The BSD Packet Filter
A New Architecture for User-level Packet Capture

Steven McCanne and Van Jacobson

(1993 Winter USENIX – San Diego, CA)

Presented by :
Suchakrapani Sharma

Papers We Love
$f(x) = x$

28th June 2017
Papers We Love - Montreal

Back in the ***olden*** days..

The BSD Packet Filter: A New Architecture for User-level Packet Capture

Steven McCanne & Van Jacobson – Lawrence Berkeley Laboratory[1]

ABSTRACT

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

Introduction

Unix has become synonymous with high quality networking and today's Unix users depend on having reliable, responsive network access. Unfortunately, this dependence means that network trouble can make it impossible to get useful work done and increasingly users and system administrators find that a large part of their time is spent isolating and fixing network problems. Problem solving requires appropriate diagnostic and analysis tools and, ideally, these tools should be available where the problems are – on Unix workstations. To allow such tools to be constructed, a kernel must contain some facility that gives user-level programs access to raw, unprocessed network traffic [7]. Most of today's workstation operating systems contain such a facil-

same hardware and traffic mix. The performance increase is the result of two architectural improvements:

- BPF uses a re-designed, register-based 'filter machine' that can be implemented efficiently on today's register based RISC CPU. CSPF used a memory-stack-based filter machine that worked well on the PDP-11 but is a poor match to memory-bottlenecked modern CPUs.
- BPF uses a simple, non-shared buffer model made possible by today's larger address spaces. The model is very efficient for the 'usual cases' of packet capture.[2]

In this paper, we present the design of BPF, outline how it interfaces with the rest of the system, and describe the new approach to the filtering mechan-

Problem Scope

Network Packet Tap

- Traditional network "tap" required copying packets in kernel buffers across kernel-userspace boundaries
  - Eg. SunOS's STREAMS NIT [10]

Network Packet Filtering

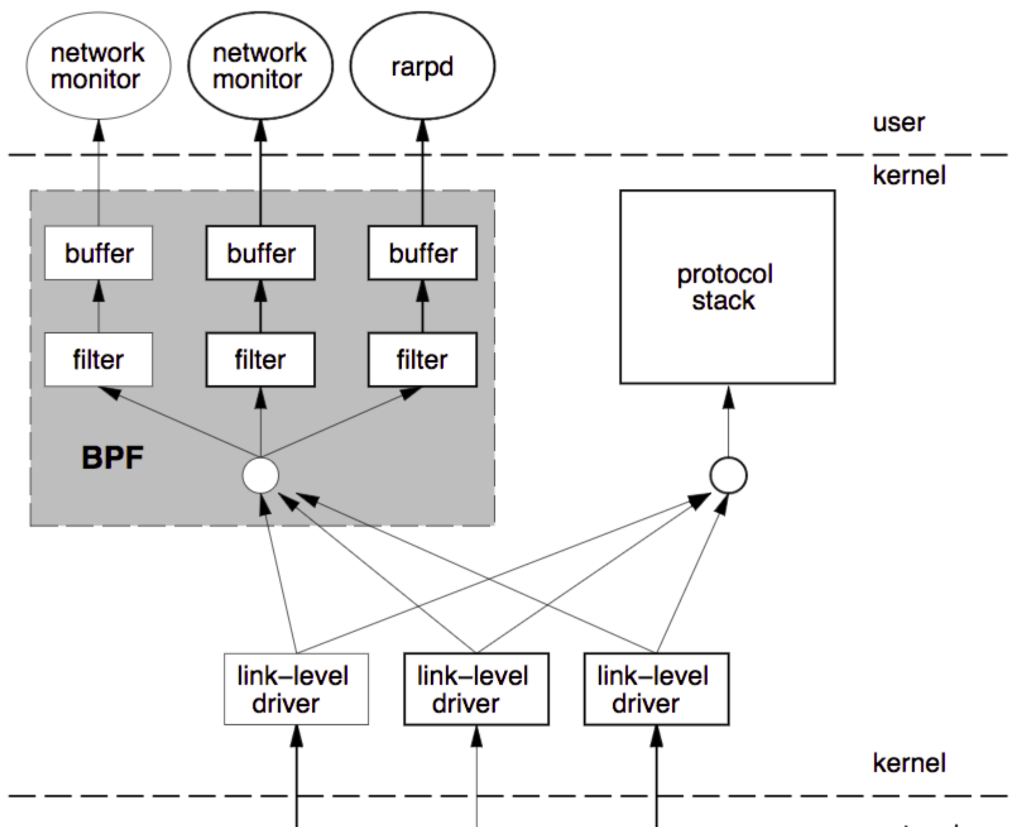
- Raw packets were accessed and filtered upstream
- Filters represented as predicate trees and processed
  - Eg. CMU/Stanford Packet Filter (CSPF) in Unix [8]
- Tree evaluation required
  - Stack simulation*
  - Redundant operations*

*Elaborated later

Network Tap

In-Kernel Filters

- Filters described in userspace but evaluated early
- If “passed”, copy buffer and pass upstream



Network Tap

BPF vs NIT

- Measure *bpf_tap()* vs *snit_intr()* + mbuf copy
- 5.7us (BPF) vs 89.2s (NIT) per packet (15x overhead)

