• Paris, 2019-05-16

All-Out Programmability in Linux
•
An Introduction to BPF as a Monitoring Tool

Quentin Monnet

<quentin.monnet@netronome.com>
@qeole

NETRONOME

Speaker

NETRONOME

Quentin Monnet

- Fast networking: 6WIND then Netronome (since 2017)
- Based in Cambridge, UK
- Working on BPF for over 3 years

Netronome

- Fabless semiconductor company, specialised in “SmartNICs”: Multicore, massively parallel, fully programmable NPUs
- ~200 people, USA/South Africa/UK
- Hardware offloads for several advanced networking features: Open vSwitch, P4… **BPF**

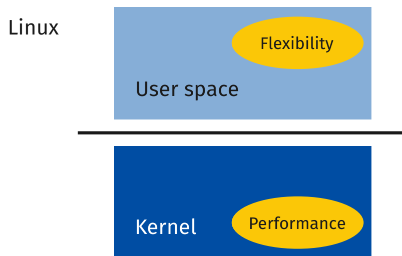
Agenda

- What is BPF?
- Using BPF for tracing, monitoring
- Other use cases for BPF
- Q&A

BPF History and Architecture

Linux: kernel and user space

- Kernel goes fast, lacks flexibility
- User space programmable, no direct access to kernel structures

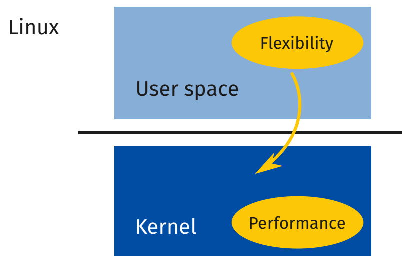


Kernel developers design well-bounded frameworks

Performance and Programmability

Linux: kernel and user space

- Kernel goes fast, lacks flexibility
- User space programmable, no direct access to kernel structures



Get out of the box: **Can we have programmability in the kernel?**

Example: tcpdump

```
# tcpdump -i eth0 tcp dst port 22 -d
(000) ldh      [12]                             # Ethertype
(001) jeq      #0x86dd          jt 2    jf 6    # is IPv6?
(002) ldb      [20]                             # IPv6 next header field
(003) jeq      #0x6             jt 4    jf 15   # is TCP?
(004) ldh      [56]                             # TCP dst port
(005) jeq      #0x16            jt 14   jf 15   # is port 22?
(006) jeq      #0x800           jt 7    jf 15   # is IPv4?
(007) ldb      [23]                             # IPv4 protocol field
(008) jeq      #0x6             jt 9    jf 15   # is TCP?
(009) ldh      [20]                             # IPv4 flags + frag. offset
(010) jset     #0x1fff          jt 15   jf 11   # fragment offset is != 0?
(011) ldxb     4*([14]&0xf)                     # x := 4 * header_length (words)
(012) ldh      [x + 16]                         # TCP dst port
(013) jeq      #0x16            jt 14   jf 15   # is port 22?
(014) ret      #262144                          #  trim to 262144 bytes, return packet
(015) ret      #0                               # drop packet
```

Filtering packets in kernel, to avoid useless copies to user space

This is a BPF program! tcpdump → libpcap → BPF bytecode → kernel

Classic BPF (cBPF)

NETRONOME

Berkeley Packet Filter

- 1993: BPF on BSD, for packet filtering (by Van Jacobson)
- 1997: ported to Linux

Architecture

- In-kernel virtual machine
- 32-bit instructions, 2 registers (32-bit)
- Use cases: packet filtering, security (segcomp)

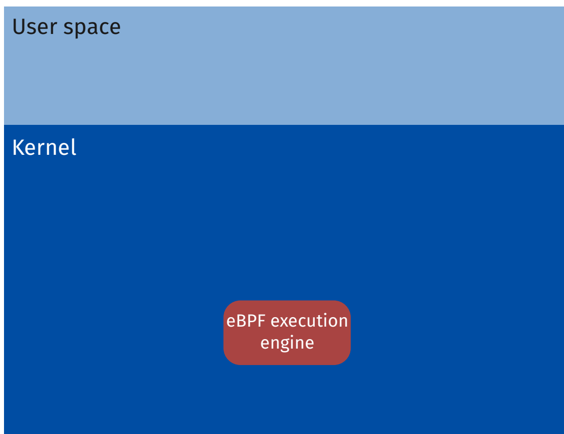
Usage

```
int s = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
setsockopt(s, SOL_SOCKET, SO_ATTACH_FILTER,
           &bpf_prog, sizeof(bpf_prog));
```

Time passes…

- 2013+: “eBPF” (extended BPF), Linux only
  (Alexei Starovoitov, in the context of the IO Visor project)

Complete rework of BPF architecture


User space
Kernel
eBPF execution engine

eBPF Virtual Machine

eBPF basics

- 10 general purpose registers (+1 stack register), 64-bit
- 512-byte stack
- New, larger set of instructions, closer to assembly

