eBPF Tutorial

CS695: Topics in Virtualization and Cloud Computing

Kevin Prafull Baua

(Hard) ways to change the Kernel

1. **Pushing your changes to Linux upstream**
   - Too hard and too long

2. **Re-compiling locally**
   - Need compilation tools, unfeasible to distribute

3. **Kernel Modules**
   - Need kernel knowledge,
     No safety guarantee


YOUR BUGS
RAISE EXCEPTIONS
MINE PANIC
THE KERNEL
WE ARE
NOT THE SAME
imgflip.com

What is eBPF?

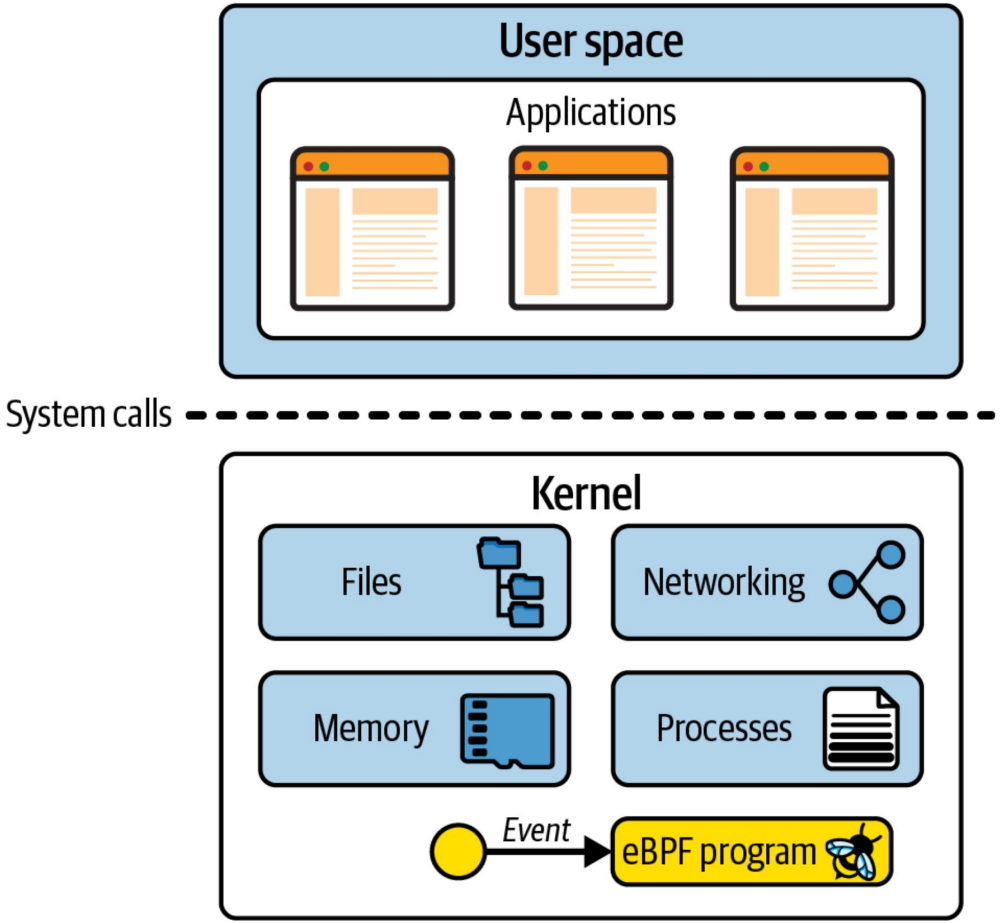
eBPF (extended Berkeley Packet Filter) enables loading custom code into the kernel **dynamically** and **securely**.

No system reboot required!

No Kernel crash!

eBPF program gets loaded into kernel and attaches to an hookpoint

eBPF program is ran each time the hookpoint event is triggered



Source: Fig 1-1 from Learning eBPF by Liz Rice

eBPF features

- Dynamic Loading:
  Starts working as soon as program is loaded and attached.

- Verification and Security:
  eBPF verifier checks the program before loading

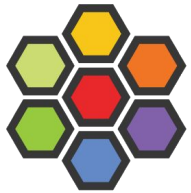
- High performance:
  Can be JIT compiled to run natively.

Origins of eBPF

- eBPF stands for extended Berkeley Packet Filter
- Supported in kernel version 3.18 onwards from 2014
- Derived from the Berkeley Packet Filter, now known as cBPF (classic BPF)
- Since then, eBPF has evolved to do much more than just packet filtering

eBPF application areas

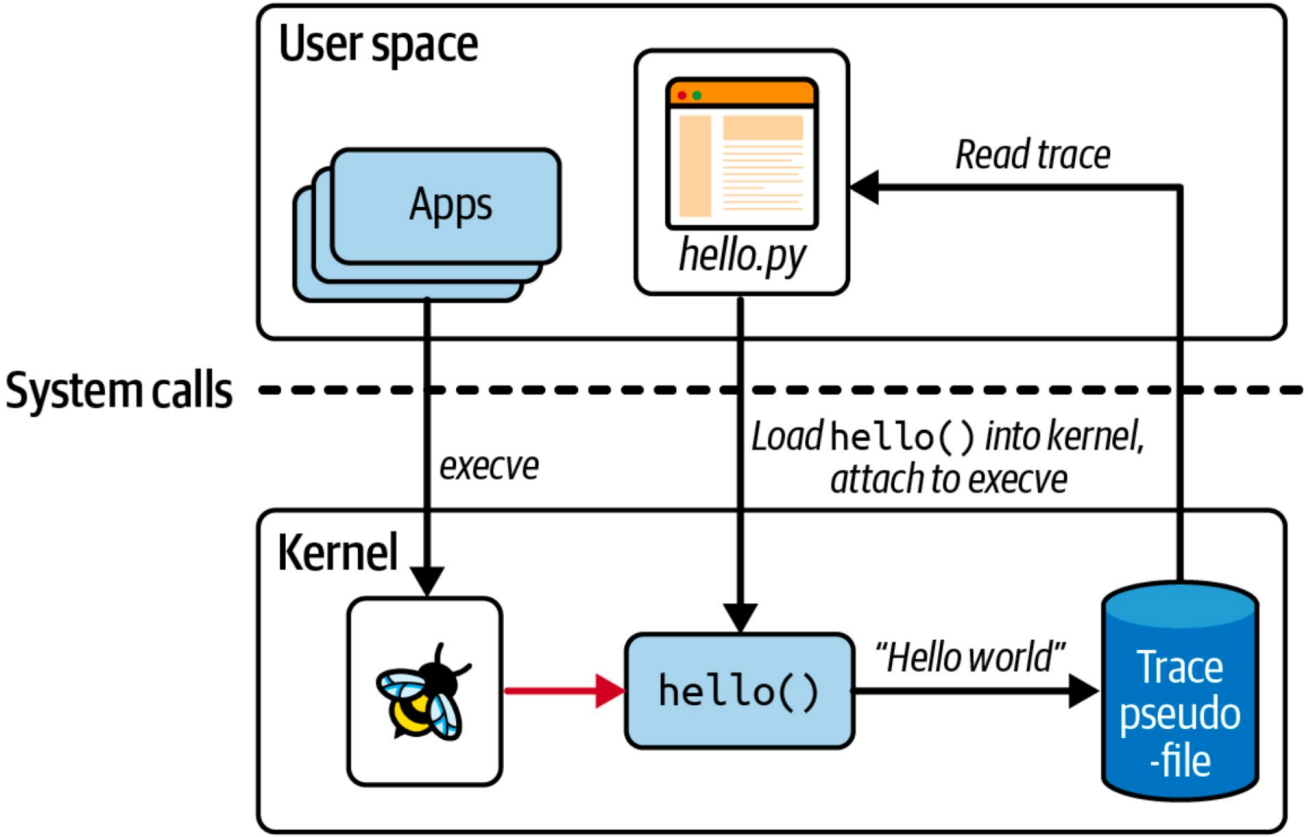
1. **Tracing**
   system call trace, monitor network connections, disk I/O latencies
2. **Networking**
   Packet Filtering, load balancing, header processing
3. **Security**
   Syscall restriction (Seccomp), Disallowing modifications to important kernel data structures (LSM)
4. **Reducing kernel overheads**
   Offloading code and short circuiting paths e.g. XRP
5. **??**



