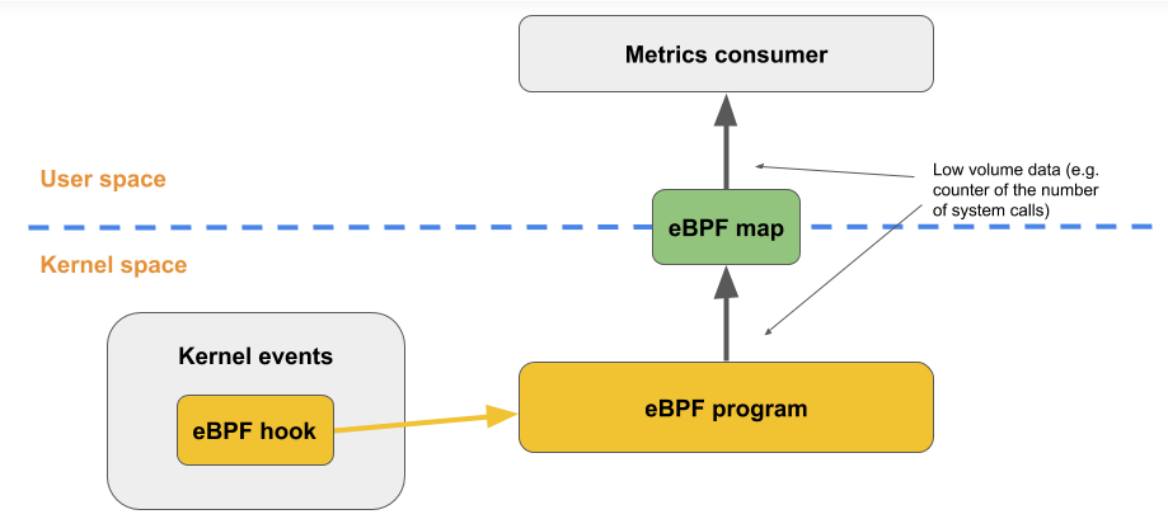
With an **uprobe** we are paying the cost of an additional context switch



Maps

eBPF programs provide visibility between the kernel and user space using BPF maps

- allows to **exchange information** between the eBPF program in kernel space and a program in user space
- efficient key/value storage



Maps are accessed from the user space through a **file descriptor**

```
fd = bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
```

They can be ***pinned*** to an eBPF file system to overcome the process lifetime

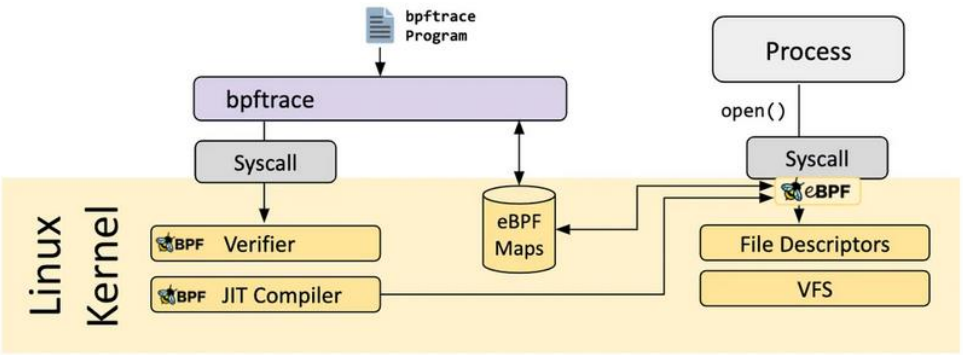
Observability in eBPF

eBPF represents a secure, isolated, and non-obtrusive observability tool to monitor events in the kernel.

Pros of eBPF for Observability

1. **Secure**: eBPF programs run in a sandboxed environment
2. **Safe**: the kernel verifies programs before it allows them to execute
3. **Kernel visibility**
4. **Efficient** and **minimal intrusion**: programs are event-driven, so CPU cycles are used only when needed
5. **Trasparent**: don't need to modify the Linux kernel or load kernel modules

bpftrace is a simple, high-level tracing language for writing eBPF programs, making it accessible even to those who are not experts in BPF.



CPU Profiling

eBPF provides full visibility about **cpu usage**. Different approaches can be used to profiling all processes that are running in the system:

1. Sampling system processes 99 times per second and counts the number of hits for each process

```
sudo bpftrace -e 'profile:hz:99 { @[comm] = count(); }'
```

2. In multicore systems the cpu number can also be provided

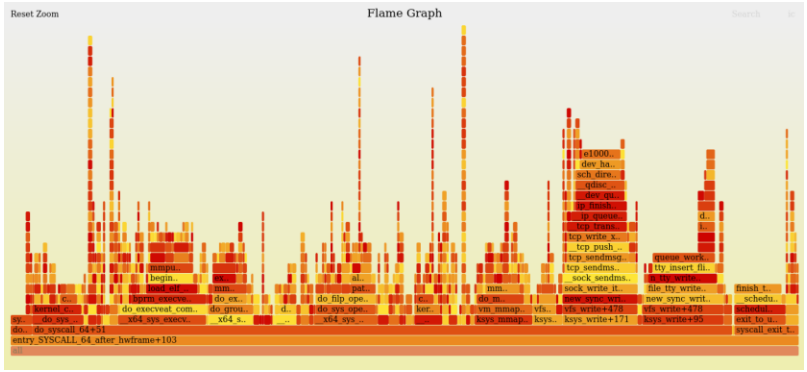
```
sudo bpftrace -e 'profile:hz:99 { @[comm, cpu] = count(); }'
```