eBPF program object

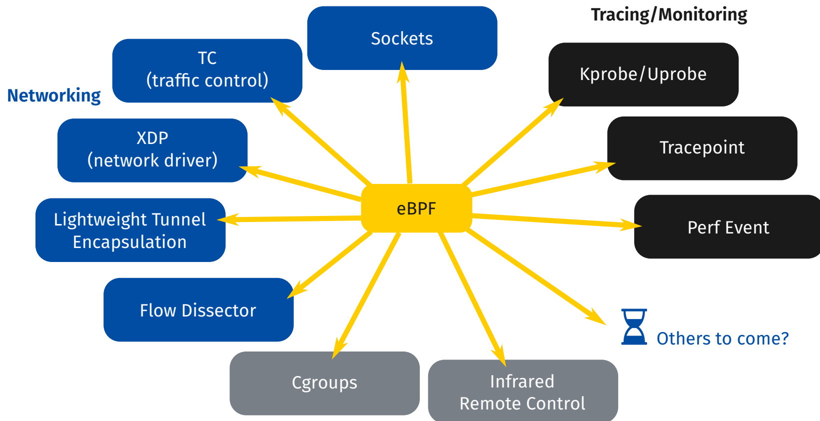
- Loaded from user space to kernel
- Attached to a given hook
- Run on events

eBPF Program Types

NETRONOME

```
BPF_PROG_TYPE_SOCKET_FILTER,           // Packet filtering
BPF_PROG_TYPE_KPROBE,                  // Tracing (any function)
BPF_PROG_TYPE_SCHED_CLS,               // Packet filtering (TC)
BPF_PROG_TYPE_SCHED_ACT,               // Packet filtering (TC)
BPF_PROG_TYPE_TRACEPOINT,              // Tracing (stable tracepoints)
BPF_PROG_TYPE_XDP,                     // Packet filtering (driver level)
BPF_PROG_TYPE_PERF_EVENT,              // Tracing (Proc. Monit. Unit events)
BPF_PROG_TYPE_CGROUP_SKB,              // Access control (IP ingress/egress)
BPF_PROG_TYPE_CGROUP_SOCK,             // Access control (socket crea/ops/...)
BPF_PROG_TYPE_LWT_IN,                  // Network tunnels
BPF_PROG_TYPE_LWT_OUT,                 // Network tunnels
BPF_PROG_TYPE_LWT_XMIT,                // Network tunnels
BPF_PROG_TYPE_SOCK_OPS,                // Update socket options
BPF_PROG_TYPE_SK_SKB,                  // Socket redirection
BPF_PROG_TYPE_CGROUP_DEVICE,           // Access control (device)
BPF_PROG_TYPE_SK_MSG,                  // Data stream filtering
BPF_PROG_TYPE_RAW_TRACEPOINT,          // Tracing
BPF_PROG_TYPE_CGROUP_SOCK_ADDR,        // Access control (socket binding)
BPF_PROG_TYPE_LWT_SEG6LOCAL,           // Network tunnels
BPF_PROG_TYPE_LIRC_MODE2,              // Infra-red remote control protocols
BPF_PROG_TYPE_SK_REUSEPORT,            // Select socket to use
BPF_PROG_TYPE_FLOW_DISSECTOR,          // Network processing
BPF_PROG_TYPE_CGROUP_SYSCTL,           // Access control (procfs)
BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE, // Tracing
```

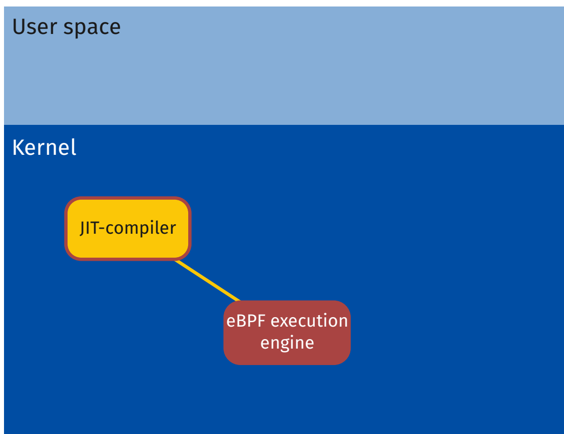
NETRONOME



- Different hooks
- Different context: packet data, function arguments (tracing), ...
- Different semantics

JIT-Compiler

NETRONOME



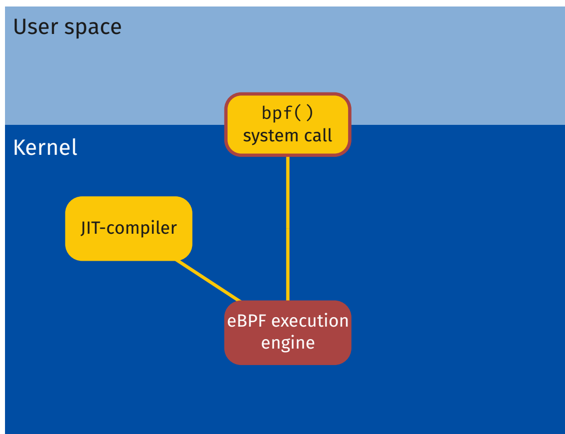
JIT-Compiler for Performance

Just-In-Time compilation: BPF instructions → native code

- Alternative to kernel interpreter, brings **performance**
- Supported architectures:
  ARM32, ARM64, MIPS, PowerPC64, RiscV, Sparc64, s390, x86_32, x86_64
- Hardware offload: NFP (Netronome)
- May be enabled/disabled via sysctl:
  ```
  # sysctl -w net.core.bpf_jit_enable=1
  ```
  Kernel config may force JIT to be used (because of Specter)