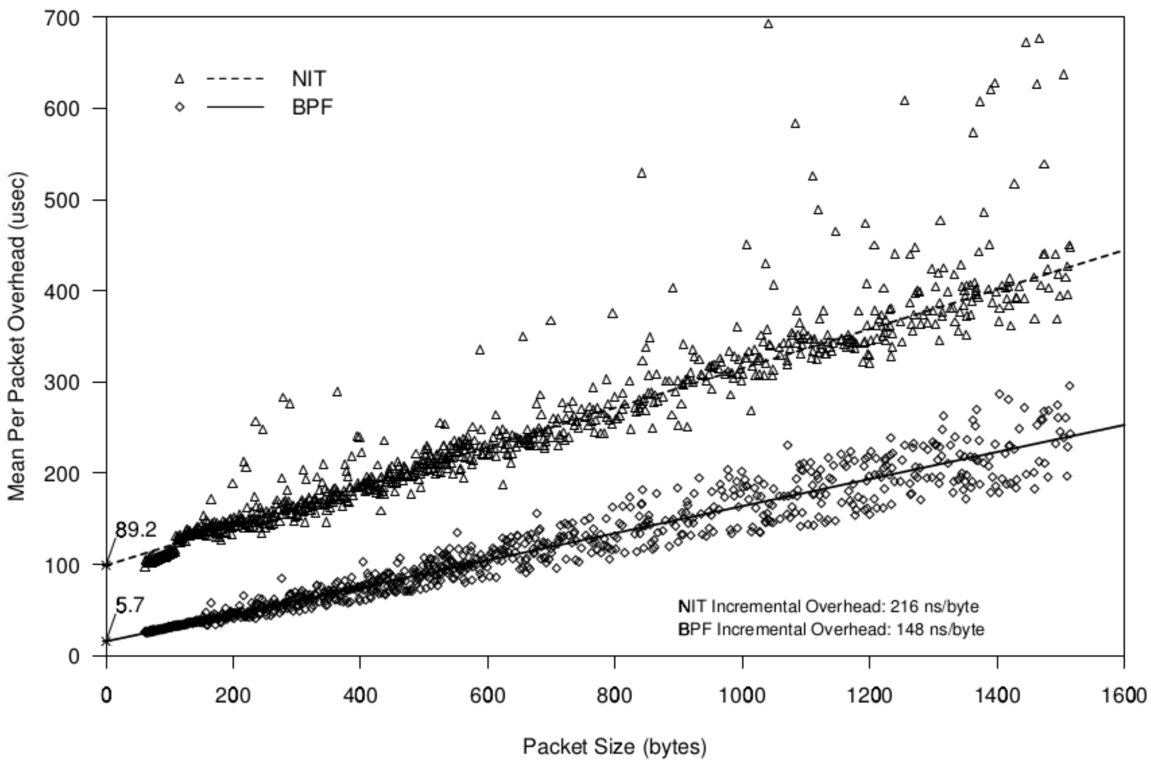
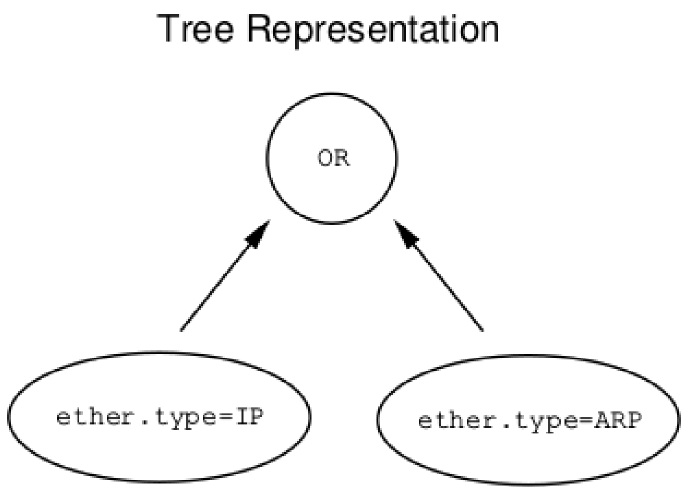


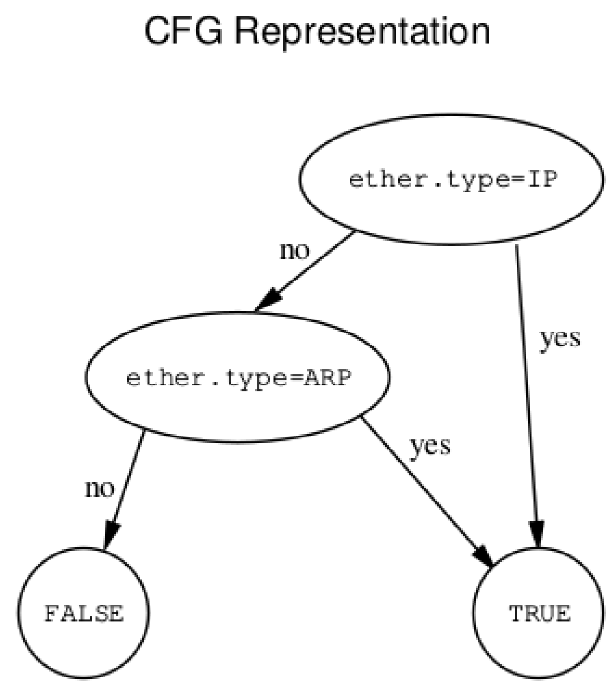
Figure 2: NIT versus BPF: "accept all"

Network Packet Filtering

Filter Model

- Boolean Expression Tree *vs* directed acyclic CFG





Network Packet Filtering

Boolean Expression Tree (CSPF)

- Easier to model with a stack based machine
- Implement load, stores to memory & simulate stack
- Redundant parses of tree needed