3. Profile kernel stacks

```
sudo bpftrace -e 'profile:hz:99 { @[kstack] = count(); }'
```

4. Sampling the CPU at every context switch

```
sudo bpftrace -e 'tracepoint:sched:sched_switch {
@[comm] = count(); }'
```



Tools like FlameGraph allows to visualise the stack traces and easily analyse the result of the CPU profile [*https://github.com/brendangregg/FlameGraph*]

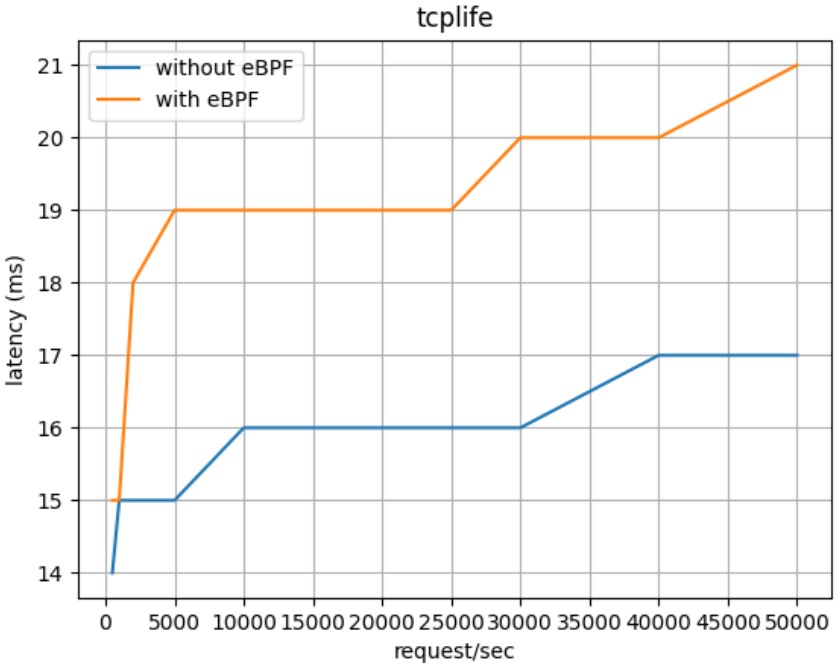
Network Monitoring

`tcplife` trace TPC session lifespans.

- Attached to `kprobe:tcp_set_state`, and triggered at every TCP state change
- On TCP_CLOSE state, it prints all the connections details.

```
vagrant@ebpf:/vagrant$ sudo bpftrace tcplife.bt
Attaching 2 probes...
PID     COMM       LADDR           LPORT RADDR           RPORT  TX_KB RX_KB  MS
568709  curl       ::1             54038 ::1             14592  0     10     3
284     apache2    ::1             80    ::1             26752  10    0      3
568687  sshd       10.0.2.15       22    10.0.2.2        11200  9     4      14948
568726  wget       10.0.2.15       48042 216.58.204.142  26880  0     0      156
568726  wget       10.0.2.15       41230 216.58.204.228  20160  0     20     100
```

Minimal intrusion test: measuring the performaces of a web server with and without eBPF network monitoring.

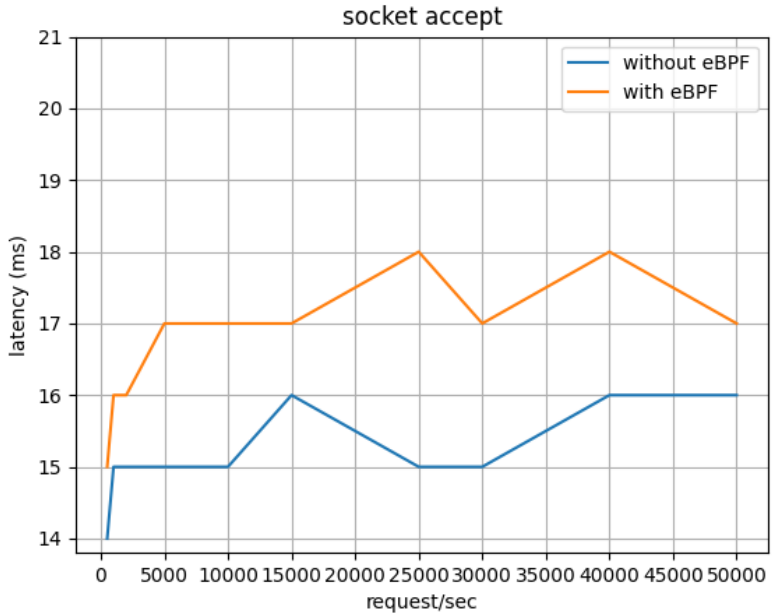


Network Monitoring

`soaccept` trace IP socket accept

- Triggered by `tracepoint:syscalls:sys_enter_accept` and `tracepoint:syscalls:sys_exit_accept`

```
vagrant@ebpf:/vagrant$ sudo bpftrace soaccept.bt
Attaching 6 probes...
PID    PROCESS          ADDRESS          PORT
272    sshd             10.0.2.2         556
284    apache2          ::1              812
284    apache2          ::1              816
```



Minimal intrusion test: measuring the performaces of a web server with and without eBPF monitoring of accept system calls.

- Every tracepoint is passed only once per request

Conclusion

eBPF is becoming a widely adopted solution for observability in distributed systems and in the cloud.

- **Dynamic**: loading and removal of eBPF programs can be done dynamically without rebooting the system
- **Low overhead**: different solutions require higher resource utilization, like service mesh.
- **Totally trasparent** to the applications: everything runs in the kernel