System Call

NETRONOME



System Call – Manage BPF Objects

```
#include <linux/bpf.h>

int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

- Control of BPF objects (programs, maps, etc.): e.g. load a program
- See `man bpf` (but out-of-date)
- C wrappers around `bpf()`: **libbpf** (shipped with kernel)

NETRONOME

How to keep a handle on a program?

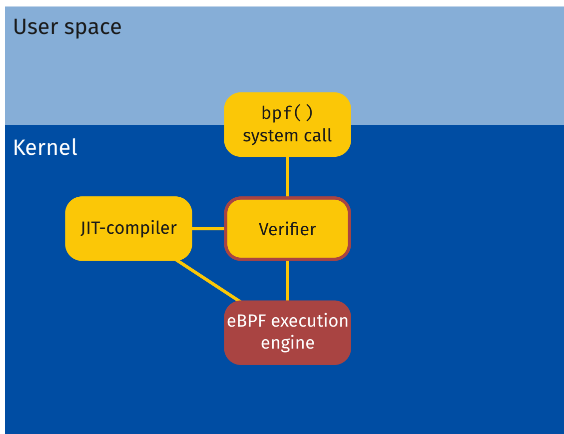
1. Program (BPF bytecode) is loaded in the kernel with `bpf()`
2. `bpf()` returns a file descriptor to the program
3. The file descriptor is used to attach that program to a hook
4. Program is automatically removed by kernel when:
   - All instances are detached
   - File descriptor is closed

Keep a program loaded after loader exits?

- Virtual file system (usually `/sys/fs/bpf/`), “bpffs”
- Pin programs (with `bpf()`), remove with `unlink()` (e.g. `rm <path>`)
- Programs kept as long as pinned in the virtual file system

Verifier

NETRONOME



Verifier – Ensure Termination and Safety

NETRONOME

a.k.a “The last rampart against evil *[programs]*”

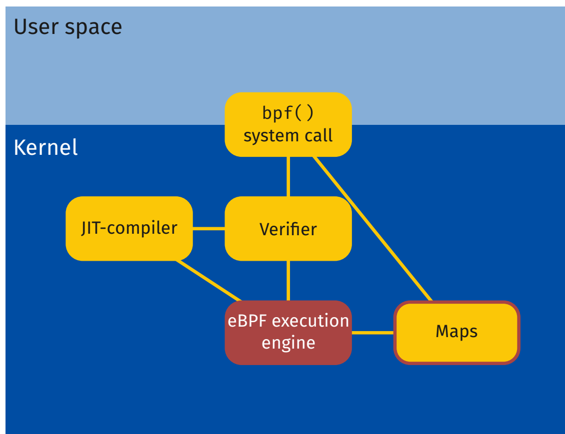
BPF programs come from user space: make sure they **terminate** / are **safe**

Termination:

- Direct Acyclic Graph (DAG), inspect instructions, prune safe paths
- Maximum: 4096 instructions… *OH WAIT* now “up to 1 million” for root
- No back edge (loops)
  - Except function calls
  - May change soon (bounded loops)

Safety:

- No unreachable code, no jump out of range
- Registers and stack states are valid for every instruction
- Program does not read uninitialised registers/memory
- Program only interacts with relevant context (prevent out-of-bound/random memory access)
- …

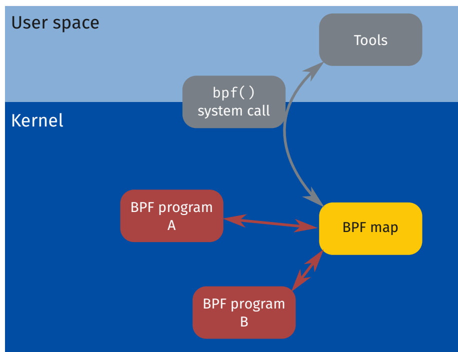

User space
Kernel
bpf()
system call
JIT-compiler
Verifier
eBPF execution
engine
Maps

Maps

NETRONOME

"Maps": special kernel memory area accessible to a program

- Shared between: several program runs, several programs, user space
- Typically, "key/value" storage: hash map, array
- Some of them have a "per-CPU" version
- Generally, RCU-protected; also, spinlocks now available in BPF

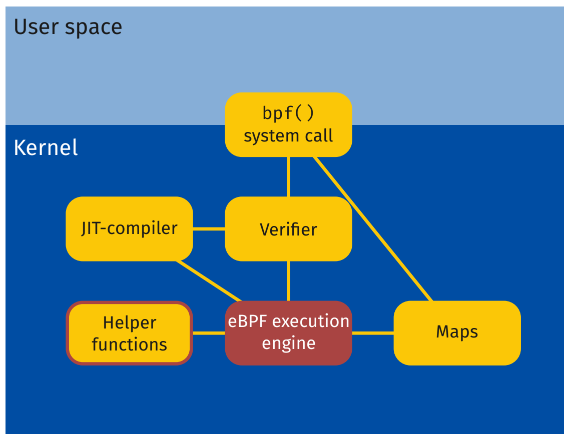


Map Types

NETRONOME