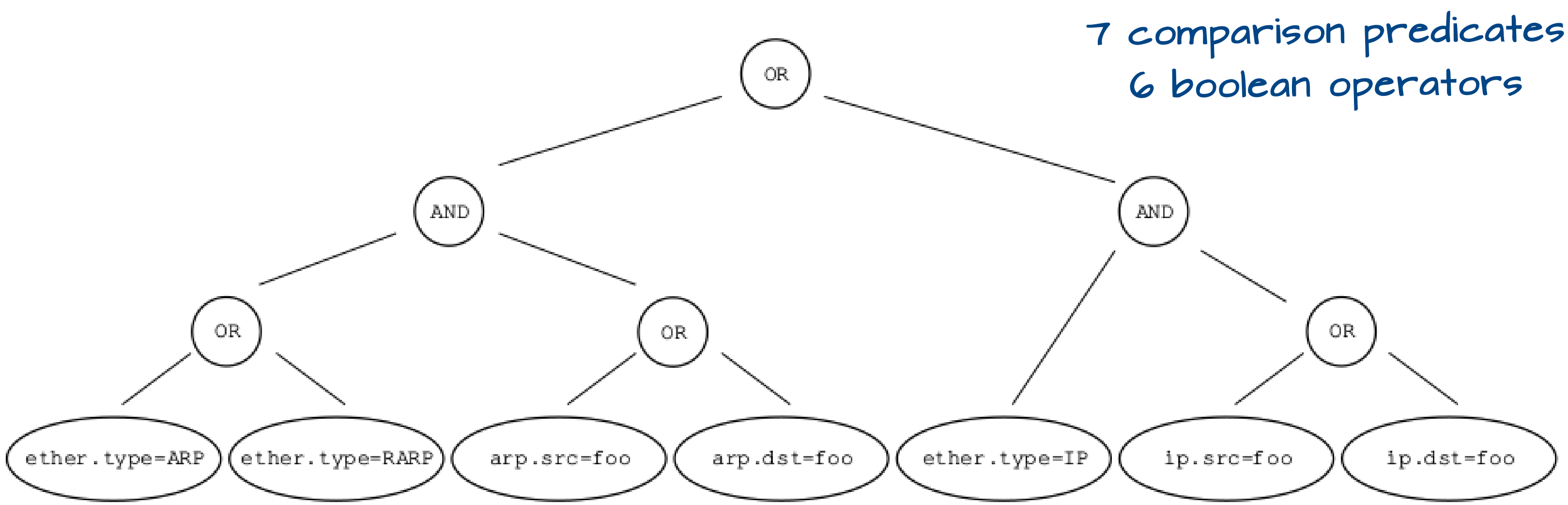
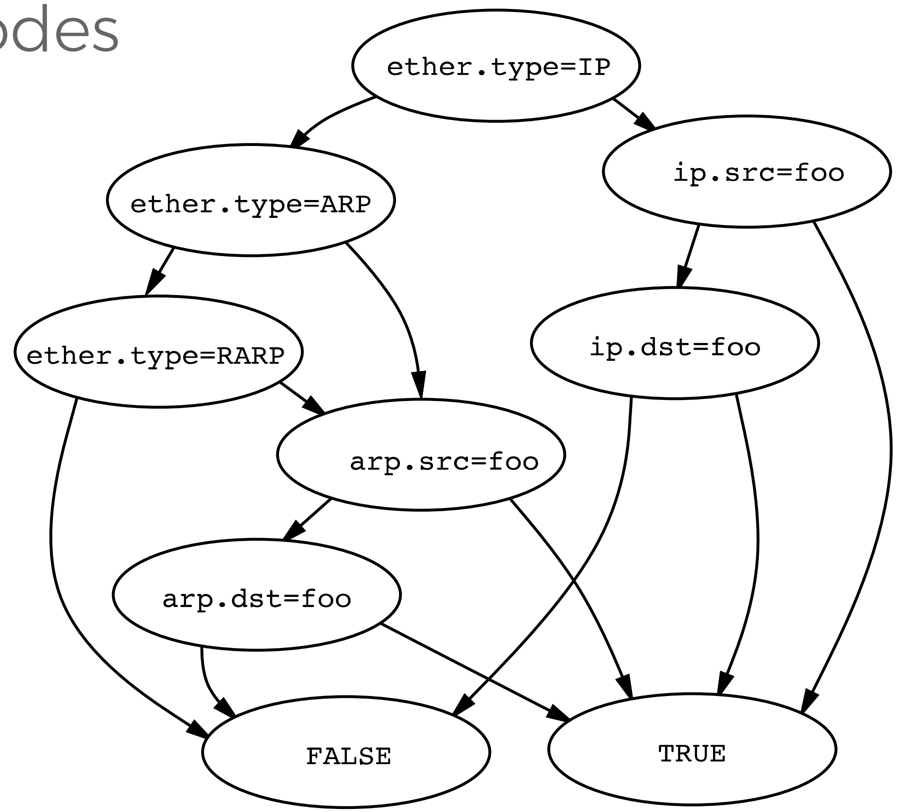


Figure 6: Tree Filter Function for “host foo”.

Network Packet Filtering

CFG (NNStat and BPF)

- Node are comparison predicates, with two final targets (TRUE/FALSE) (easier to model on registers)
- No redundant paths – but requires reordering of graph nodes



Max 5 comparisons

Network Packet Filtering

BPF Virtual Machine

- Not tied to any protocol. Packets are byte arrays
- A generic machine, easily programmable
- Variable length packets support*
- Simple switch-case dispatch mechanism
- Simple instruction set; A, X and scratch memory registers

Instruction Format

opcode:16	jt:8	jf:8
k:32		

Instr Representation

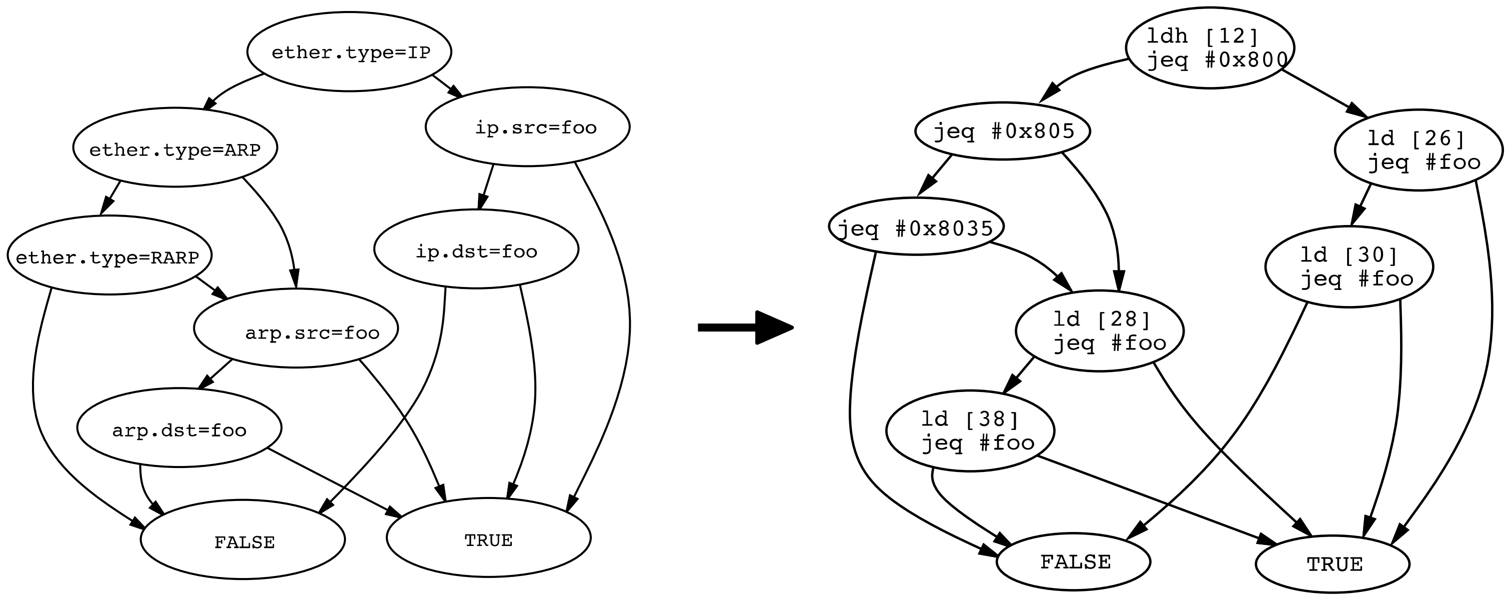
```
l0: ldh [12]
l1: jeq #0x800, l3, l2
l2: jeq #0x805, l3, l8
L3:
...
l7: ret #0xffff
l8: ret #0
```

Sample Instructions { OP, JT, JF, K }

```
{ 0x28,  0,  0, 0x0000000c },     /* 0x28 is opcode for ldh */
{ 0x15,  1,  0, 0x00000800 },     /* jump next to next instr if A = 0x800 */
{ 0x15,  0,  5, 0x00000805 },     /* jump to FALSE (offset 5) if A != 0x805 */
```

