eBPF Implementation for FreeBSD

Yutaro Hayakawa

Mail: yhayakawa3720@gmail.com
Twitter: @YutaroHayakawa

About me

Name: Yutaro Hayakawa

Affiliation: Keio University, Japan (Master student)

Research topic: Network (SDN/NFV), Operating Systems

Misc: Now on GSoC for FreeBSD and job hunting

Agenda

1. Linux eBPF the Basic
2. eBPF implementation for FreeBSD
3. Usecase: VALE-BPF

Agenda

1. Linux eBPF the Basic
2. eBPF implementation for FreeBSD
3. VALE-BPF

What’s eBPF?

Extended general perpose BPF virtual machine ISA

- Closer to modern CPU ISA (64bit registers * 11, 64bit wide instructions...)
- C calling convention and LLVM backend
- Call instruction
    - Maps (in-kernel key-value store shared with user space program)
    - Write data to tracing buffer
    - etc…

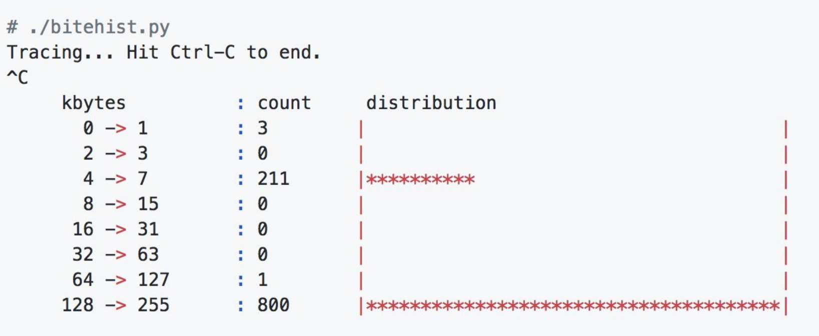
More performance optimization (JIT, static code analysis)

bpf(2) for loading program, creating maps, manipulating maps ...

Use cases?

Use cases: Dynamic tracing

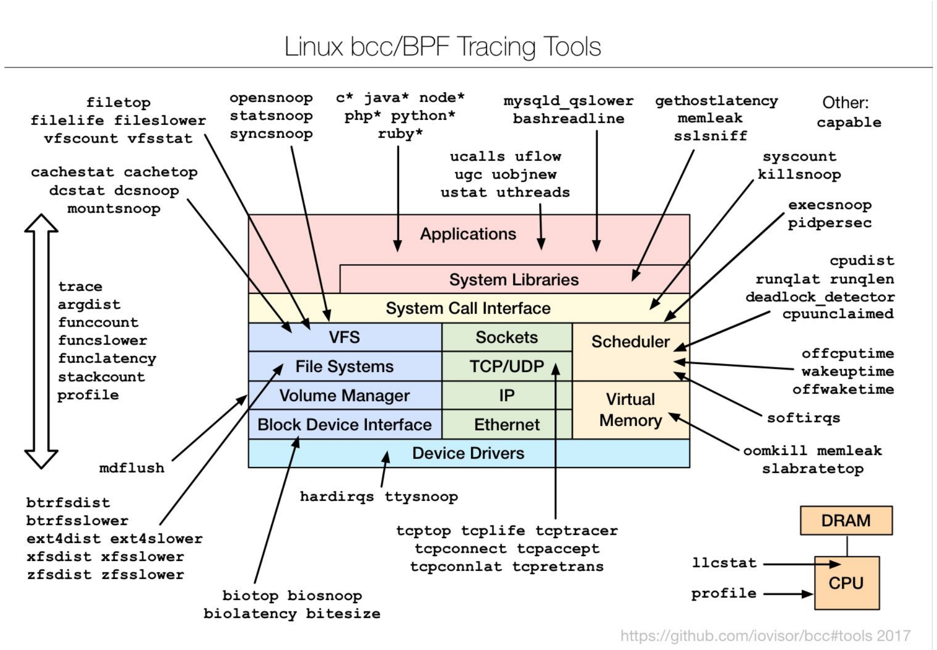
Use eBPF as a backend of dynamic tracing (like DTrace)



```
# ./bitehist.py
Tracing... Hit Ctrl-C to end.
^C
     kbytes               : count     distribution
       0 -> 1             : 3        |                                        |
       2 -> 3             : 0        |                                        |
       4 -> 7             : 211      |**********                              |
       8 -> 15            : 0        |                                        |
      16 -> 31            : 0        |                                        |
      32 -> 63            : 0        |                                        |
      64 -> 127           : 1        |                                        |
     128 -> 255           : 800      |****************************************|
```

https://github.com/iovisor/bcc

http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html



Linux bcc/BPF Tracing Tools

https://github.com/iovisor/bcc#tools 2017

Use cases: XDP (eXpress Data Path)