```
BPF_MAP_TYPE_HASH,                      // Hash map
BPF_MAP_TYPE_ARRAY,                     // Array
BPF_MAP_TYPE_PROG_ARRAY,                // Store BPF programs for tail calls
BPF_MAP_TYPE_PERF_EVENT_ARRAY,          // Stream info to user space
BPF_MAP_TYPE_PERCPU_HASH,               // Per-CPU hash map
BPF_MAP_TYPE_PERCPU_ARRAY,              // Per-CPU array
BPF_MAP_TYPE_STACK_TRACE,               // Stack info for tracing
BPF_MAP_TYPE_CGROUP_ARRAY,              // Store references to cgroups
BPF_MAP_TYPE_LRU_HASH,                  // Least-Recently-Used (cache)
BPF_MAP_TYPE_LRU_PERCPU_HASH,           // Per-CPU Least-Recently-Used (cache)
BPF_MAP_TYPE_LPM_TRIE,                  // Longest-Prefix Match
BPF_MAP_TYPE_ARRAY_OF_MAPS,             // Array of BPF maps
BPF_MAP_TYPE_HASH_OF_MAPS,              // Hash map of BPF maps
BPF_MAP_TYPE_DEVMAP,                    // Redirect packet to device
BPF_MAP_TYPE_SOCKMAP,                   // Redirect packet to socket
BPF_MAP_TYPE_CPUMAP,                    // Redirect packet to CPU
BPF_MAP_TYPE_XSKMAP,                    // Redirect packet to AF_XDP socket
BPF_MAP_TYPE_SOCKHASH,                  // Redirect packet to socket
BPF_MAP_TYPE_CGROUP_STORAGE,            // Store data per cgroup
BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,       // Select a socket for packet
BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,     // Per-CPU cgroup storage
BPF_MAP_TYPE_QUEUE,                     // Queue (FIFO)
BPF_MAP_TYPE_STACK,                     // Stack (LIFO)
BPF_MAP_TYPE_SK_STORAGE,                // Store data per socket
```

NETRONOME



Helper Functions – A Library for the Complex Bits

NETRONOME

"Standard library" of functions implemented in the kernel

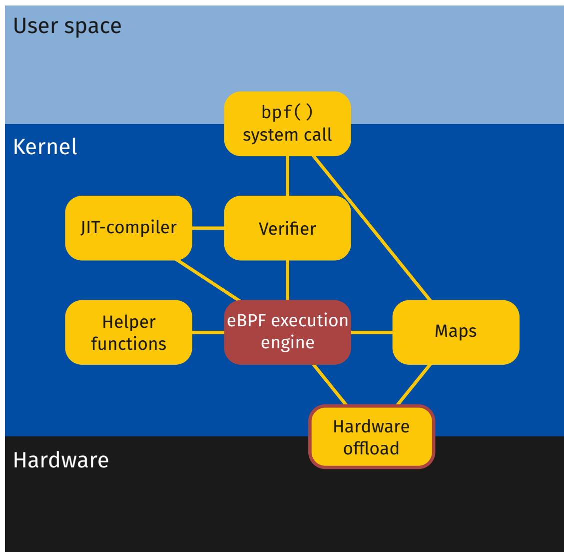
- Can be called from BPF programs
- Ease some tasks, manipulate maps, context, ..., e.g.:
  - Map lookup, update
  - Get kernel time
  - `printk()` equivalent
  - Change packet size
  - Redirect a packet
  - Safely dereference a kernel pointer
- Up to 5 arguments
- Some of them restricted to GPL-compatible BPF programs

More than 100 helper functions in the kernel already

Additional Features

- Tail calls: “long jumps” into other BPF programs (max: 33 tail calls)
- Function calls
- BTF: BPF Type Format, for debug (and more)
- Bounded loop: soon?

NETRONOME



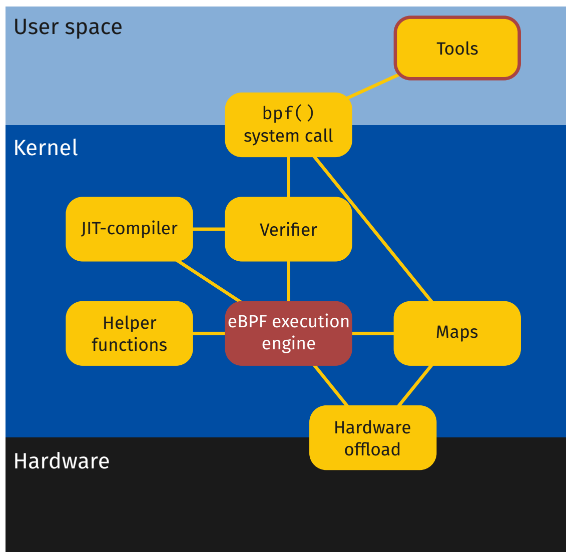
Hardware Offload

NETRONOME

- Hardware offload support for packet processing (Netronome)

... but let's keep this for another time `:-)`

NETRONOME



Good news: *Nobody* writes BPF bytecode directly
Use the BPF **clang/LLVM backend**

- Store BPF bytecode in ELF object file
- Compile from C to BPF

