A gentle introduction to

[e]BPF

michael@kinvolk.io

About me

- Software Engineer at Kinvolk in Berlin
- We work mostly on Linux system-level and cloud software

  `{Containers, Kubernetes, Kernel}`
- https://kinvolk.io

```
BPF(2)          Linux Programmer's Manual          BPF(2)

NAME
       bpf  -  perform a command on an extended BPF map
       or program

SYNOPSIS
       #include <linux/bpf.h>

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

DESCRIPTION
       The bpf() system call performs a range of opera-
       tions  related  to extended Berkeley Packet Fil-
       ters.
```

Agenda

- History: classic → extended BPF
- Architecture
- Instruction Set
- Development
- Tools

What is BPF?

often described as in-kernel bytecode virtual machine or engine

Used tcpdump? → (classic) BPF user

1992: The BSD Packet Filter

A New Architecture for User-level Packet Capture

Classic BPF

- 32 bit accumulator A
- 32 bit register X
- 16x 32 bit memory store `M[]`

```
tcpdump -p -ni wlp4s0 -d \
  "ip and tcp and dst port 80"
```

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 10
(002) ldb      [23]
(003) jeq      #0x6             jt 4    jf 10
(004) ldh      [20]
(005) jset     #0x1fff          jt 10    jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 16]
(008) jeq      #0x50            jt 9    jf 10
(009) ret      #262144
(010) ret      #0
```

BPF tools in Linux by D. Borkman

- `tools/net/bpf_asm.c`
- `tools/net/bpf_dbg.c`

bpftools by Cloudflare

- Helper tools to create BPF rules (e.g. from pcap dumps)
- https://github.com/cloudflare/bpftools
- iptables `xt_bpf`

Today: [e]xtended BPF

- Richer instruction set, more features, more use cases
  - Networking (XDP)
  - Tracing (tracepoints, kprobes, etc)
  - Security

Design properties

- fast, performance equal to native code
- no overhead for calls into/from BPF
- ...

Architecture

- a general purpose instruction set
- eleven 64 bit registers

  r0 ... r10

- a program counter
- 512 bytes stack

Architecture

r0	return value from in-kernel function + exit value for eBPF program
r1 - r5	arguments from eBPF program to in-kernel function
r6 - r9	callee saved registers that in-kernel function will preserve
r10	read-only, holds the frame pointer address

Architecture

- Maps as key/value stores
- Helper functions by the kernel
- Tail calls into other BPF programs
- Pseudo filesystems `/sys/fs/bpf`

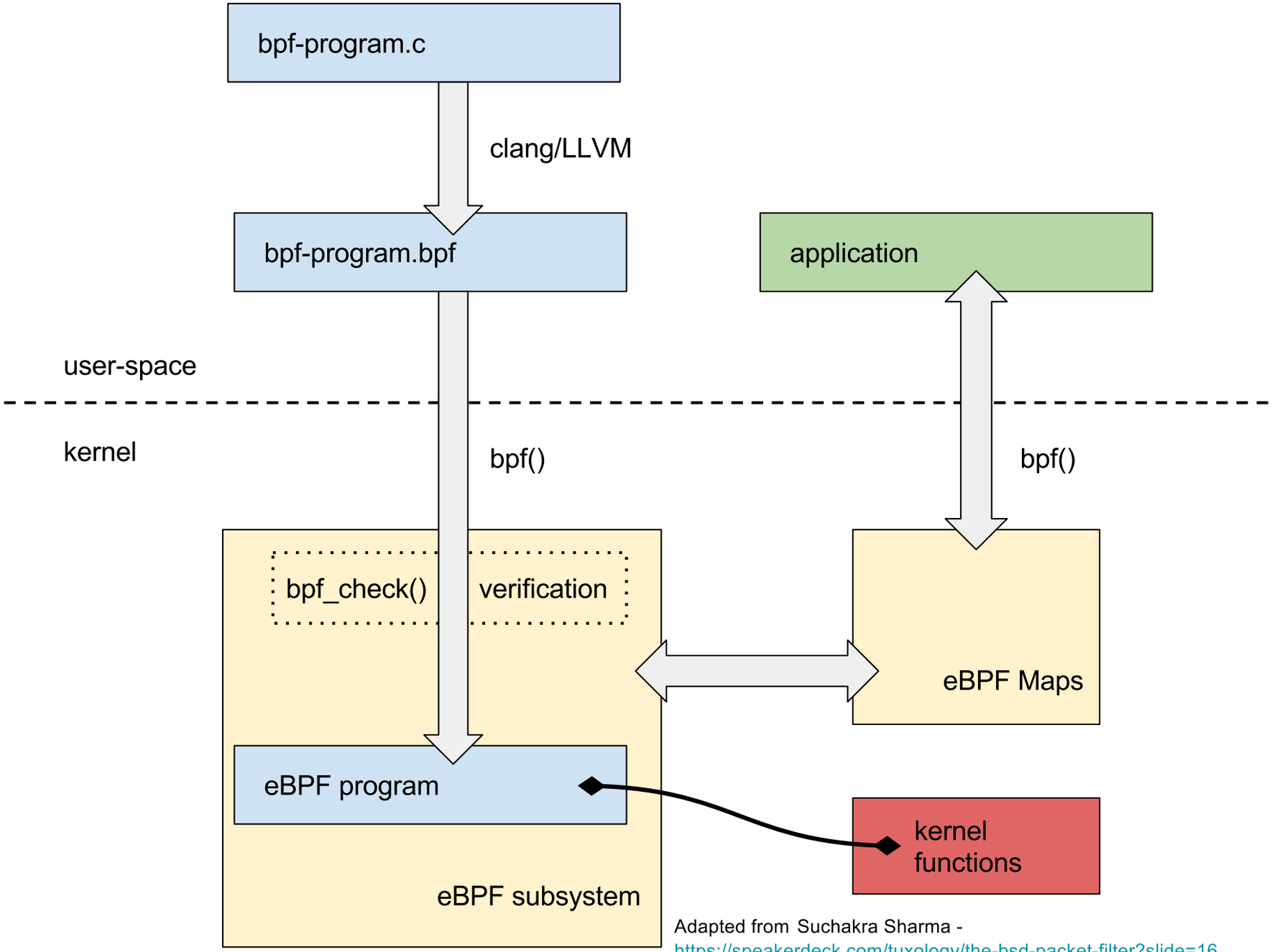
The verifier

- 4096 instructions limit
- no loops, unreachable instructions, etc.
- kernel/bpf/verifier.c#L24

`bpf(2)` syscall

All interaction happens through syscall with `union bpf_attr`