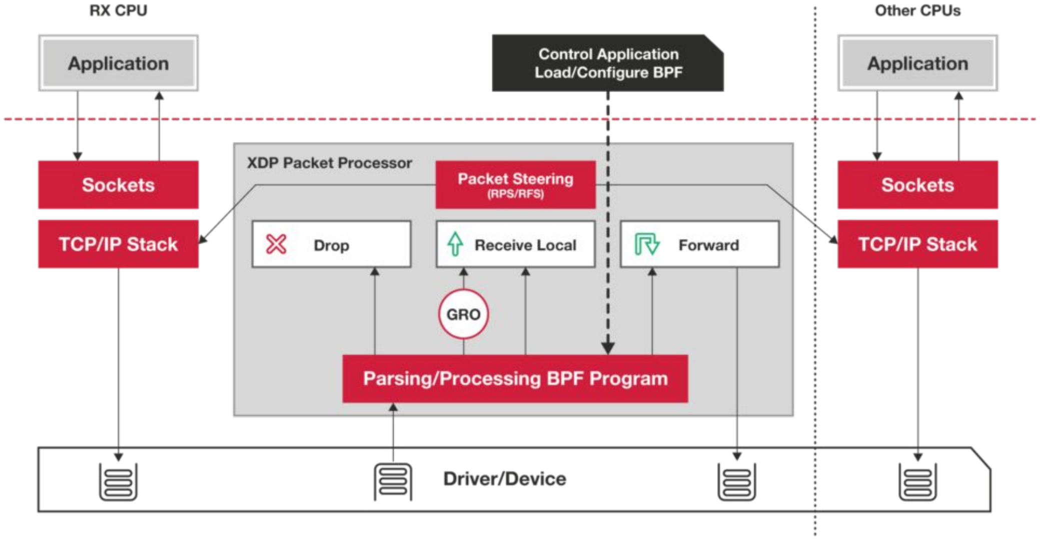
No “kernel bypass” (e.g. DPDK, netmap)

Hook and process packet right after reception inside the driver by eBPF

- DDos mitigation: Droplet
- Load balancing: Katran
- IDS/IPS backend: Surikata

Hardware offloading

- Netronome Agilio



https://www.iovisor.org/technology/xdp

Tooling?

eBPF Tooling

Linux kernel provides only very premitive API to users

- bpf(2)
- Program loader (e.g. Netlink, setsockopt, ioctl...)
- Some useful libraries (but very primitive)

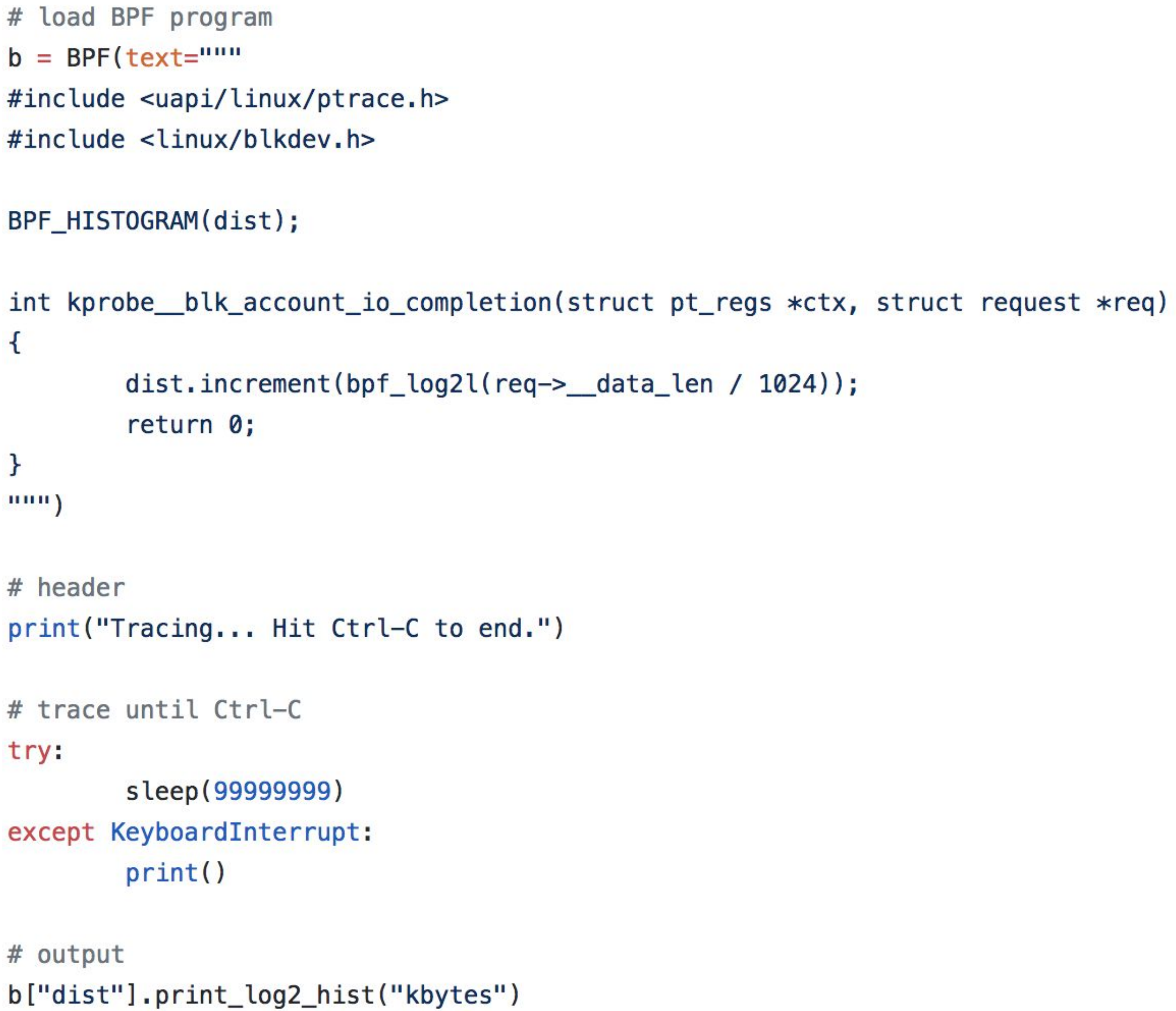
Need tooling for better utilization

Tooling: BCC (BPF Compiler Collection)