More Info: [eBPF case studies]

eBPF Hello World

- Let’s write simple hello world program using eBPF
- Should print “Hello World” each time evecve() syscall is triggered.



Source: Fig 2-4 from Learning eBPF by Liz Rice

eBPF and Driver Program

eBPF program (Kernelspace code):

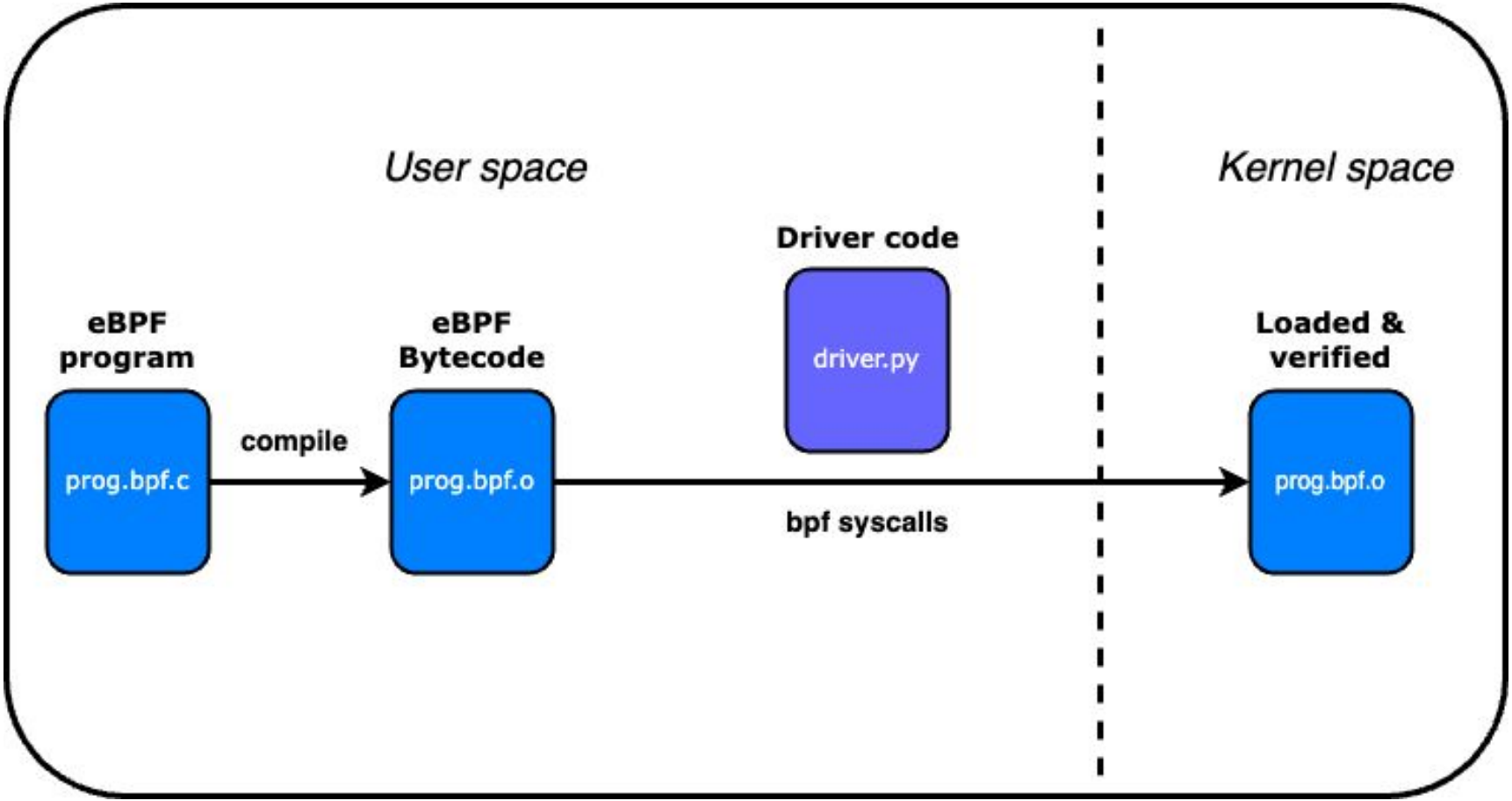
- Actual code that will be run in kernel

- Can be written in C or Rust

Driver program (Userspace code):

- Used to load and attach eBPF program into the kernel using BPF syscalls

- Can be written in any language Python, C, Rust, Go etc


User space
Kernel space
Driver code
eBPF program
eBPF Bytecode
Loaded & verified
driver.py
prog.bpf.c
compile
prog.bpf.o
bpf syscalls
prog.bpf.o

Hello World in BCC

```c
int hello(void *ctx) {
    bpf_trace_printk("Hello World!");
    return 0;
}
```

hello.bpf.c

(eBPF program)

```python
#!/usr/bin/python3
from bcc import BPF

b = BPF(src_file="hello.bpf.c")

syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")

b.trace_print()
```

driver.py

(Driver program)