```
syscall(__NR_bpf, ..., &attr, sizeof(attr))
```



Adapted from Suchakra Sharma -
https://speakerdeck.com/tuxology/the-bsd-packet-filter?slide=16

A first program

eBPF program

```
struct bpf_insn prog[] = {
    BPF_MOV64_IMM(BPF_REG_0, 11),
    BPF_EXIT_INSN(),
};
```

bpf_attr for loading

```
union bpf_attr attr = {
    .prog_type = BPF_PROG_TYPE_SCHED_CLS,
    .insn_cnt = sizeof(prog) / sizeof(struct bpf_insn),
    .insns = (__u64) (unsigned long) prog,
    .license = (__u64) (unsigned long) license,
};
```

Loading the program

```
fd = syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr));
```

```
struct bpf_insn {
    __u8    code;           /* opcode */
    __u8    dst_reg:4;      /* dest register */
    __u8    src_reg:4;      /* source register */
    __s16   off;            /* signed offset */
    __s32   imm;            /* signed immediate constant */
};
```

```
#define BPF_MOV64_IMM(DST, IMM)
    ((struct bpf_insn) {
        .code    = BPF_MOV | BPF_K | BPF_ALU64,
        .dst_reg = DST,
        .src_reg = 0,
        .off     = 0,
        .imm     = IMM })
```

Opcode encoding

```
// arithmetic and jump instructions
  +----------------+--------+--------------------+
  |   4 bits       |  1 bit |   3 bits           |
  | operation code | source | instruction class  |
  +----------------+--------+--------------------+
  (MSB)                                    (LSB)