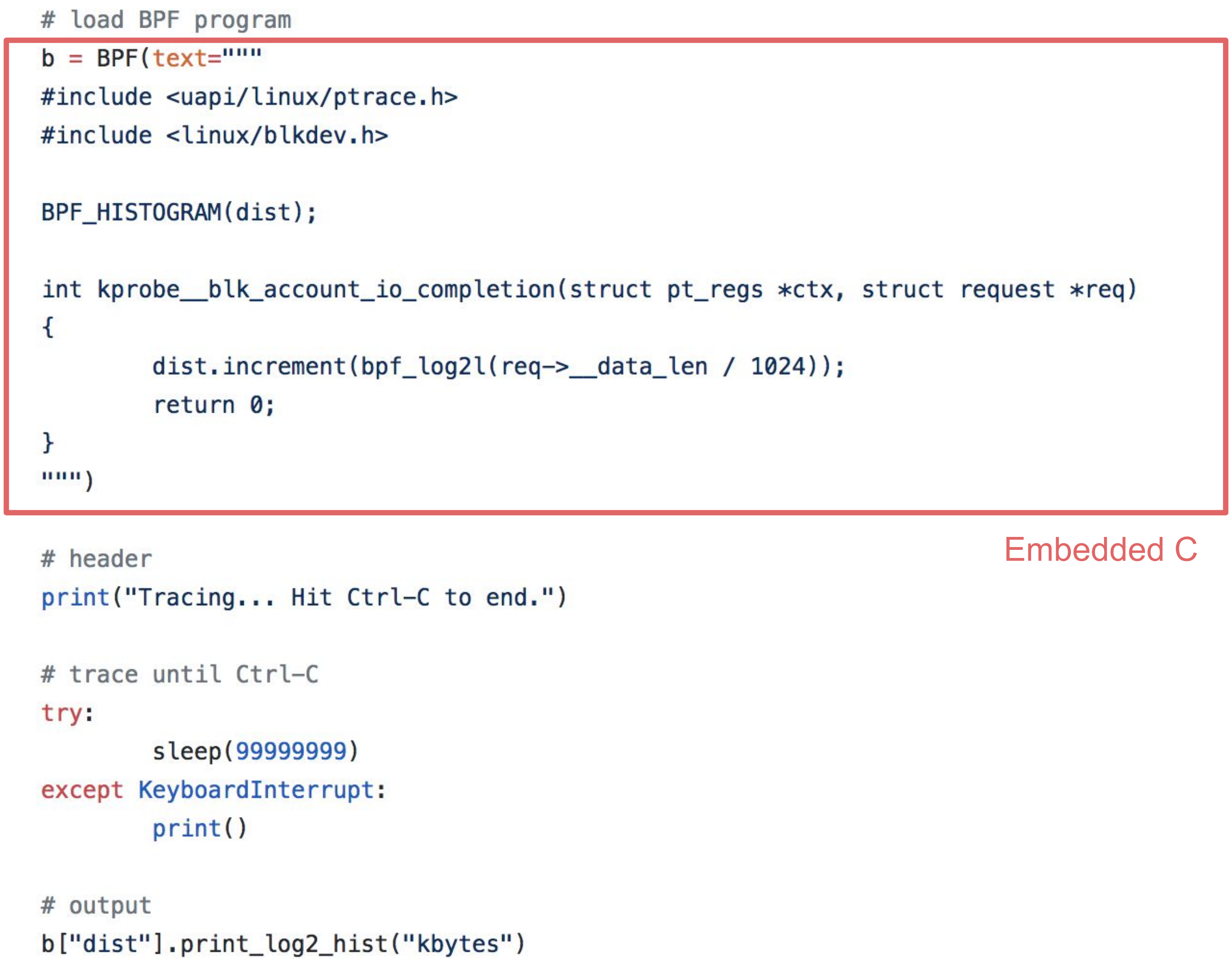
Compiler driver and useful libraries for eBPF

- Deal with restricted C, call clang/llvm
- Compiler frontend for various languages (C, P4)
- ELF parsing, Map libraries
- Language bindings (Python, C++, Lua…)