Network Packet Filtering

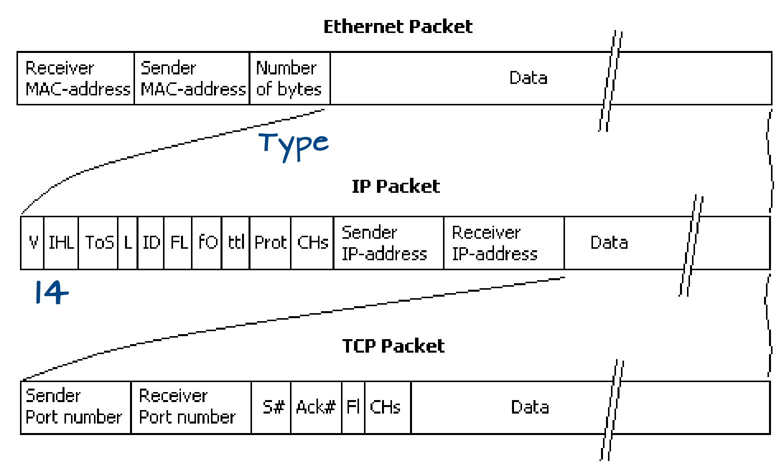
BPF Virtual Machine



Variable Length Packets Example (TCP)
(special addressing mode)

```
ldx 4*([14]&0xf)
ldh [x+16]
jeq #N, L1, L2
ret #TRUE
ret #0
```

Find length (IHL)
Then 16 bytes from that
(TCP destination port)



Network Packet Filtering

Sample BPF Interpreter (Linux Kernel v3.14)

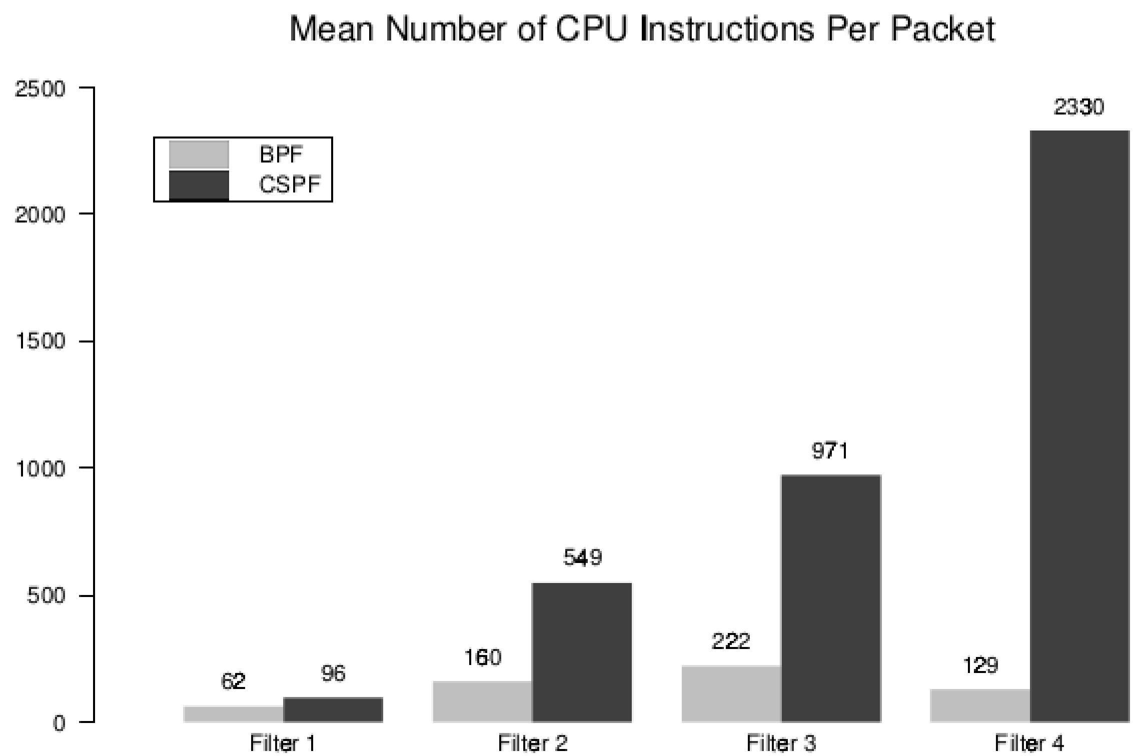
```
	u32 A = 0;			/* Accumulator */
	u32 X = 0;			/* Index Register */
	u32 mem[BPF_MEMWORDS];		/* Scratch Memory Store */
	u32 tmp;
	int k;

	/*
	 * Process array of filter instructions.
	 */
	for (;; fentry++) {
#if defined(CONFIG_X86_32)
#define K (fentry->k)
#else
		const u32 K = fentry->k;
#endif

		switch (fentry->code) {
		case BPF_S_ALU_ADD_X:
			A += X;
			continue;
		case BPF_S_ALU_ADD_K:
			A += K;
			continue;
..
```

Network Packet Filtering

BPF vs CSPF



Filter 1	IP packets
Filter 2	IP packets with src or dst "horse"
Filter 3	TCP packets with src or dst port of finger, domain, login, or shell
Filter 4	IP, ARP or RARP packets between hosts "horse" and "gauguin"

Figure 8: BPF/CSPF Filter Performance

▶▶ Fast forward to *present*

BPF in Linux Kernel

Classical BPF (cBPF)

- Network packet filtering, eventually seccomp
- Filter Expressions → Bytecode → Interpret*
- Small, in-kernel VM. Register based, switch dispatch interpreter, few instructions

