eBPF Tooling and Debugging Infrastructure

Quentin Monnet

Fall eBPF Webinar Series • 2018-10-09

© Netronome 2018

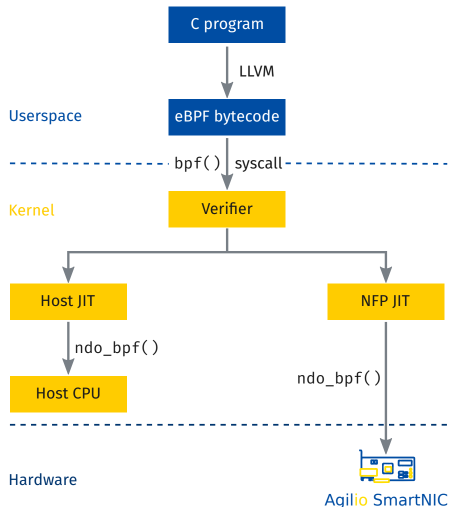
Injecting Programs into the Kernel

NETRONOME

- eBPF programs are usually compiled from C (or Go, Rust, Lua...) to eBPF bytecode
- They are injected into the kernel with the `bpf()` system call
- Safety and termination are ensured by the kernel verifier
- Programs can be JIT (Just-In-Time) compiled
- Once loaded, programs can be attached to a hook in the kernel (socket, TC, XDP...)

eBPF for Network Packet Processing

NETRONOME

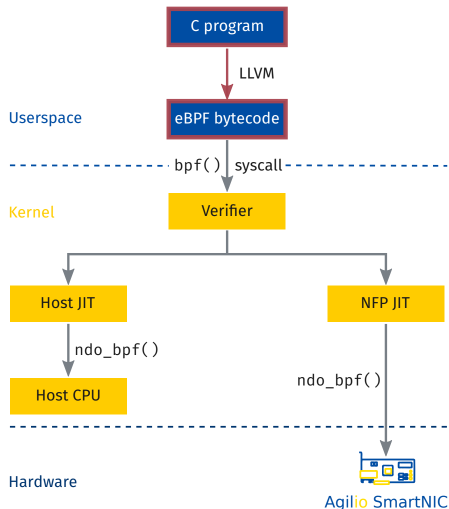


NETRONOME

- Short reminder about eBPF infrastructure and program loading... DONE
- Understand the basic tools available for working with eBPF
- Understand how to dump the eBPF instructions at the different stages of the process
- Learn how to avoid some common mistakes
- Learn where to find more resources for troubleshooting other issues

eBPF for Network Packet Processing

NETRONOME



From C to eBPF Bytecode: Tools

- Compile with clang

```
$ clang -O2 -emit-llvm -c sample_ret0.c -o - | \
      llc -march=bpf -mcpu=probe -filetype=obj -o sample_ret0.o
```

- Dump with llvm-objdump (v4.0+)

```
$ llvm-objdump -d -r -print-imm-hex sample_ret0.o

sample_ret0.o:  file format ELF64-BPF

Disassembly of section .text:
func:
       0:       b7 00 00 00 00 00 00 00         r0 = 0
       1:       95 00 00 00 00 00 00 00         exit
```

- If `-g` is passed to clang, `llvm-objdump -S` can dump the original C code

From C to eBPF Bytecode: Common Mistakes

- Unroll loops

```
next_iph_u16 = (u16 *)iph;

#pragma clang loop unroll(full)
for (i = 0; i < sizeof(*iph) >> 1; i++)
    csum += *next_iph_u16++;
```

- Force function inlining on older kernels (before v4.16)
- Functional errors will be detected only at load time by the verifier