Source: https://github.com/iovisor/bcc

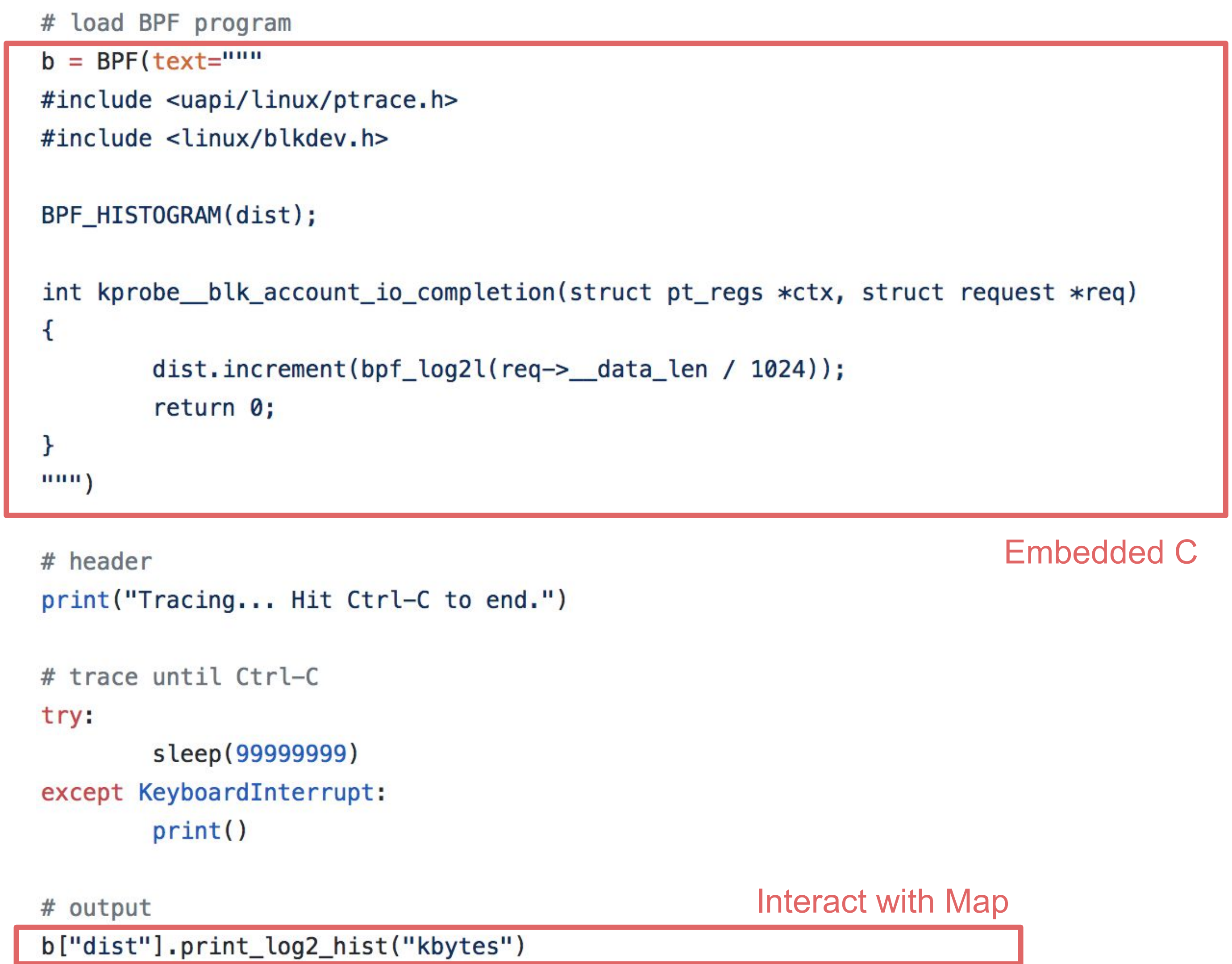


```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>

BPF_HISTOGRAM(dist);

int kprobe__blk_account_io_completion(struct pt_regs *ctx, struct request *req)
{
        dist.increment(bpf_log2l(req->__data_len / 1024));
        return 0;
}
""")

# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
        sleep(99999999)
except KeyboardInterrupt:
        print()

# output
b["dist"].print_log2_hist("kbytes")
```



```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>

BPF_HISTOGRAM(dist);

int kprobe__blk_account_io_completion(struct pt_regs *ctx, struct request *req)
{
        dist.increment(bpf_log2l(req->__data_len / 1024));
        return 0;
}
""")

# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
        sleep(99999999)
except KeyboardInterrupt:
        print()

# output
b["dist"].print_log2_hist("kbytes")
```

Embedded C



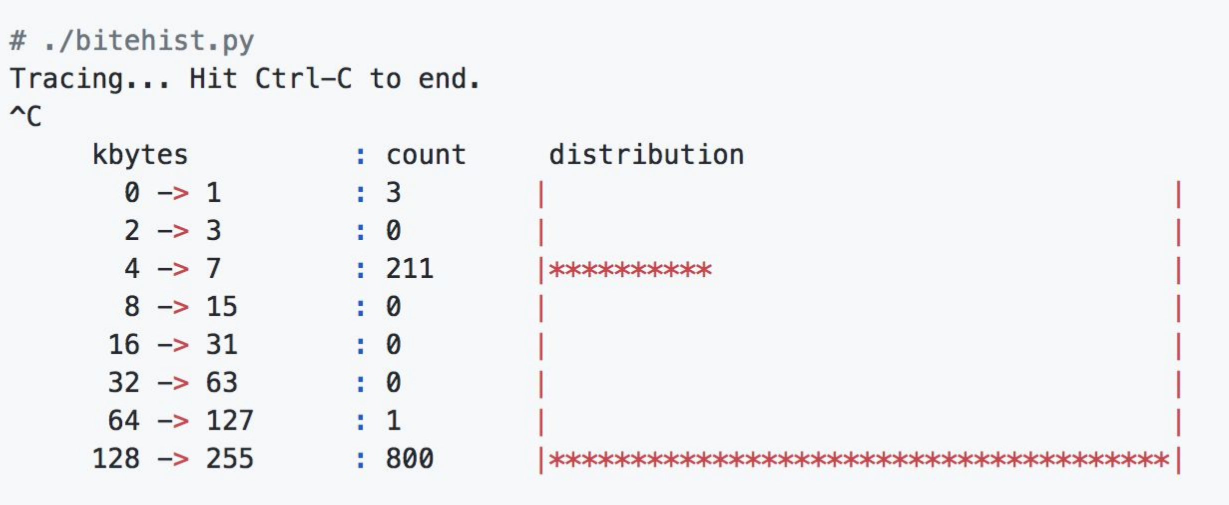
```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>

BPF_HISTOGRAM(dist);

int kprobe__blk_account_io_completion(struct pt_regs *ctx, struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""")

# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print()

# output
b["dist"].print_log2_hist("kbytes")
```