```
clang -O2 -g -emit-llvm -c prog.c -o - | \
        llc -march=bpf -mcpu=probe -filetype=obj -o prog.o
```

Other alternatives: Lua, Rust, ...

NETRONOME

Full BPF workflow:

1. Write a program in C
2. Compile with clang into ELF file
3. Get map information from ELF file, create maps (`bpf()`)
4. Get program information from ELF file, load program (`bpf()`)
5. Attach program (`bpf()` or other, depends on program type)
6. [Program runs...]
7. Detach program

Tooling – Other BPF tools

- Management, introspection: **bpftool**, perf
- Network processing: iproute2 (ip, tc)
- llvm-objdump: ELF file inspection
- Higher-level tools: **bcc**, **bpftrace**, etc.

Tracing and Monitoring

Disclaimer: I am not an expert in tracing!

You may have heard of...

- ftrace
- perf
- SystemTap
- LTTng
- Sysdig
- DTrace

Dynamic Tracing on Linux – Getting the Data

- Kprobes, kretprobes
  - kprobes: patch the first instructions of the function to execute custom code
  - kretprobes: patch return address of the function to execute custom code
  - Work for almost **any** (non-inlined) function, see `/proc/kallsyms`
  - But kernel internals are not API, may evolve and break probing!
- Uprobes, uretprobes
  - Same thing as above, for user space applications
- Tracepoints
  - Specific break points added prior to compiling the kernel
  - Disabled by default (no overhead), can be enabled (some overhead)
  - Much more stable than kprobes between kernel versions
  - E.g. all system calls have one
  - User space version: USDT, a.k.a. “DTrace probes”
- Perf_events
  - Relies on CPU Performance Monitor Unit (PMU): software and hardware counters
- Other kernel modules
  - Hack your own tracing system…

Dynamic Tracing Tools

Tracer/Front-end	Data sources	Data collection
ftrace	kprobe, uprobe, tracepoints, USDT	ftrace (sysfs)
perf	perf_events (+kprobes, tracepoints...)	perf ring buffer
SystemTap	kprobe, uprobe, tracepoints, USDT	kernel module
LTTng	Specific events, or user space tracing	kernel module
Sysdig	syscalls (not sure how)	Sysdig ring buffer
DTrace	DTrace probes... but not on Linux!	

Limitations:

- SystemTap and LTTng require building and inserting kernel modules
- ftrace, perf, LTTng lack programmability
- SystemTap not user friendly, could crash the kernel
- Sysdig limited to system calls
- DTrace very powerful but not in Linux kernel (Some out-of-tree ports available)

BPF for Tracing

NETRONOME

BPF can attach to:

- kprobes/kretprobes
- uprobes/uretprobes
- tracepoints, USDT
- perf_events

Data collection:

- ftrace/sysfs
- perf ring buffer
- BPF maps

Advantages:

- Programmable
- Fast, secure
- No kernel module

What BPF Does

NETRONOME

Typically, attaching a BPF program allows one to:

- Examine the arguments of a function
- Examine its context (PID, parent, stack, etc.)
- Examine its return value (retprobes)
- Aggregate, process all of these as desired
- Collect statistics

BPF *cannot* be used to modify the behaviour of a function, the content of its variables, etc.

Error injection (changing return value) possible

BPF for Tracing – Limitations

NETRONOME

BPF is not perfect

- Recent framework
  - Needs recent kernel, especially for latest features
  - Still evolving a lot (but at least, no API break)
- Still difficult to use on its own
  - Programming not so easy (verifier is picky)
  - Needs some time investment
  - Debugging gets better but still needs some work

Example Code!

Kprobe on `do_sys_open(dfd, filename, flags)`: print file name, flags

```
#include <linux/ptrace.h>
#include "bpf_helpers.h"

int trace_open(struct pt_regs *ctx, int dfd,
               const char __user *filename, int flags)
{
        u64 id = bpf_get_current_pid_tgid();
        u32 pid = id >> 32;

        bpf_trace_printk("%d: open(%s, %x)\n",
                         pid, filename, flags);

        return 0;
}
```

(Some simplification on `bpf_trace_printk()`, but this is just a matter of adding the correct macro)

Example Code – How to Run?

NETRONOME

Reminder: all the steps you don't want to take care of

1. Compile from C to BPF (ELF object file) with clang/LLVM
   (Get all header inclusion right)
2. Perform ELF relocation steps
3. Extract BPF bytecode and map data from ELF file
4. Create maps if any (`bpf()`)
5. Load program (`bpf()`)
6. Attach program (`perf_event_open()`, `ioctl()`)
7. Read and print collected data (sysfs, perf buffer)
8. Detach program

What can we do instead?

BPF Tracing Tools: bcc, bpftrace, ...

Example Code – With bcc

bcc: Framework for BPF tools, mostly a set of Python wrappers

```
from bcc import BPF

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

int trace_open(struct pt_regs *ctx, int dfd,
               const char __user *filename, int flags)
{
        u64 id = bpf_get_current_pid_tgid();
        u32 pid = id >> 32;

        bpf_trace_printk("%d: open(%s, %x)\\n",
                         pid, filename, flags);

        return 0;
}
""")
b.attach_kprobe(event="do_sys_open",
                fn_name="trace_open").trace_print()
```

```
# ./my_open_tracer.py
irqbalance-822   [006] .... 101740.269261: 0: 822: open(/proc/irq/8/smp_affinity, 8000)
irqbalance-822   [006] .... 101740.269277: 0: 822: open(/proc/irq/9/smp_affinity, 8000)
irqbalance-822   [006] .... 101740.269293: 0: 822: open(/proc/irq/10/smp_affinity, 8000)
      nvim-15918 [013] .... 101741.705896: 0: 15918: open(/tmp/nvim0nCfpU/2.1.py, 88241)
        sh-16847 [004] .... 101741.708720: 0: 16847: open(/etc/ld.so.cache, 88000)
        sh-16847 [004] .... 101741.708765: 0: 16847: open(/lib/x86_64-linux-gnu/libc.so.6, 88000)
        sh-16847 [004] .... 101741.708859: 0: 16847: open(/lib/x86_64-linux-gnu/libdl.so.2, 88000)
        sh-16848 [015] .... 101741.709583: 0: 16848: open(/tmp/nvim0nCfpU/1.1.py, 8241)
       git-16849 [016] .... 101741.710264: 0: 16849: open(/etc/ld.so.cache, 88000)
```

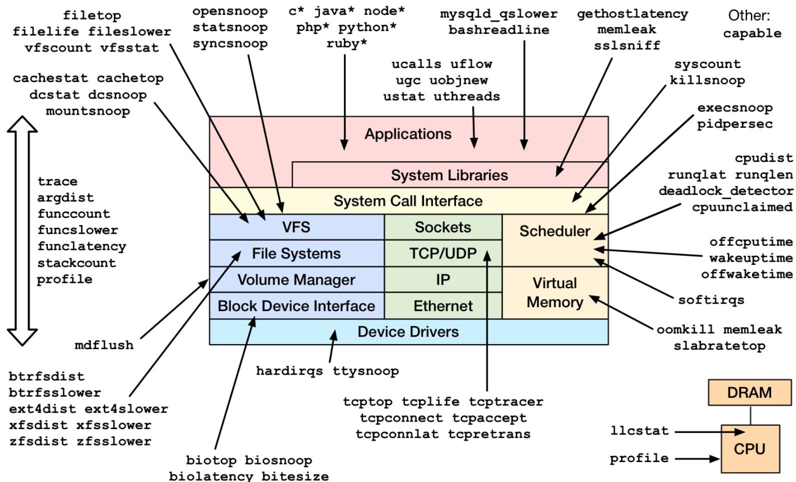
NETRONOME

bcc

- Framework for BPF tools, mostly a set of Python wrappers
- Also comes with a set of tools (87 tracing tools for now, +examples)

bcc – List of Tools (2018)

NETRONOME



Credits: Brendan Gregg

NETRONOME

opensnoop: Monitor usage of open() system call

- Attach a kprobe (and a kretprobe) to sys_do_open()