From C to eBPF, in Two Steps: eBPF Assembly

NETRONOME

- Compile from C to eBPF assembly file

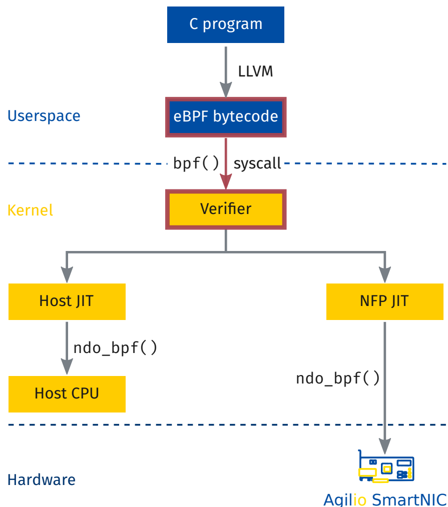
```
$ clang -target bpf -S -o sample_ret0.S sample_ret0.c
$ cat sample_ret0.S
        .text
        .globl  func                    # -- Begin function func
        .p2align        3
func:                                   # @func
# %bb.0:
        r0 = 0
        exit
                                        # -- End function
```

- ... Hack...
- Then compile from assembly to eBPF bytecode (LLVM v6.0+)

```
$ clang -target bpf -c -o sample_ret0.o sample_ret0.S
```

eBPF for Network Packet Processing

NETRONOME



NETRONOME

- TC hook: create a qdisc and attach the program as a filter, with `tc`

```
# tc qdisc add dev eth0 clsact
# tc filter add dev eth0 ingress bpf \
      object-file bpf_program.o section ".text" direct-action
# tc filter show dev eth0 ingress
filter pref 49152 bpf chain 0
filter pref 49152 bpf chain 0 handle 0x1 sample_ret0.o:[.text] \
id 73 tag b07f8eff09a9a611
```

- XDP: attach to the driver (or as "generic XDP") with `ip link`

```
# ip -force link set dev eth0 xdp object sample_ret0.o section ".text"
# ip link show dev eth0
11: eth0: <BROADCAST,NOARP> mtu 1500 xdpoffload qdisc noop state DOWN \
mode DEFAULT group default qlen 1000
    link/ether 0e:41:b5:45:47:51 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 74 tag 704bfda100a6df93
```

Loading eBPF: Common Mistakes

NETRONOME

- tc, ip: `Error fetching program/map!`
  → Make sure to pass the correct section name (defaults to `".text"`)
- With `tc`, `direct-action` (`da`) option is recommended (mandatory for offload): makes TC consider return values as actions (pass, drop...) instead of queues id.
- `RTNETLINK answers: Device or resource busy`
  → `-force` option with `ip link` to overwrite a program previously loaded
- Make sure your version of iproute2 is recent enough
  If in doubt, download and compile the latest version
  For offload: v4.18 (iproute2-ss180813) recommended for perf map support
  (see also Netronome *eBPF – Getting Started Guide*)

The Kernel eBPF Verifier: Checking Programs for Safety

The verifier performs many checks on control flow graph and individual instructions
It complains about:

- Erroneous syntax (unknown instruction, incorrect usage for the instruction)
- Too many instructions or maps or branches
- Back edges (i.e. loops) in the control flow graph
- Unreachable instructions
- Jump out of range
- Out of bounds memory access (data or stack, including passing stack pointers to functions)
- Access to forbidden context fields (read or write)
- Reading access to non-initialized memory (stack or registers)
- Use of forbidden helpers for the current type of program
- Use of GPL helpers in non-GPL program (mostly tracing)
- `R0` not initialized before exiting the program
- Memory access with incorrect alignment
- Missing check on result from `map_lookup_elem()` before accessing map element
- ...

Problem: error messages are not always easy to understand. Examples...

Extending Kernel eBPF Verifier for Offload

NETRONOME

- The NFP driver hooks into the verifier to add its own checks, but output any error in the console just as the kernel verifier does