Embedded C

Interact with Map

```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
```

Output



```
# ./bitehist.py
Tracing... Hit Ctrl-C to end.
^C
     kbytes          : count     distribution
       0 -> 1        : 3        |                                      |
       2 -> 3        : 0        |                                      |
       4 -> 7        : 211      |**********                            |
       8 -> 15       : 0        |                                      |
      16 -> 31       : 0        |                                      |
      32 -> 63       : 0        |                                      |
      64 -> 127      : 1        |                                      |
     128 -> 255      : 800      |**************************************|
```

```
except KeyboardInterrupt:
        print()

# output
b["dist"].print_log2_hist("kbytes")
```

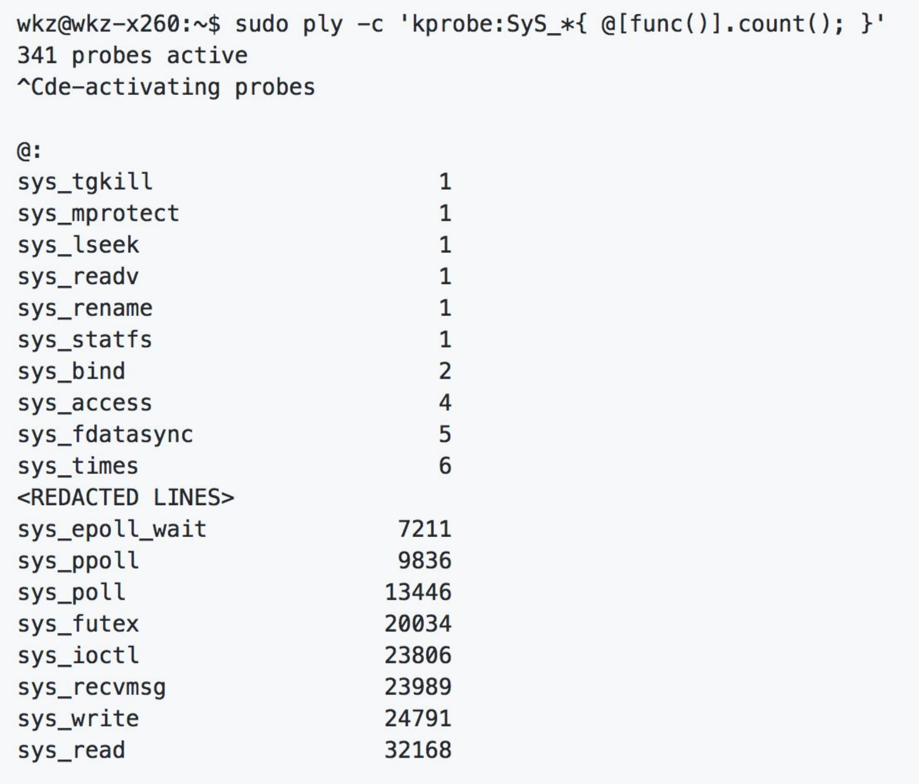
Interact with Map

Tooling: PLY

Tracing frontend which is heavily inspired by DTrace

dtrace -n syscall:::entry'{@syscalls[probefunc] = count();}'

Source: https://github.com/iovisor/ply



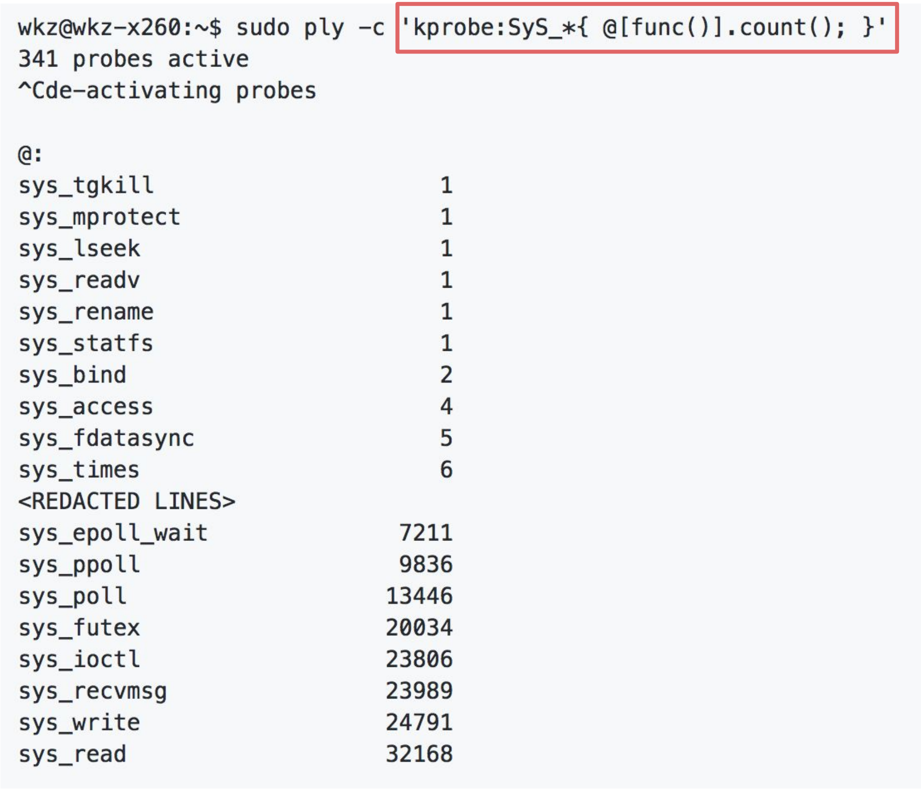
```
wkz@wkz-x260:~$ sudo ply -c 'kprobe:SyS_*{ @[func()].count(); }'
341 probes active
^Cde-activating probes

@:
sys_tgkill                     1
sys_mprotect                   1
sys_lseek                      1
sys_readv                      1
sys_rename                     1
sys_statfs                     1
sys_bind                       2
sys_access                     4
sys_fdatasync                  5
sys_times                      6
<REDACTED LINES>
sys_epoll_wait              7211
sys_ppoll                   9836
sys_poll                   13446
sys_futex                  20034
sys_ioctl                  23806
sys_recvmsg                23989
sys_write                  24791
sys_read                   32168
```