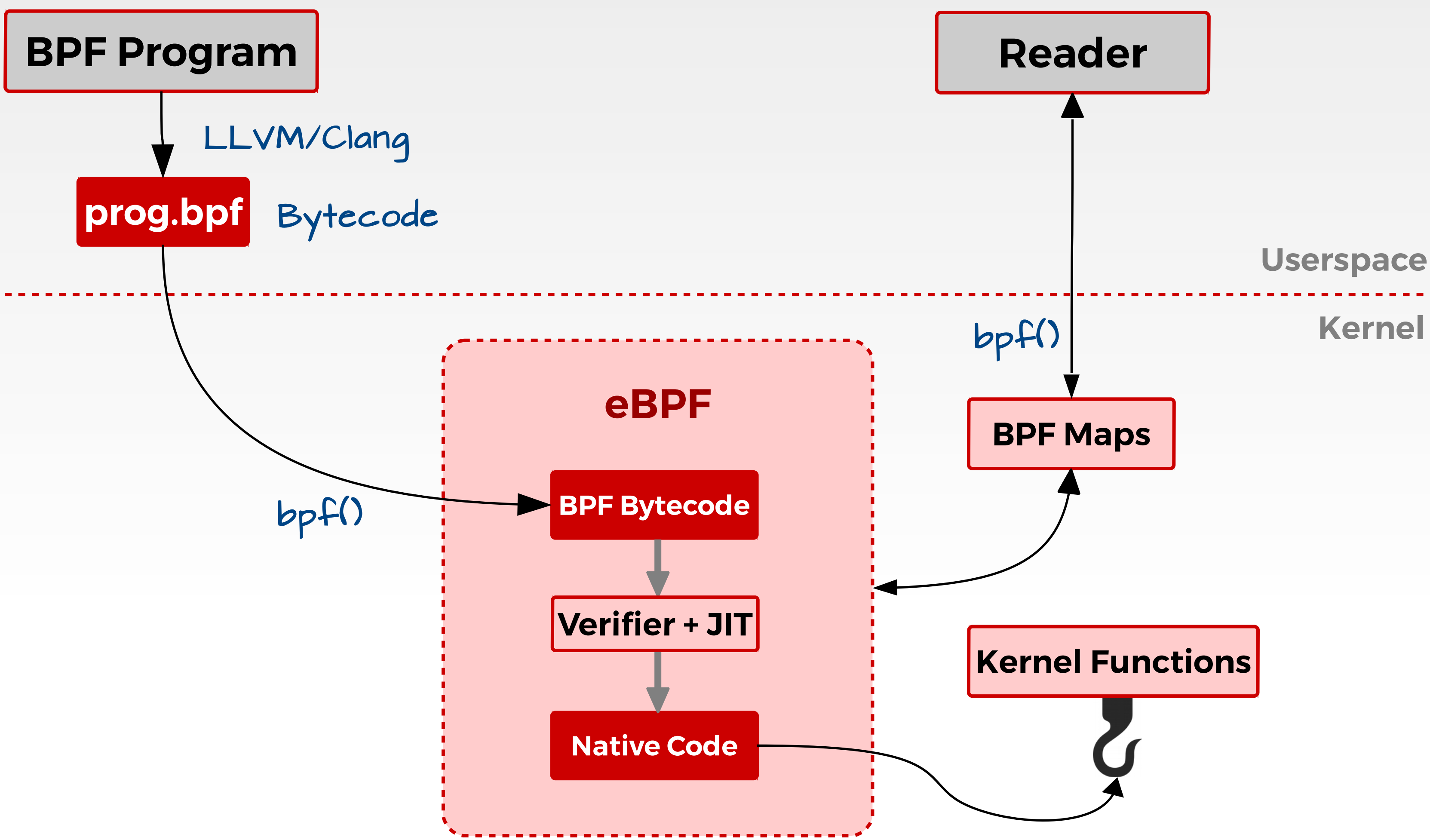
Extended BPF (eBPF) [Alexei Starovoitov, Borkmann et al.]

- More registers, JIT compiler (flexible/faster), verifier
- Attach on Tracepoint/Kprobe/Uprobe/USDT
- In-kernel trace aggregation & filtering
- Control via **`bpf()`**, trace collection via **BPF Maps**
- Upstream in Linux Kernel (**`bpf()`** syscall, v3.18+)
- Bytecode compilation upstream in LLVM/Clang