Output: $ sudo python3 driver.py

```
b'     sh-46728    [006] ...21 435223.474246: bpf_trace_printk: Hello World!'
b'     sh-46730    [006] ...21 435223.476841: bpf_trace_printk: Hello World!'
...
```

bpftool program inspections commands

$ sudo bpftool prog show
$ sudo bpftool prog show name hello --pretty
$ sudo bpftool prog dump xlated name hello

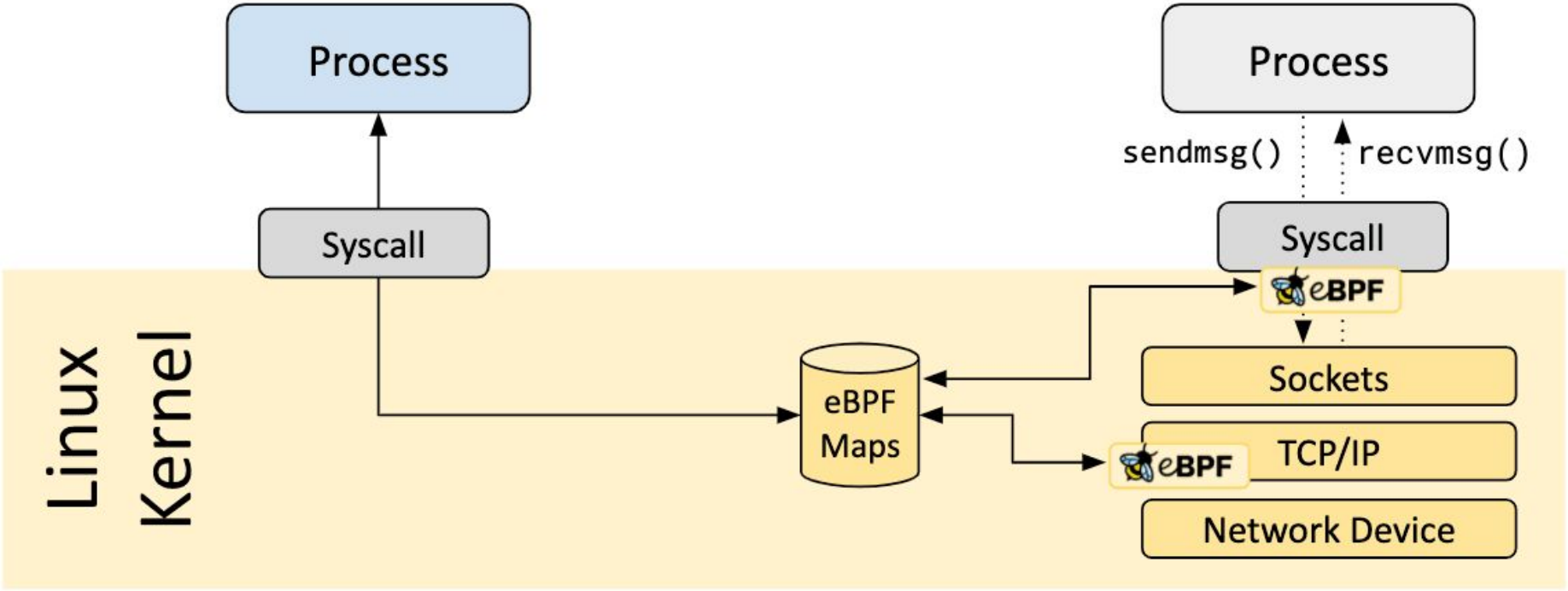
eBPF maps

Used to share data between eBPF programs and userspace.

Usually key value stores but can be specialized data structures.

Global variables are also stored as maps.

Examples BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_RINGBUF



Source: https://ebpf.io/what-is-ebpf/

eBPF maps

Typical usecases:

- Config information from userspace to eBPF program
- Storing state information to be used by next invocation
- Returning results and metrics back to userspace

```
BPF_HASH(counter_table);

int hello(void *ctx) {
    u64 uid;
    u64 counter = 0;
    u64 *p;

    uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
    p = counter_table.lookup(&uid);
    if (p != 0) {
       counter = *p;
    }
    counter++;
    counter_table.update(&uid, &counter);
    return 0;
}
```

Output (No. of execve calls for each uid):

ID 1000: 1
ID 1000: 2
ID 1000: 2
ID 0: 8 ID 1000: 2
ID 0: 9 ID 1000: 2
ID 0: 9 ID 1000: 2
ID 0: 9 ID 1000: 2
ID 0: 9 ID 1000: 2
ID 0: 9 ID 1000: 2
ID 0: 10 ID 1000: 3
.....

bpftool map inspections commands

$ sudo bpftool map show
$ sudo bpftool map dump name counter_table

BCC tool examples

- opensnoop
  Trace open() syscalls.

  $ sudo /usr/sbin/opensnoop-bpfcc

- tcptrace
  Trace TCP established connections (connect(), accept(), close())

  $ sudo /usr/sbin/tcptracer-bpfcc

More tools at:: BCC Github

eBPF Program and Attachment types

Each eBPF program has a program type .
(e.g. BPF_PROG_TYPE_KPROBE)

A program type can have multiple compatible attachment types.

They determine:

- Context provided (e.g. CPU registers)
- Helper functions and kfuncs available (e.g. bpf_trace_printk())
- Return code semantics

eBPF Program and Attachment types

- **Kprobes and Kretprobes:**
  - Used to attach to any instruction in kernel
- **Tracepoints:**
  - Maintained stable hookpoints in kernel
- **XDP:**
  - Intercept packets at driver level before skb is formed
- **Socket related types:**
  - Hooks operating at socket level

XDP (Xpress Data Path)

- Intercepts packets at early on even before any kernel stack processing
- Context:

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    ...
};
```

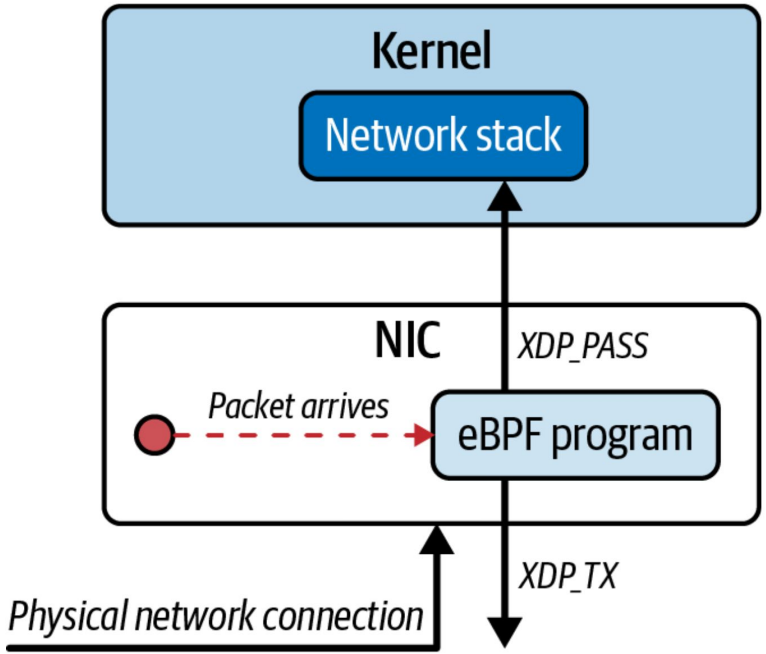
- Return Values:

  XDP_ABORTED, XDP_DROP,

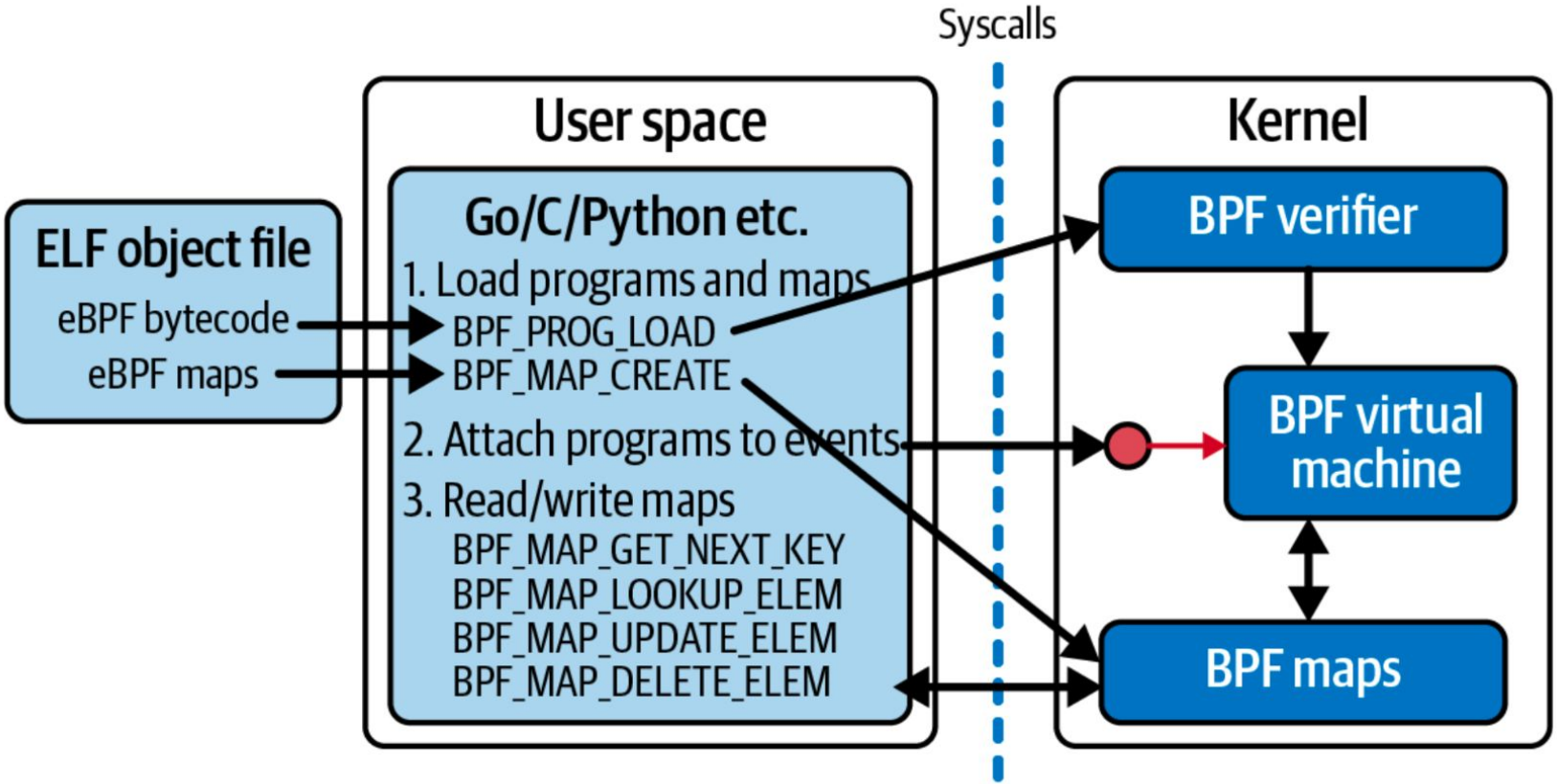
  XDP_PASS, XDP_TX,

  XDP_REDIRECT

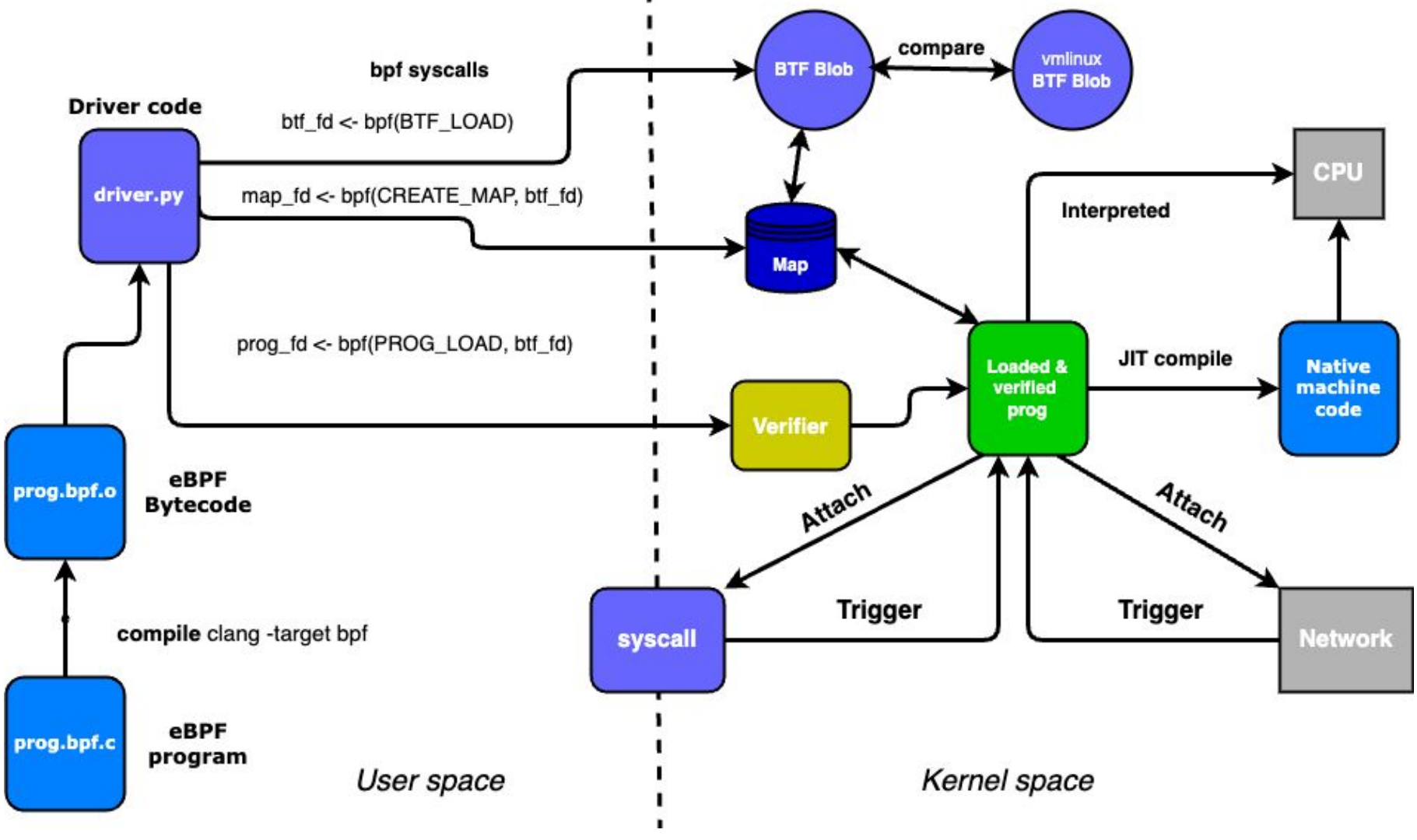
BPF_PROG_TYPE_XDP



Source: Fig 8-3 from Learning eBPF by Liz Rice

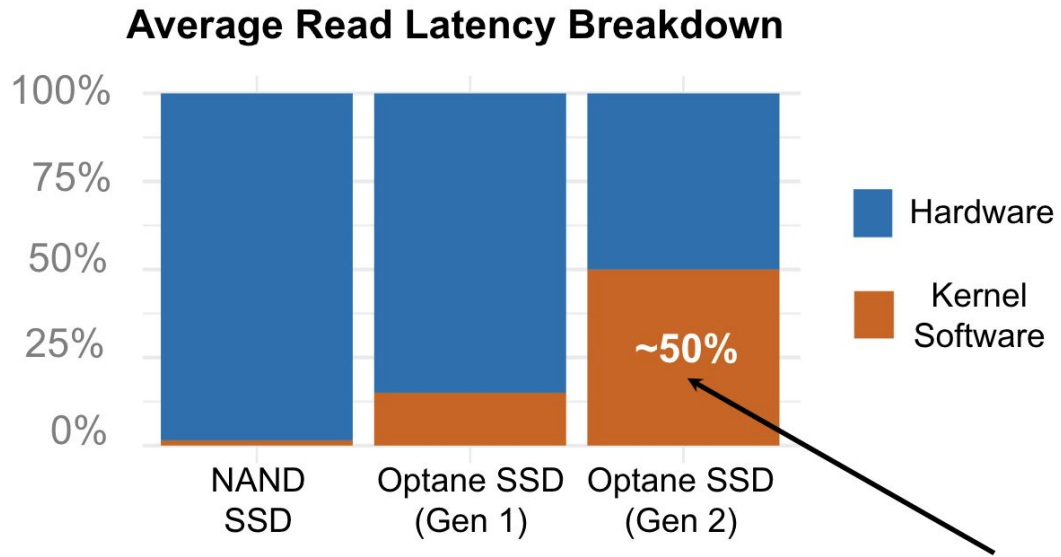


Source: Figure 4-1 from Learning eBPF


bpf syscalls
Driver code
btf_fd <- bpf(BTF_LOAD)
BTF Blob
compare
vmlinux
BTF Blob
driver.py
map_fd <- bpf(CREATE_MAP, btf_fd)
Map
CPU
Interpreted
prog_fd <- bpf(PROG_LOAD, btf_fd)