```
# ./opensnoop.py
PID    COMM      FD ERR PATH
1576   snmpd     11   0 /proc/sys/net/ipv6/neigh/lo/retrans_time_ms
1576   snmpd     11   0 /proc/sys/net/ipv6/conf/lo/forwarding
1576   snmpd     11   0 /proc/sys/net/ipv6/neigh/lo/base_reachable_time_ms
1576   snmpd      9   0 /proc/diskstats
1576   snmpd      9   0 /proc/stat
1576   snmpd      9   0 /proc/vmstat
1956   supervise  9   0 supervise/status.new
1956   supervise  9   0 supervise/status.new
17358  run        3   0 /etc/ld.so.cache
[...]
```

- Kprobe stores command name, filename, fd in a map
- Kretprobe retrieves info from map and prints it, with return value

`capable`: Monitor usage of capabilities (permissions) on Linux

- Attach a kprobe to `cap_capable()`

```
# ./capable.py
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
22:11:23  114    2676   snmpd            12   CAP_NET_ADMIN        1
22:11:23  0      6990   run              24   CAP_SYS_RESOURCE     1
22:11:23  0      7003   chmod            3    CAP_FOWNER           1
22:11:23  0      7003   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        7    CAP_SETUID           1
22:11:24  0      7013   run              24   CAP_SYS_RESOURCE     1
22:11:24  0      7026   chmod            3    CAP_FOWNER           1
22:11:24  0      7026   chmod            4    CAP_FSETID           1
[...]
```

NETRONOME

`bashreadline`: See all commands entered in bash

- Add a uretprobe (return probe, user space) to symbol `readline` in `/bin/bash`

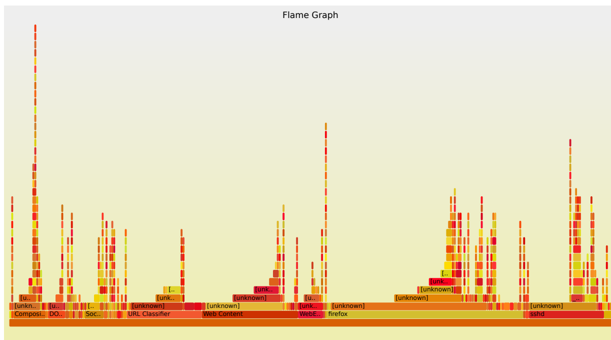
```
# ./bashreadline.py
TIME      PID    COMMAND
05:28:25  21176  ls -l
05:28:35  21176  echo "Hello Sqreen"
05:29:04  3059   echo "command from another shell"
```

- See also `sslsniff`: user probes in SSL libs to dump unencrypted data

Profile CPU usage: “flame graph” indicating how much time functions run

- Poll software perf_event `CPU_CLOCK`, collect stack data
- Info and `flamegraph.pl` script at https://github.com/brendangregg/FlameGraph
- Also usable for Python stack, Ruby, PHP, C*, Java, Node.js, ...

```
# ./profile.py -f 10 > data.out
$ ./flamegraph.pl data.out > graph.svg
```



Another BPF tracing tool, higher-level: bpftrace

- Awk-like syntax
- Sits on top of bcc
- Embeds a number of built-in functions and variables
- “Linux equivalent to DTrace”
- Express programs as one-liners, or very short scripts
- Also comes with a number of ready-to-use scripts
- (Has very good documentation!)

Our simple example to monitor `open()`:

```
# bpftrace -e 'kprobe:do_sys_open { printf("%d-%s: %s\n", pid, comm, str(arg1)) }'
```

- Approximative syntax:
  `probe_type:probe_target /filter/ { command block }`
- Variables include `pid`, `nsecs`, `comm`, `cpu`, `arg0..argN`, `rand`, …
- Functions for printing, manipulating maps, drawing histograms…

Some on-liners from bpftrace documentation

```
# Syscall count by program
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Read bytes by process
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret/ { @[comm] = sum(args->ret); }'

# Read size distribution by process
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Count page faults by process
bpftrace -e 'software:faults:1 { @[comm] = count(); }'

# Count LLC cache misses by process name and PID (uses PMCs)
bpftrace -e 'hardware:cache-misses:1000000 { @[comm, pid] = count(); }'