Tooling: PLY

Tracing frontend which is heavily inspired by DTrace

dtrace -n syscall:::entry'{@syscalls[probefunc] = count();}'

Source: https://github.com/iovisor/ply



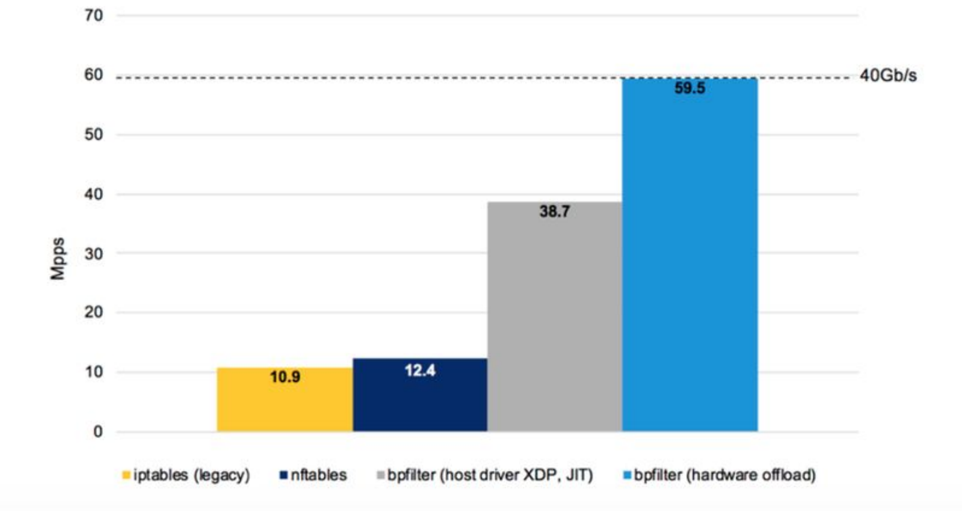
```
wkz@wkz-x260:~$ sudo ply -c 'kprobe:SyS_*{ @[func()].count(); }'
341 probes active
^Cde-activating probes

@:
sys_tgkill                     1
sys_mprotect                   1
sys_lseek                      1
sys_readv                      1
sys_rename                     1
sys_statfs                     1
sys_bind                       2
sys_access                     4
sys_fdatasync                  5
sys_times                      6
<REDACTED LINES>
sys_epoll_wait              7211
sys_ppoll                   9836
sys_poll                   13446
sys_futex                  20034
sys_ioctl                  23806
sys_recvmsg                23989
sys_write                  24791
sys_read                   32168
```

Tooling: bpfilter

iptables (Linux’s ipfw or pf) which uses XDP as a backend

Transparently accerelates existing iptables

RFC patch: https://www.mail-archive.com/netdev@vger.kernel.org/msg217095.html



https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/

Conclusion for this section

Recent Linux implements a lot of interesting features using eBPF

- Dynamic tracing
- Very fast packet processing framework
- etc ...

The community also introduces a lot of interesting tools

- BCC, PLY, bpfilter

More information

- https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/
- Really useful collection of links

Agenda

1. Linux eBPF the Basic
2. eBPF implementation for FreeBSD
3. VALE-BPF

generic-ebpf

Generalized multi-platform eBPF implementation

- Currently supports FreeBSD user/kernel, Linux user/kernel and macOS user
  - About 200 lines of glue code for each platform
  - Shares most of the code (easy to test in userspace)
- Interpreter and JIT compiler for x86-64 based on ubpf
- Maps which uses tommyds as a backend
- Verifier is not yet implemented...

Source: https://github.com/YutaroHayakawa/generic-ebpf

Current status

/dev/ebpf + ioctl(2) interface (Linux bpf(2))