// load and store instructions
  +--------+--------+--------------------+
  | 3 bits | 2 bits |   3 bits           |
  |  mode  |  size  | instruction class  |
  +--------+--------+--------------------+
  (MSB)                           (LSB)
```

Using the verification `log_buf`

```
union bpf_attr attr = {
    ...
    .log_buf = (__u64) (unsigned long) log_buf,
    .log_size = sizeof(log_buf),
    .log_level = 2,
}
```

```
0: R1=ctx R10=fp
0: (b7) r0 = 11
1: R0=imm11,min_value=11,max_value=11,min_align=1 R1=ctx R10=fp
1: (95) exit
processed 2 insns, stack depth 0
```

Some program types must match kernel version

```
union bpf_attr attr = {
    ...
    .kern_version = LINUX_VERSION_CODE,
}
```

Maps

- eBPF offers different types of maps, e.g.
  - `BPF_MAP_TYPE_HASH`
  - `BPF_MAP_TYPE_PROG_ARRAY`
- Maps are used for user-space - kernel-space data passing

Maps

```
struct bpf_map_def SEC("maps/syscall_count") syscall_count = {
        .type = BPF_MAP_TYPE_PERCPU_HASH,
        .key_size = sizeof(__u32),
        .value_size = sizeof(__u64),
        .max_entries = 1024,
};
```

Using clang/LLVM

- LLVM (>= 3.7) has a BPF backend

  → programs can be written in C

Everything needs to be inlined

```
#ifndef __inline
#define __inline \
  inline __attribute__((always_inline))
#endif
```

`printk` debugging

```
#define printt(fmt, ...)                                        \
    ({                                                          \
        char ____fmt[] = fmt;                                   \
        bpf_trace_printk(____fmt, sizeof(____fmt), ##__VA_ARGS__); \
    })
```

```
cat /sys/kernel/debug/tracing/trace_pipe
```

Add DWARF info

- clang >= 4 can add DWARF info, use `-g`
- `llvm-objdump` to get assembler annotated with C code
  - corresponds to output of kernel verifier log

Demo `syscount`

Tools

- bcc (BPF Compiler Collection) toolkit
  - includes C wrapper around LLVM
  - Python + Lua frontends
- clang/LLVM to build .elf files
- gobpf to load and use eBPF from Go
- ...

BPF features by Linux version

https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

`bpf_prog_test_run`

- Since Linux 4.12
- Test run `skb` and `xdp` programs
- More prog types yet to be done

bpf_prog_test_run

```
attr.test.prog_fd = fd;
attr.test.data_in = ptr_to_u64((void *) data);
attr.test.data_out = ptr_to_u64((void *) data_out);
attr.test.data_size_in = data_size;
attr.test.repeat = repeat;

ret = syscall(__NR_bpf, BPF_PROG_TEST_RUN, &attr, sizeof(attr));
if (data_out_size)
                *data_out_size = attr.test.data_size_out;
if (retval)
                *retval = attr.test.retval;
if (duration)
                *duration = attr.test.duration;
```

sysctl options

```
// enable JIT compiler
net.core.bpf_jit_enable

// mitigate JIT spraying
net.core.bpf_jit_harden

// export to /proc/kallsyms
net.core.bpf_jit_kallsysms
```

Tracing with eBPF

- http://www.brendangregg.com/ebpf.html

Projects using eBPF

- https://github.com/cilium/cilium
- https://github.com/weaveworks/tcptracer-bpf
- https://github.com/pmem/vltrace

BPF_EXIT_INSN()

Questions?

Slides can be found here soon: https://speakerdeck.com/schu

michael@kinvolk.io

Resources

- http://www.tcpdump.org/papers/bpf-usenix93.pdf
- https://www.kernel.org/doc/Documentation/networking/filter.txt
- http://docs.cilium.io/en/latest/bpf/
- https://blog.cloudflare.com/bpf-the-forgotten-bytecode/
- https://blog.cloudflare.com/introducing-the-bpf-tools/
- https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

Resources

- https://github.com/iovisor/bcc
- https://github.com/iovisor/gobpf