# Profile user-level stacks at 99 Hertz, for PID 189
bpftrace -e 'profile:hz:99 /pid == 189/ { @[ustack] = count(); }'
```

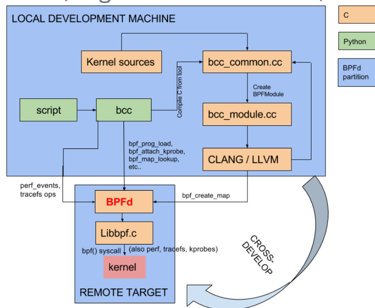
Read size distribution by process, and present it as a histogram

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'
Attaching 1 probe...
^C

@[cat]:
[0]                     1 |@@@@@@@@@@@@@@@@@                                   |
[1]                     0 |                                                    |
[2, 4)                  0 |                                                    |
[4, 8)                  0 |                                                    |
[8, 16)                 0 |                                                    |
[16, 32)                0 |                                                    |
[32, 64)                0 |                                                    |
[64, 128)               0 |                                                    |
[128, 256)              0 |                                                    |
[256, 512)              0 |                                                    |
[512, 1K)               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1K, 2K)                1 |@@@@@@@@@@@@@@@@@                                   |

@[wc]:
[...]
```

More BPF-Related Tools

NETRONOME

- ply: same principle as bpftrace (but older), no dependency on bcc
- Sysdig: now with an alternative eBPF backend
- perf: supports BPF programs, also helps for BPF introspection
- bpfd: BPF daemon (target: containers)
- BPFd: BPF daemon too (target: android devices)

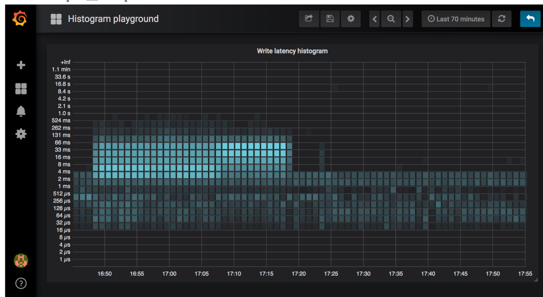


Credits: Joel Fernandes

NETRONOME

Monitoring systems with some BPF compatibility:

- Prometheus: ebpf_exporter tool



Credits: Ivan Babrou, Cloudflare

- Weave Scope has a plug-in for BPF

Some notable companies doing tracing and monitoring with BPF include Netflix, Cloudflare, Facebook, Google

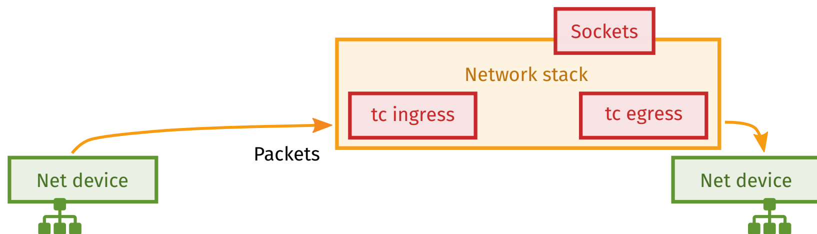
Other Use Cases for BPF

Major use cases for BPF

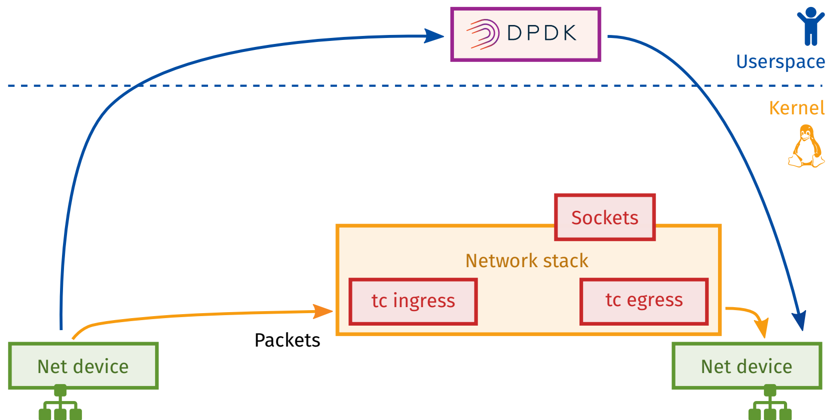
- Filtering (firewalling)
- Load balancing
- Protection against DDoS

Hooks on sockets, TC (traffic control), XDP

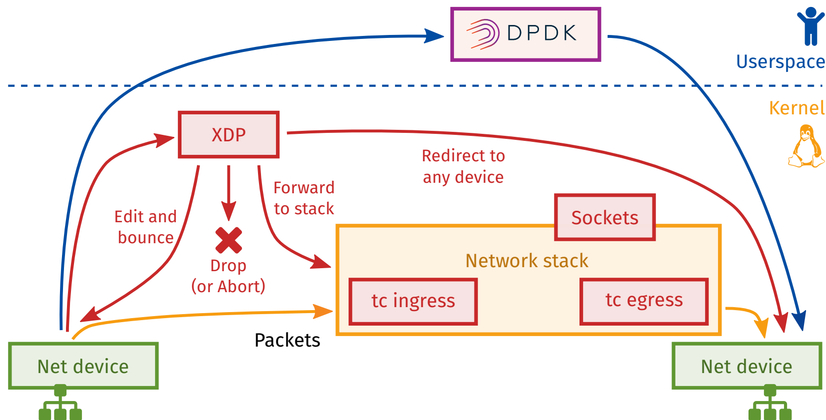
NETRONOME



- Historically: low performance for Linux kernel stack (socket buffers)
- One core: ~2Mpps (far from 10Gb/s link: ~14Mpps)