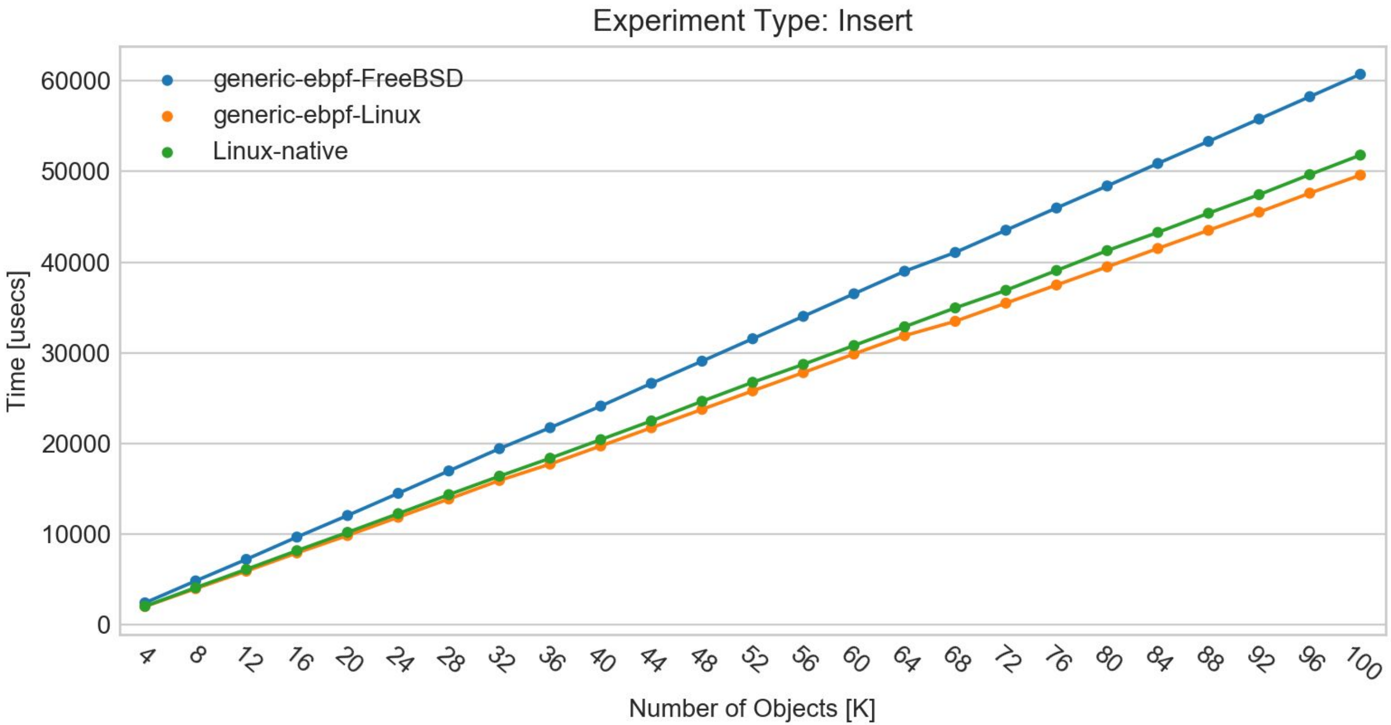
- load program, create and manipulate maps, run simple test

Interpreter and JIT compiler for x86-64

- Most of the instructions are implemented
  - atomic operations are missing

Array, Hashtable maps

Hashtable map benchmark

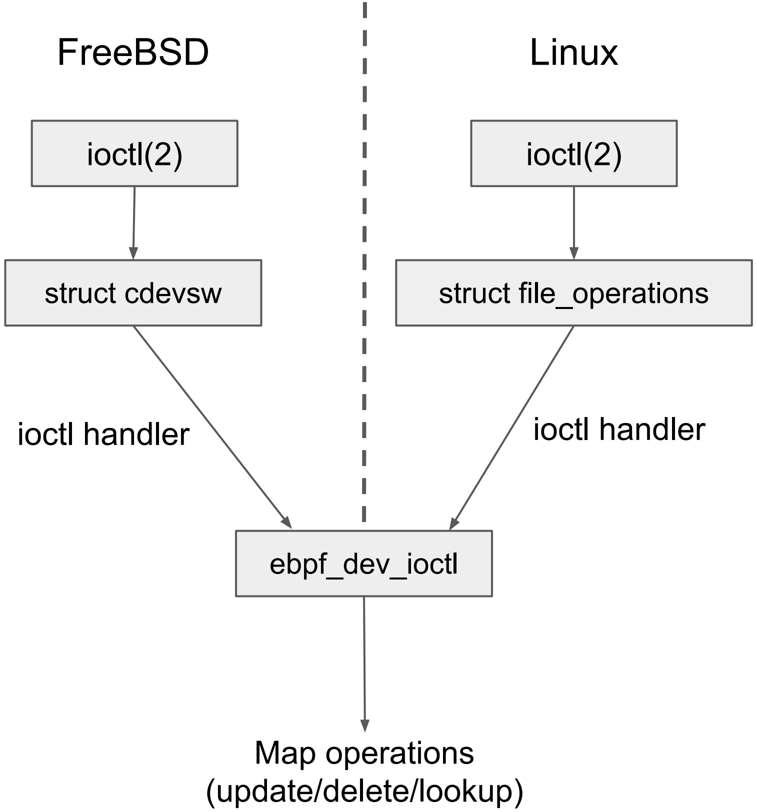


For more details: **https://github.com/YutaroHayakawa/generic-ebpf/tree/master/benchmark**

Why is FreeBSD case so slow?

Experiment

- Simply returns immediately from ioctl handler
- See latency of ioctl



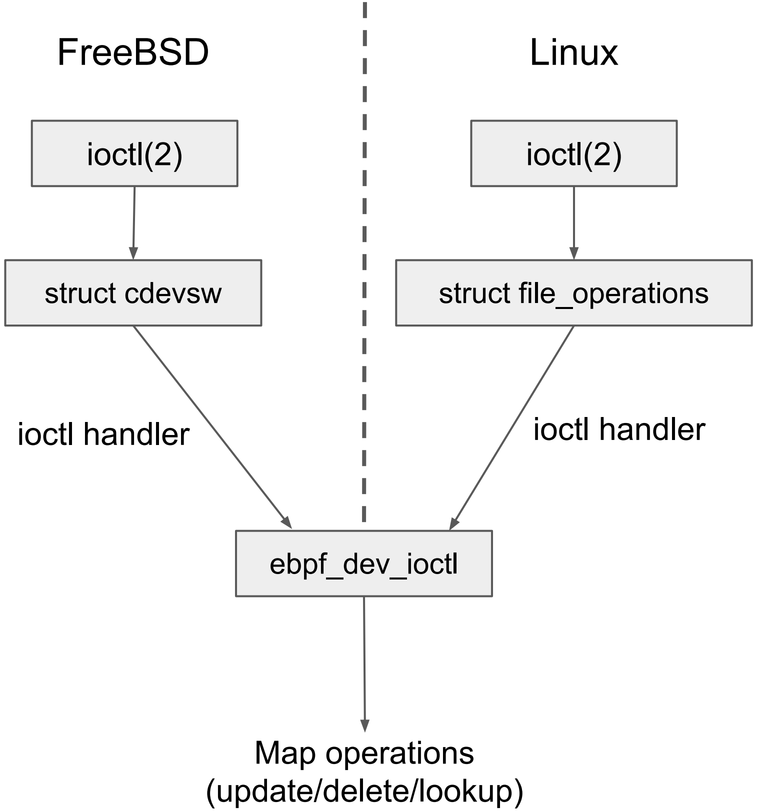
Why is FreeBSD case so slow?