*JIT support eventually landed in kernel

BPF in Linux Kernel

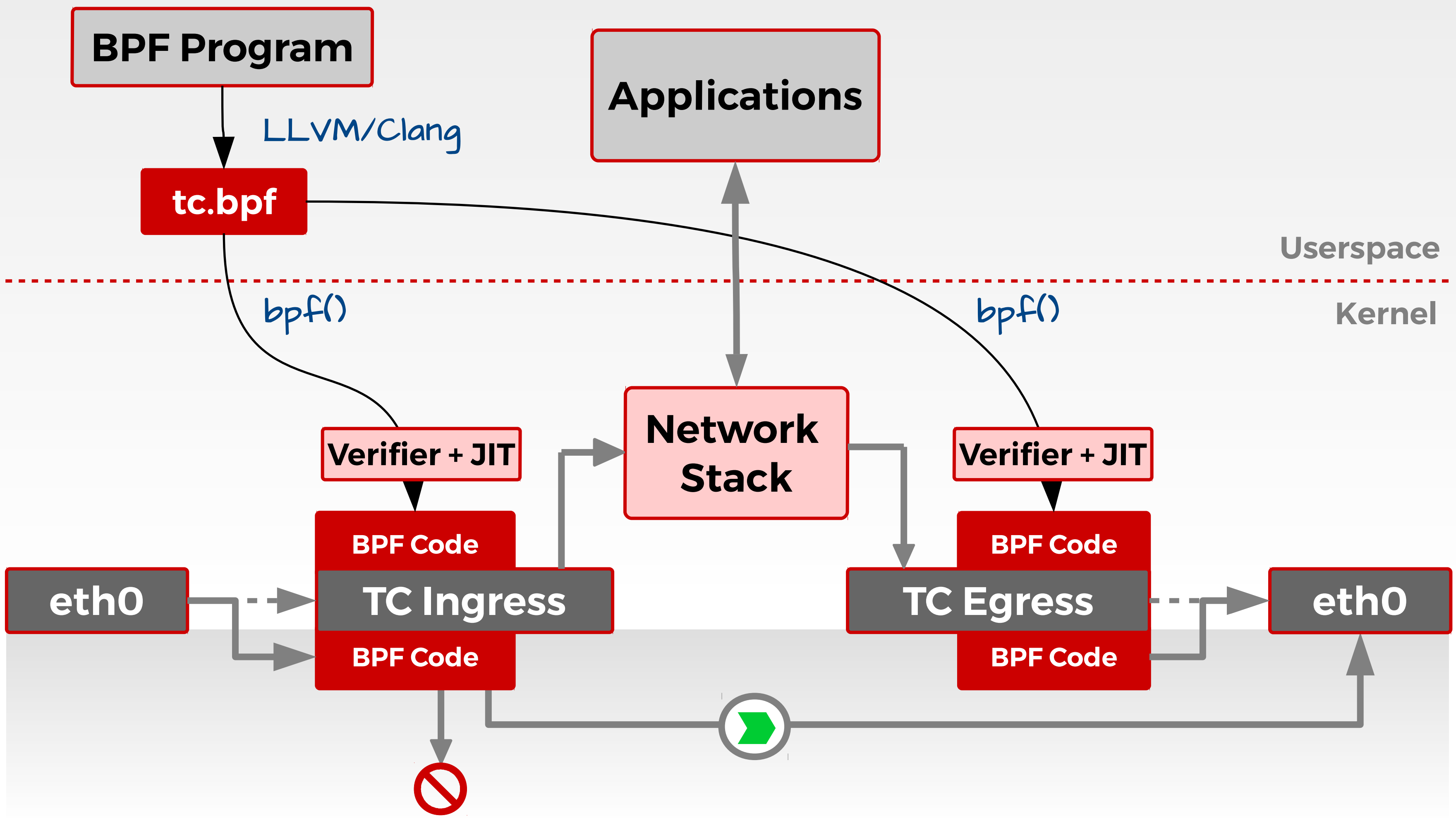
Modern eBPF Programs



eBPF for Networking

Traffic Control/XDP

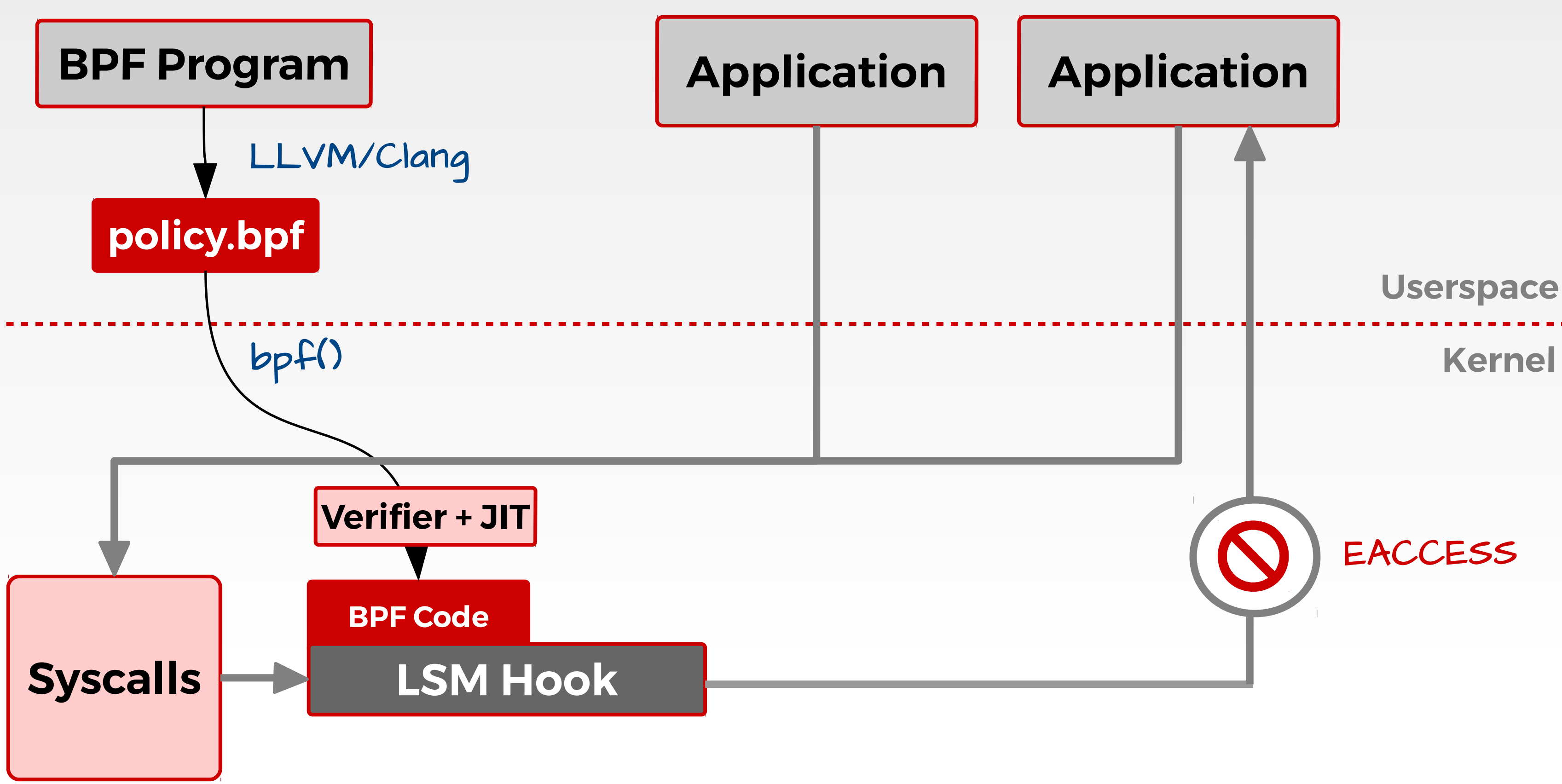
- TC with `cls_bpf` [Borkmann, 2016] `act_bpf` and XDP



Adapted from Thomas Graf's presentation "Cilium - BPF & XDP for containers"