- Performance reached in user space: DPDK
- Cost: driver support required, polling, “out of kernel”, can be complex

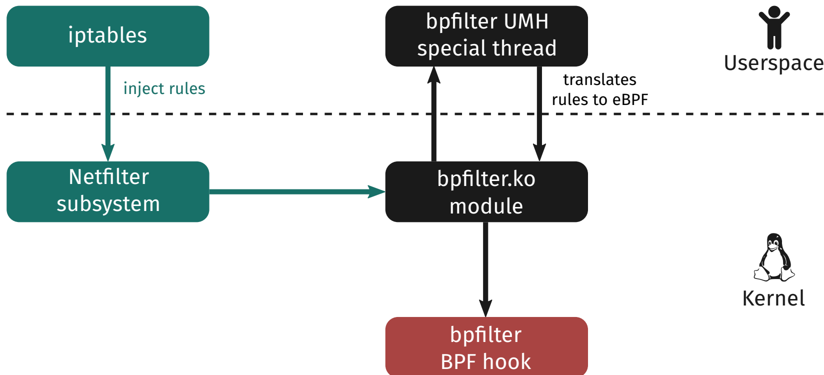


- XDP: low-level BPF hook (driver level, offload possible)
- Cooperates with kernel stack, no reboot required for new protocols, …

NETRONOME

bpfilter: Work-in-progress Linux kernel back-end for iptables

- iptables rules transparently converted into BPF programs
- Front-end (iptables) unchanged
- End of "sequential filtering"
- Better performance, security, hardware offload
- Some of it in kernel 4.18+, not complete yet



Load balancing

- Katran from Facebook
  Facebook is one of BPF's main contributors and users

Protection against DDoS

- Droplet (not published), also from Facebook
- Some work by Cloudflare

Fast packet capture

- Suricata (Network IDS)

Switching, data plane programming

- Open vSwitch (virtual switching: BPF-based data path in progress)
- Target for P4 (forwarding plane description language)
- DPDK: AF_XDP-based poll-mode driver

Access Control

NETRONOME

Concept

- Allow or deny access to resources
- Based on cgroups

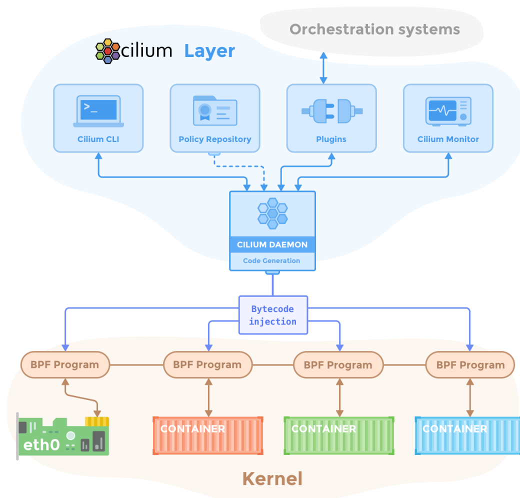
Examples

- (segcomp, with cBPF: AC on syscalls, e.g. for Firefox/Chrome)
- Landlock: Modern BPF-based security framework, not merged yet
- systemd: IP accounting for systemd services
  Hence e.g. Ubuntu 18.04 having BPF programs loaded at startup

Cilium: "*API-aware Networking and Security*" for containers

- BPF-based framework for ACLs in container clusters
- Rules at multiple layers: L2 to L7 (API)
- Avoid multiple traversal of the network stack for packets
- Integration with multiple frameworks (K8s, Istio, Docker, etc.)

Also among the main contributors to BPF



Credits: Cilium Authors

Wrapping up

Berkeley Packet Filter

- In-kernel virtual machine
- BPF programs: verified, JIT-compiled, share data with user space
- Programmability, security, performance

Use cases

- Networking (Filtering, load-balancing, anti-DDoS, switching)
- Tracing, monitoring
- Access control
- A few others... And new ones yet to come?

BPF development is extremely active!

BPF for tracing

- Kernel and user space probes, tracepoints
- No kernel module (or related safety issues)
- Programmable (more complex use cases, data aggregation)
- Very flexible, "endless possibilities"

But keep in mind...

- Linux only
- BPF, especially new features require recent kernels
- Currently missing e.g. loops, BPF libraries, ...
- BPF alone can be complex to use

Main tools and wrappers

- bcc
- bpftrace (~DTrace for Linux)
- ply, perf, BPFd, ...

NETRONOME

Thank You!

Discussion

NETRONOME

Some Additional resources

BPF and XDP Reference Guide
http://docs.cilium.io/en/latest/bpf/

Why is the kernel community replacing iptables with BPF?
https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/

Linux BPF Superpowers
http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html

Linux tracing systems & how they fit together
https://jvns.ca/blog/2017/07/05/linux-tracing-systems/

Linux Extended BPF (eBPF) Tracing Tools
http://www.brendangregg.com/ebpf.html

Using eBPF in Kubernetes
https://kubernetes.io/blog/2017/12/using-ebpf-in-kubernetes/

eBPF Tooling and Debugging Infrastructure
https://www.slideshare.net/Netronome/ebpf-tooling-and-debugging-infrastructure

Compilations

(Because there would be way too many things to list here)

Dive into BPF: a list of reading material
https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

Awesome eBPF
https://github.com/zoidbergwill/awesome-ebpf