Verifier
Loaded &
verified
prog
JIT compile
Native
machine
code
prog.bpf.o
eBPF
Bytecode
Attach
Attach
Trigger
Trigger
syscall
Network
compile clang -target bpf
prog.bpf.c
eBPF
program
User space
Kernel space

XRP: In-Kernel Storage Functions with eBPF
(OSDI '22 Best Paper Award)

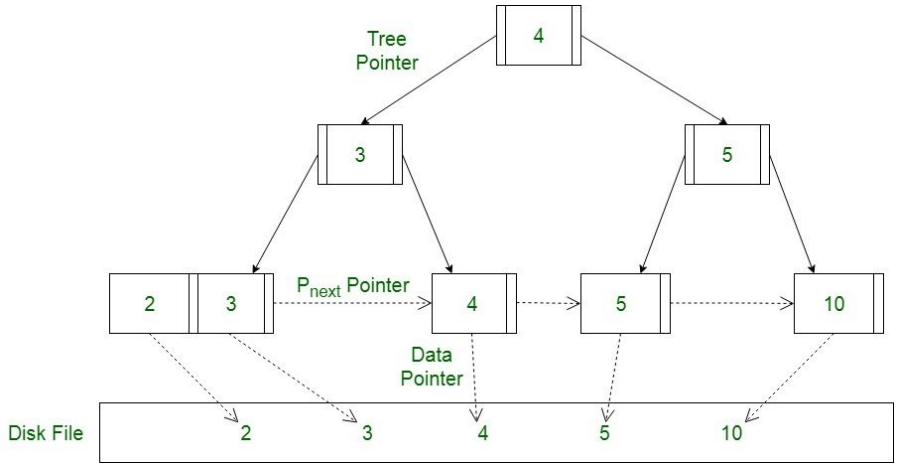
Kernel Software is Becoming the Bottleneck for Storage



Kernel software overhead accounts for ~50% of read latency on Optane SSD Gen 2

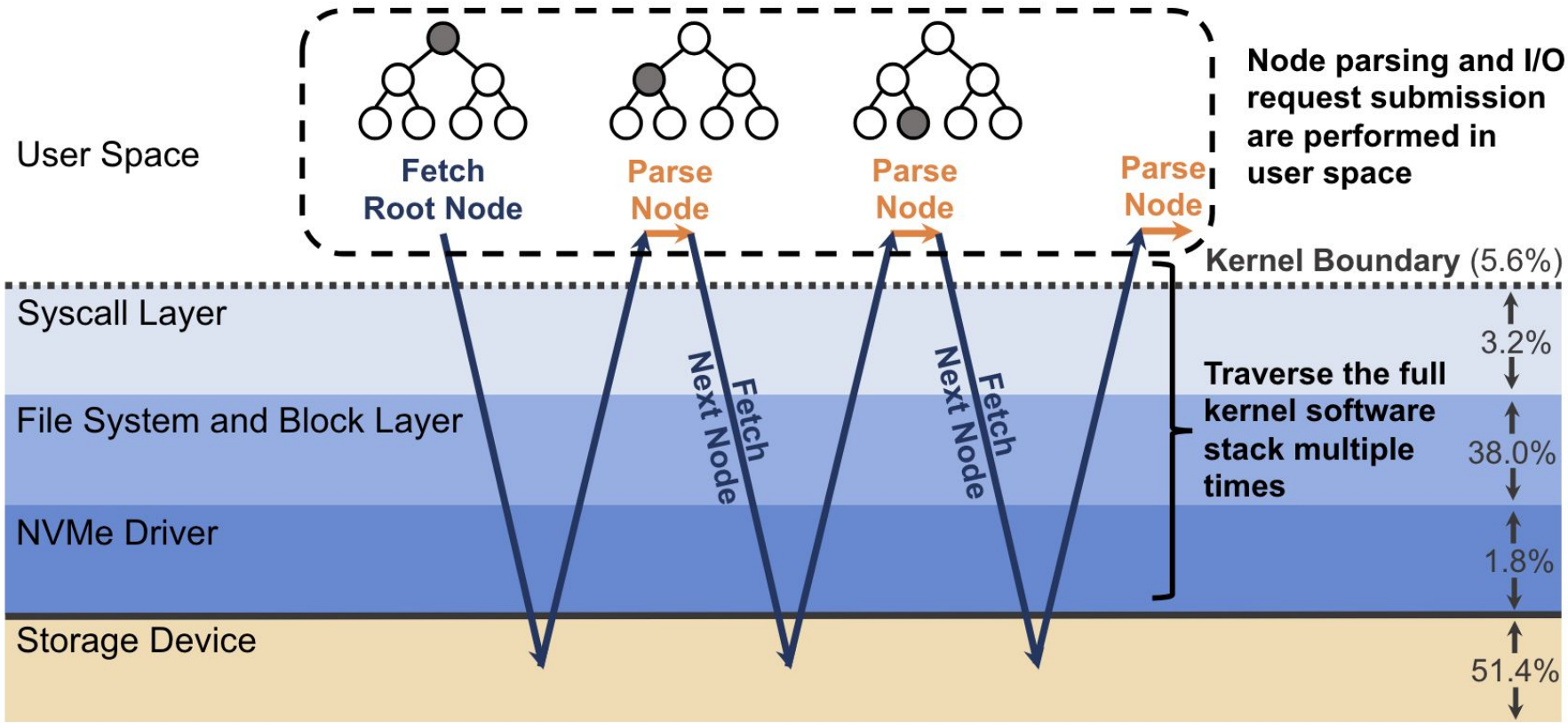
Key value store using B+ trees

- B+ trees are usually used to implement Database and file systems