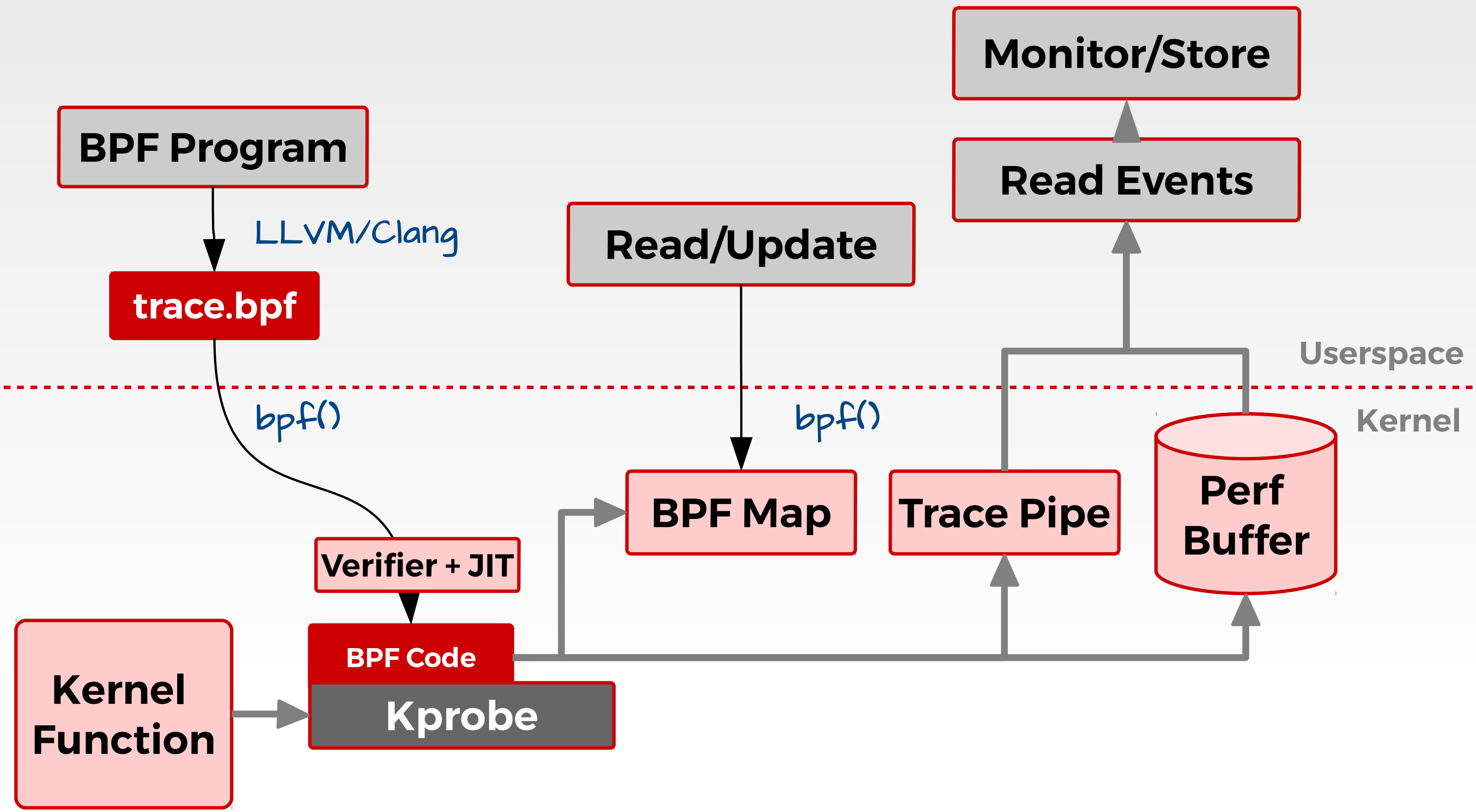
eBPF for Security

LSM Hooks



eBPF for Tracing

Kprobes/Kretprobes



eBPF Features & Support

Major BPF Milestones by Kernel Version*

- 3.18 : `bpf()` syscall
- 3.19 : Sockets support, BPF Maps
- 4.1 : Kprobe support
- 4.4 : Perf events
- 4.6 : Stack traces, per-CPU Maps
- 4.7 : Attach on Tracepoints
- 4.8 : XDP core and act
- 4.9 : Profiling, attach to Perf events
- 4.10 : cgroups support (socket filters)
- 4.11 : *Tracerception* – tracepoints for eBPF debugging

*Adapted from "BPF: Tracing and More" by Brendan Gregg (Linux.Conf.au 2017)

eBPF Features & Support

Program Types

- BPF_PROG_TYPE_UNSPEC
- BPF_PROG_TYPE_SOCKET_FILTER
- BPF_PROG_TYPE_KPROBE
- BPF_PROG_TYPE_SCHED_CLS
- BPF_PROG_TYPE_SCHED_ACT
- BPF_PROG_TYPE_TRACEPOINT
- BPF_PROG_TYPE_XDP
- BPF_PROG_TYPE_PERF_EVENT
- BPF_PROG_TYPE_CGROUP_SKB
- BPF_PROG_TYPE_CGROUP_SOCK
- BPF_PROG_TYPE_LWT_IN
- BPF_PROG_TYPE_LWT_OUT
- BPF_PROG_TYPE_LWT_XMIT
- BPF_PROG_TYPE_LANDLOCK

Tracing

Cgroups

Security

http://lxr.free-electrons.com/source/include/uapi/linux/bpf.h

eBPF Features & Support

Map Types

- BPF_MAP_TYPE_UNSPEC
- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_TYPE_PROG_ARRAY
- BPF_MAP_TYPE_PERF_EVENT_ARRAY
- BPF_MAP_TYPE_PERCPU_HASH
- BPF_MAP_TYPE_PERCPU_ARRAY
- BPF_MAP_TYPE_STACK_TRACE
- BPF_MAP_TYPE_CGROUP_ARRAY
- BPF_MAP_TYPE_LRU_HASH
- BPF_MAP_TYPE_LRU_PERCPU_HASH

http://lxr.free-electrons.com/source/include/uapi/linux/bpf.h

eBPF for Tracing

Frontends

- IOVisor BCC – Python, C++, Lua, Go (`gobpf`) APIs
- Compile BPF programs directly via LLVM interface
- Helper functions to manage maps, buffers, probes

Kprobes Example

Complete Program
trace_fields.py

```
from bcc import BPF

prog = """
int hello(void *ctx) {
  bpf_trace_printk("Hello, World!\\n");
  return 0;
}
"""
b = BPF(text=prog)
b.attach_kprobe(event="sys_clone", fn_name="hello")
print "PID MESSAGE"
b.trace_print(fmt="{1} {5}")
```

prog compiled to BPF bytecode

Attach to Kprobe event

Print trace pipe

eBPF for Tracing

Tracepoint Example (v4.7+)

Program Excerpt

Attach to tracepoint

Filter on args

```
# define EXIT_REASON 18

prog = """
TRACEPOINT_PROBE(kvm, kvm_exit) {
    if (args->exit_reason == EXIT_REASON) {
        bpf_trace_printk("KVM_EXIT exit_reason : %d\\n", args->exit_reason);
    }
    return 0;
}

TRACEPOINT_PROBE(kvm, kvm_entry) {
    if (args->vcpu_id = 0) {
        bpf_trace_printk("KVM_ENTRY vcpu_id : %u\\n", args->vcpu_id);
    }
}
"""
```

Output

```
# ./kvm-test.py
2445.577129000     CPU 0/KVM        8896    KVM_ENTRY vcpu_id : 0
2445.577136000     CPU 0/KVM        8896    KVM_EXIT exit_reason : 18
```

eBPF for Tracing

Uprobes Example

Program Excerpt

```
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

int get_fname(struct pt_regs *ctx) {
    if (!ctx->si)
       return 0;
    char str[NAME_MAX] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->si);
    bpf_trace_printk("%s\\n", &str);
    return 0;
};
"""
b = BPF(text=bpf_text)
b.attach_uprobe(name="/usr/bin/vim", sym="readfile", fn_name="get_fname")
```