Experiment

- Simply returns immediately from ioctl handler
- See latency of ioctl

About 85% of the difference comes from ioctl

Need more precise analysis...



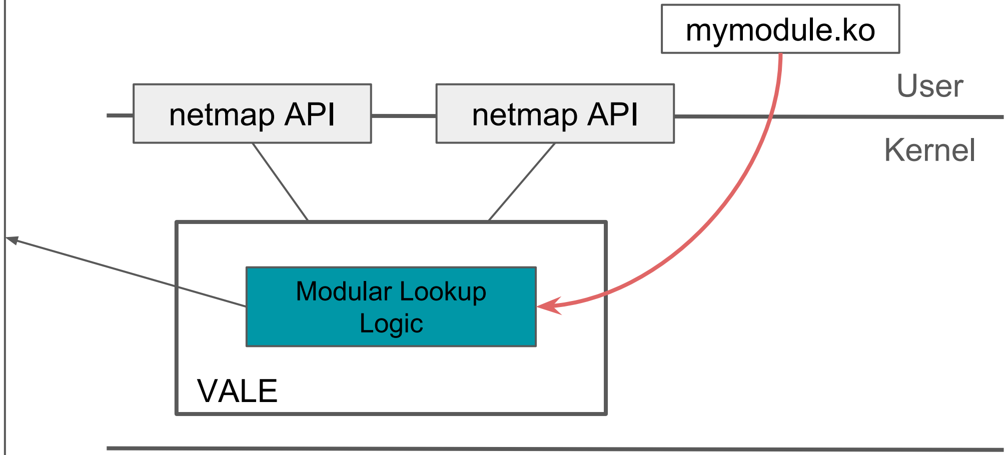
Agenda

1. Linux eBPF the Basic
2. eBPF implementation for FreeBSD
3. VALE-BPF

VALE (Virtual Local Ethernet)

Fast and modular software switch (a.k.a mSwitch)

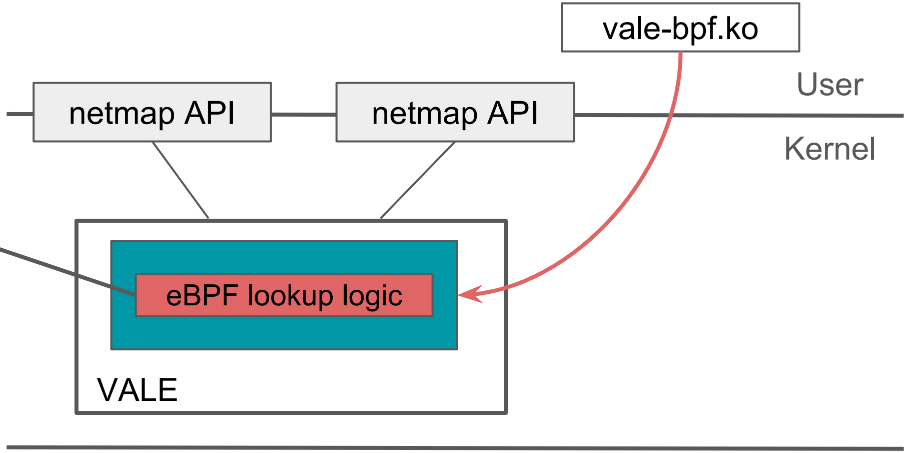
```
uint32_t
mylookup(struct nm_bdg_fwd *ft, uint8_t *dst_ring,
    struct netmap_vp_adapter *na, void *private_data)
{
    struct ip *iph;

    iph = (struct ip)(buf + ETHER_HDR_LEN);
    if (iph - ft->ft_buf > ft->ft_len) {
        return NM_BDG_DROP;
    }

    return ntohl(iph->ip_dst) & 0xff;
}
```



VALE-BPF

VALE module which enhances eBPF programmability to VALE

```
uint32_t
vale_bpf_lookup(struct vale_bpf_md *md)
{
    struct ip iph;

    iph = (struct ip)(md->buf + ETHER_HDR_LEN);
    if (iph > md->buf_end) {
        return VALE_BPF_DROP;
    }

    return ntohl(iph->ip_dst) & 0xff;
}
```



Source: https://github.com/YutaroHayakawa/vale-bpf

Performance evaluation

Forward packets between two virtual ports with different logic

- Learning bridge
- No logic

	Learning Bridge [Mpps]	No Logic [Mpps]
VALE	17.74	27.71
VALE-BPF	8.52	23.66

For more details: https://docs.google.com/document/d/1rdrHIeap8gYRh3es4yCnuWkuA6zDDot4UDFgEyiuG3E/edit?usp=sharing

Demo

Miscellaneous ideas

Networking

- ng_ebpf: Netgraph module for eBPF
- XDP emulator: Compatibility with XDP program
- Hardware offloading

Security

- Systemcall filtering like seccomp

Sammary

1. eBPF is a hot technology among Linux community and they introduce a lot of interesting features and useful tools around that

2. eBPF implementation for FreeBSD is going on

3. VALE-BPF, a extension module which enhances eBPF programmability to VALE switch improves the programmability of VALE switch

Questions?