- The tree needs to be traversed from root to leaf to get the value corresponding to given key
- Since the tree lies in a file on a disk, getting each node requires a costly disk access



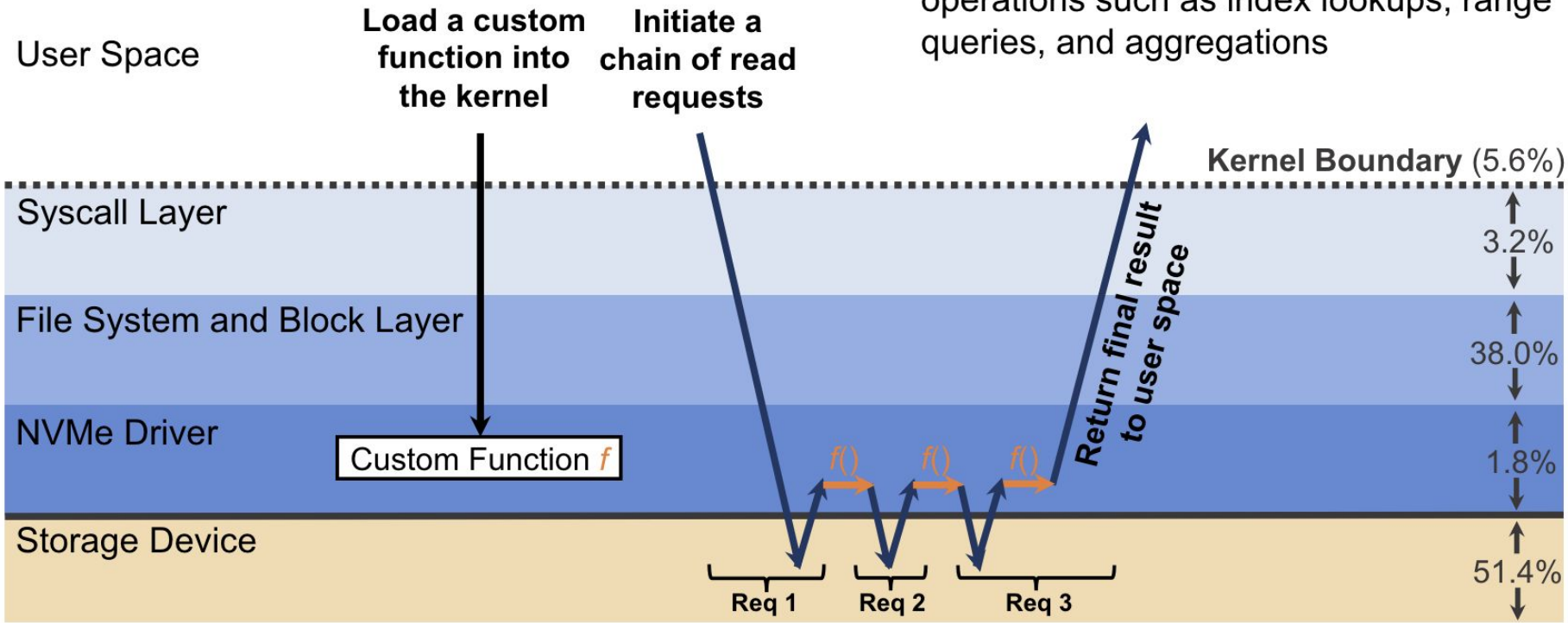
Source: GFG

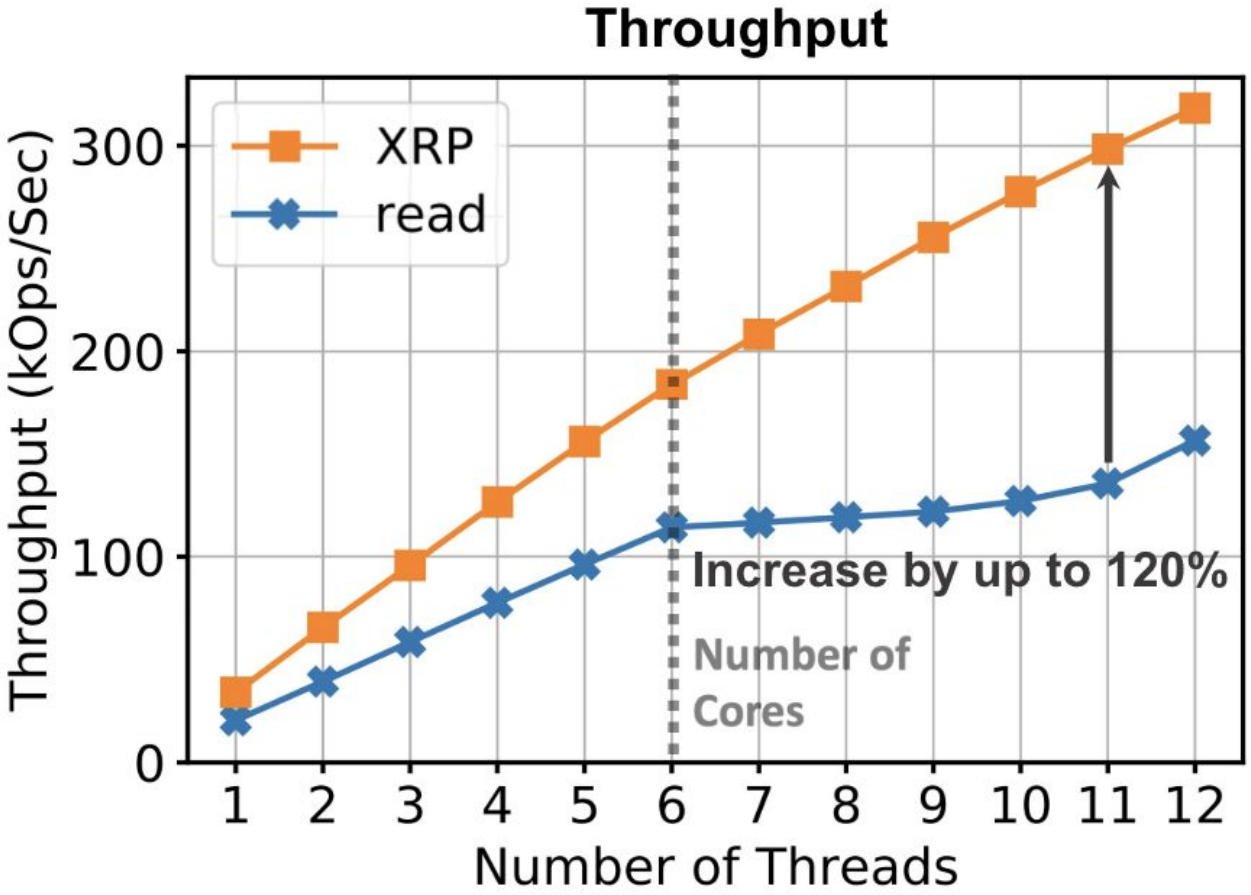
B+ Tree Index Lookup from User Space



XRP: A Framework for In-Kernel Storage Functions

XRP can accelerate many types of operations such as index lookups, range queries, and aggregations




Throughput
Throughput (kOps/Sec)
300
200
100
0
XRP
read
Increase by up to 120%
Number of
Cores
1
2
3
4
5
6
7
8
9
10
11
12