Get 2nd argument

Process

Symbol

Output

```
# ./vim-test.py
TASK    PID     FILENAME
vim     23707   /tmp/wololo
```

eBPF for Tracing

USDT Example

Program Excerpt
nodejs_http_server.py

```
from bcc import BPF, USDT
.
.
bpf_text = """
#include <uapi/linux/ptrace.h>
int do_trace(struct pt_regs *ctx) {
    uint64_t addr;
    char path[128]={0};
    bpf_usdt_readarg(6, ctx, &addr);
    bpf_probe_read(&path, sizeof(path), (void *)addr);
    bpf_trace_printk("path:%s\\n", path);
    return 0;
};
"""

u = USDT(pid=int(pid))
u.enable_probe(probe="http__server__request", fn_name="do_trace")
b = BPF(text=bpf_text, usdt_contexts=[u])
```

Get 6th Argument

Read to local variable

Target PID

Probe in Node

eBPF for Tracing

USDT Example

Output

```
# ./nodejs_http_server.py 24728
TIME(s)            COMM             PID    ARGS
24653324.561322998 node             24728  path:/index.html
24653335.343401998 node             24728  path:/images/welcome.png
24653340.510164998 node             24728  path:/images/favicon.png
```

Supported Frameworks

- MySQL : `--enable-dtrace` (Build)
- JVM : `-XX:+ExtendedDTraceProbes` (Runtime)
- Node : `--with-dtrace` (Build)
- Python : `--with-dtrace` (Build)
- Ruby : `--enable-dtrace` (Build)

eBPF for Tracing

BPF Maps – Filters, States, Counters

Program Excerpt
tcpv4connect.py

Key

Value type

```
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <net/sock.h>
#include <bcc/proto.h>

BPF_HASH(currsock, u32, struct sock *);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk)
{
    u32 pid = bpf_get_current_pid_tgid();
    // stash the sock ptr for lookup on return
    currsock.update(&pid, &sk);
    return 0;
};
.
.
.
```

Update hash map

eBPF for Tracing

BPF Maps – Filters, States, Counters

Program Excerpt
tcpv4connect.py

```
int kretprobe__tcp_v4_connect(struct pt_regs *ctx)
{
    int ret = PT_REGS_RC(ctx);
    u32 pid = bpf_get_current_pid_tgid();
    struct sock **skpp;
    skpp = currsock.lookup(&pid);
    if (skpp == 0) {
        return 0;    // missed entry
    }
    if (ret != 0) {
        // failed to send SYNC packet, may not have populated
        currsock.delete(&pid);
        return 0;
    }

    struct sock *skp = *skpp;
    u32 saddr = 0, daddr = 0;
    u16 dport = 0;
    bpf_probe_read(&saddr, sizeof(saddr), &skp->__sk_common.skc_rcv_saddr);
    bpf_probe_read(&daddr, sizeof(daddr), &skp->__sk_common.skc_daddr);
    bpf_probe_read(&dport, sizeof(dport), &skp->__sk_common.skc_dport);
    bpf_trace_printk("trace_tcp4connect %x %x %d\\n", saddr, daddr, ntohs(dport));
    currsock.delete(&pid);
    return 0;
}
"""
```

ax reg

Get Key

Lookup

Delete

Read stuff from sock ptr

Delete

eBPF for Tracing

BPF Maps – Filters, States, Counters

Output

```
# ./tcpv4connect.py
PID    COMM         SADDR            DADDR            DPORT
1479   telnet       127.0.0.1        127.0.0.1        23
1469   curl         10.201.219.236   54.245.105.25    80
1469   curl         10.201.219.236   54.67.101.145    80
```

More Uses

- Record latency (Δt)
  - `biosnoop.py`
- Flags for keeping track of events
  - `kvm_hypercall.py`
- Counting events, histograms
  - `cachestat.py`
  - `cpudist.py`

eBPF for Tracing

BPF Perf Event Output

- Build perf events and save to per-cpu perf buffers

Program Excerpt

```
prog = """
#include <linux/sched.h>
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>
struct data_t {
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
    char fname[NAME_MAX];
};
BPF_PERF_OUTPUT(events);
int handler(struct pt_regs *ctx) {
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid();
    data.ts = bpf_ktime_get_ns();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    bpf_probe_read(&data.fname, sizeof(data.fname),
                   (void *)PT_REGS_PARM1(ctx));
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
"""
```

Event Struct

Init Event

Build Event

Send to buffer

eBPF Trace Visualization

Current State

- Using ASCII histograms, ASCII escape codes
- eBPF trace driven Flamegraphs

Output
argdist.py

```
# ./argdist -H 'p:c:write(int fd, void *buf, size_t len):size_t:len:fd==1'
[01:47:19]
p:c:write(int fd, void *buf, size_t len):size_t:len:fd==1
 len :  count    distribution
   0 -> 1   : 0  |                                        |
   2 -> 3   : 0  |                                        |
   4 -> 7   : 0  |                                        |
   8 -> 15  : 3  |*********                               |
  16 -> 31  : 0  |                                        |
  32 -> 63  : 5  |***************                         |
  64 -> 127 : 13 |****************************************|
```

eBPF Trace Visualization

Current State

- Using ASCII histograms, ASCII escape codes
- eBPF Flamegraphs, some web-based views

Further Reading

Papers

[Begel et al. 1999] BPF+: exploiting global data-flow optimization in a generalized packet filter architecture, *ACM SIGCOMM '99*

[Wu et al. 2008] Swift: A Fast Dynamic Packet Filter, *USENIX NSDI (2008)*

[Sharma et al. 2016] Enhanced Userspace and In-Kernel Trace Filtering for Production Systems, *J. Comput. Sci. Technol. (2016), Springer US*

[Clément 2016] Linux Kernel packet transmission performance in high-speed networks, *Masters Thesis (2016), KTH, Stockholm*

[Borkmann 2016] Advanced programmability and recent updates with tc's cls_bpf, *NetDev 1.2 (2016) Tokyo*

References

Links

- IOVisor BPF Docs
- bcc Reference Guide
- bcc Python Developer Tutorial
- bcc/BPF Blog Posts
- Dive into BPF: a list of reading material (Quentin Monnet)
- Cilium - Network and Application Security with BPF and XDP (Thomas Graf)
- Landlock LSM Docs (Mickaël Salaün et al.)
- XDP for the Rest of Us (Jesper Brouer & Andy Gospodarek, Netdev 2.1)
- USDT/BPF Tracing Tools (Sasha Goldshtein)
- Linux 4.x Tracing : Performance Analysis with bcc/BPF (Brendan Gregg, SCALE 15X)
- BPF/bcc for Oracle Tracing
- Weaveworks Scope HTTP Statistics Plugin

Ack

DORSAL Lab, Polytechnique Montréal
IOVisor Project Contributors
Hopper.com
Papers We Love

Fin!

suchakrapani.sharma@polymtl.ca
@tuxology