Number of Threads

Cilium

Cilium is an open source project to provide networking, security, and observability for **cloud native environments** such as Kubernetes clusters and other container orchestration platforms.

- Kube proxy replacement
- Firewall
- Enforcing network policies
- Metrics and tracing

Load balancing in Kubernetes

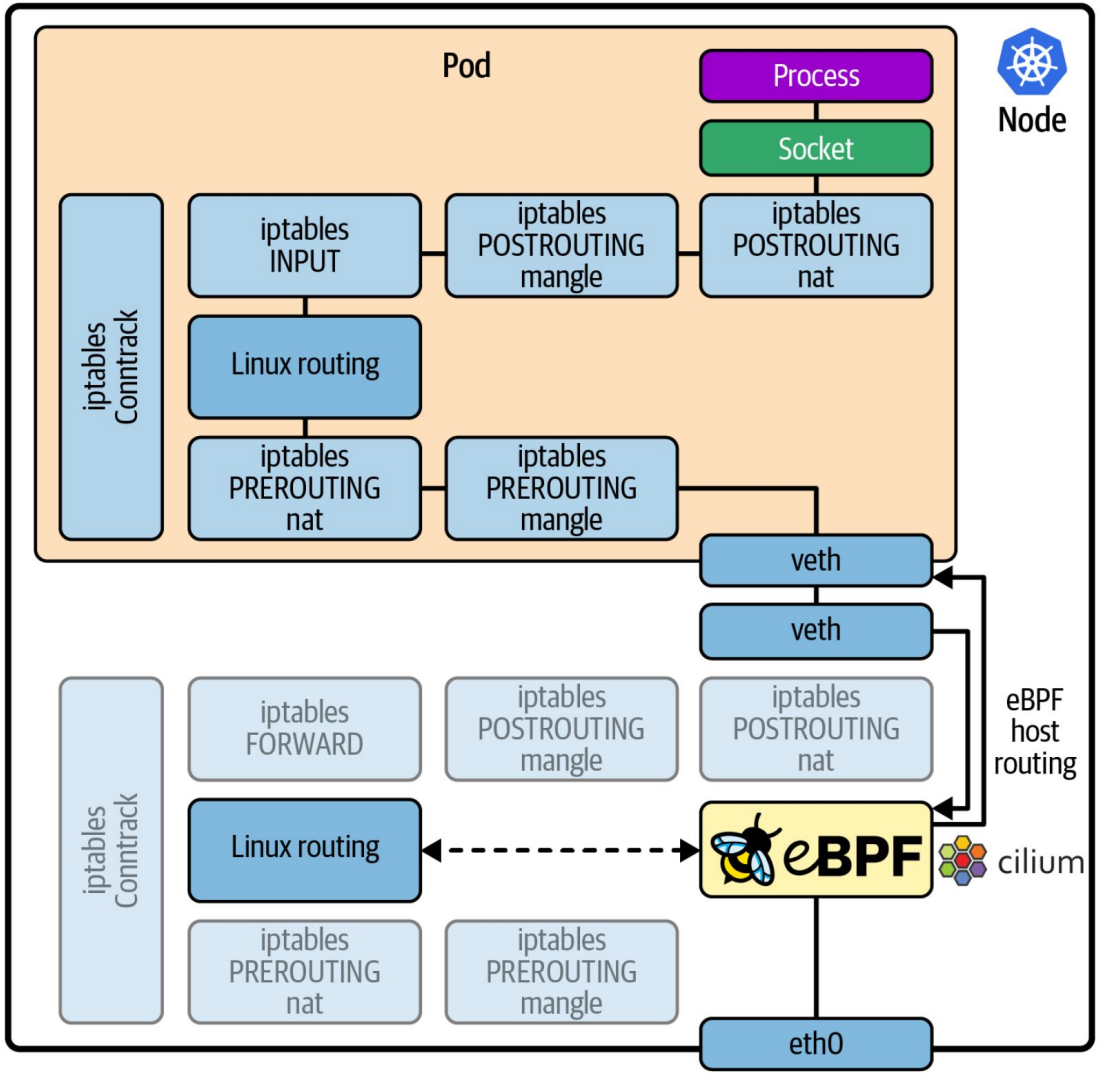
- K8s runs services in pods with each pod typically have separate IP address
- K8s actions might include adding new pods or changing load balancing rules

Kube proxy

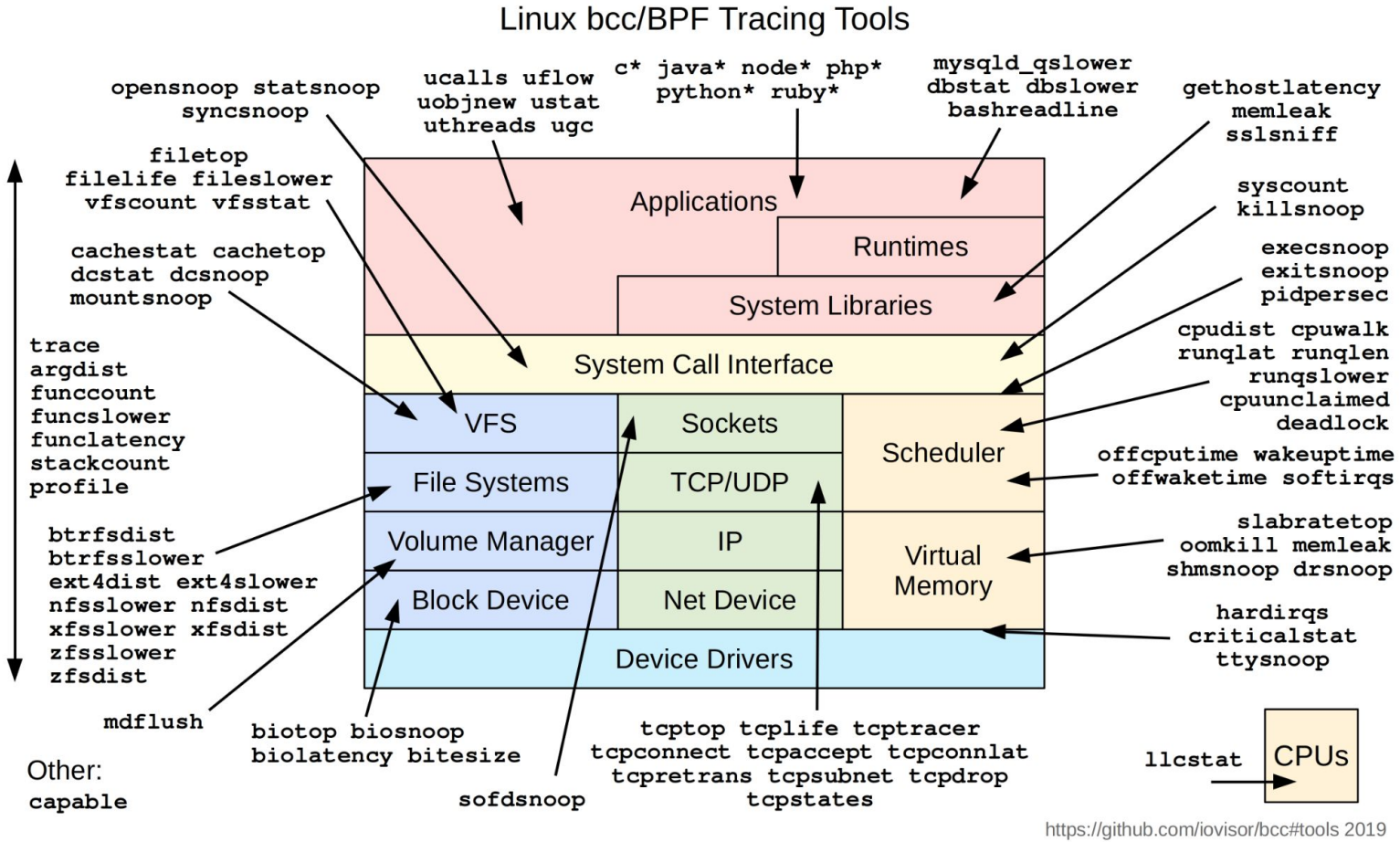
- Implements load balancing through IP tables
- Requires iptable update when pod IP changes
- Slow O(n) lookup and addition

Cilium

- Uses eBPF to encode routing logic
- Efficient O(1) logic lookup using eBPF hash maps



Source: Fig 8-6 from Learning eBPF by Liz Rice



Source: https://www.brendangregg.com/ebpf.html

Thanks

Links:

1. Learning eBPF by Liz Rice: https://www.oreilly.com/library/view/learning-ebpf/9781098135119/
2. BCC github https://github.com/iovisor/bcc/tree/master
3. eBPF docs: https://docs.ebpf.io/
4. bpftool tutorial https://qmonnet.github.io/whirl-offload/2021/09/23/bpftool-features-thread/