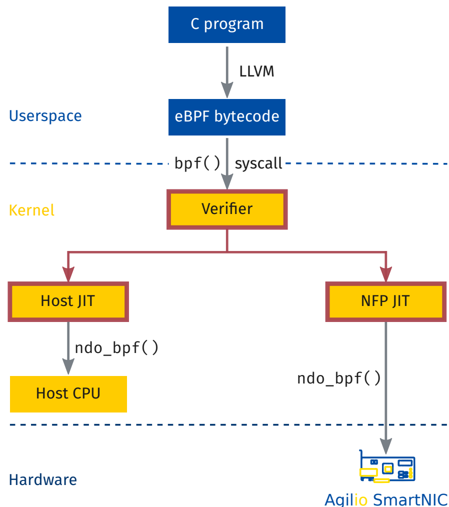
```
Verifier Analysis:

    0: (b7) r2 = 0x32
    1: (07) r2 += -8
    2: (b7) r1 = 0x0
    3: (85) call 6

    [nfp] unsupported function id: 6
```

eBPF for Network Packet Processing

NETRONOME



JIT-compile eBPF Programs for the Hardware

- Most programs will be offloaded smoothly if they have passed the verifiers.
- Some error messages at JIT-compiling time cannot reuse the verifier buffer, they are sent to the kernel logs (in `/var/log/kernel`, or print with `dmesg`)

```
[88613.915838] nfp 0000:04:00.0 nfp_p0: stack too large: program 576B > FW stack 512B
```

Program is Loaded: Introspection

NETRONOME

We have passed the verifier! The program is loaded in the kernel

- For map and program introspection: bpftool
  - List maps and programs
  - Load a program, pin it
  - Dump program instructions (eBPF or JIT-ed)
  - Dump and edit map contents
  - etc.

Program is Loaded: Introspection (Program on the Host)

NETRONOME

- Dump kernel-translated instructions

```
# bpftool prog dump xlated id 4
    0: (b7) r0 = 0
    1: (95) exit
```

- Dump JIT-ed instructions

```
# bpftool prog dump jited id 4
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   sub    $0x28,%rsp
   b:   sub    $0x28,%rbp
   f:   mov    %rbx,0x0(%rbp)
  13:   mov    %r13,0x8(%rbp)
  [...]
  33:   mov    0x18(%rbp),%r15
  37:   add    $0x28,%rbp
  3b:   leaveq
  3c:   retq
```

Program is Loaded: Introspection (Offloaded Program)

NETRONOME

- Dumping instructions of an offloaded program works exactly the same:

```
# bpftool prog dump jited id 4
   0:     .0  immed[gprB_6, 0x3fff]
   8:     .1  alu[gprB_6, gprB_6, AND, *l$index1]
  10:     .2  immed[gprA_0, 0x0], gpr_wrboth
  18:     .3  immed[gprA_1, 0x0], gpr_wrboth
  20:     .4  br[.15000]
  [...]
```

NFP support for disassembler available in latest version of libbfd (binutils-dev v2.31)

Attaching eBPF

NETRONOME

In our examples, attaching was actually performed by `tc` and `ip link` right after program load

- Netlink "extended ack" (*extack*) messages in the console
  Example: `RTNETLINK answers: Device or resource busy`
- Same thing for offloaded programs

Debugging at Runtime with bpf_trace_printk()

- eBPF helper `bpf_trace_printk()` prints to `/sys/kernel/debug/tracing/trace`

```
const char fmt[] = "First four bytes of packet: %x\n";
bpf_trace_printk(fmt, sizeof(fmt), *(uint32_t *)data);
```

Debugging at Runtime with Perf Events

- Also, support for “perf event arrays”, more efficient
  Example: dump data from packet

```
struct bpf_map_def SEC(”maps”) pa = {
        .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        .key_size = sizeof(int),
        .value_size = sizeof(int),
        .max_entries = 64,
};

int xdp_prog1(struct xdp_md *xdp)
{
        int key = 0;
        bpf_perf_event_output(xdp, &pa, 0x20ffffffffULL, &key, 0);
        return XDP_PASS;
}
```

Debugging at Runtime: Miscellaneous

NETRONOME

- No eBPF debugger at this time
- User space interpreters: uBPF, rbpf
  (Minor differences, some features missing, no verifier)

Tools: eBPF Programming Environment

NETRONOME

- Libraries for managing eBPF programs: libbpf (kernel tree, *tools/lib/bpf*), libbcc (bcc tools)
- strace: support for `bpf()` system call
  ```
  strace -e bpf ip link set dev nfp_p0 xdpoffload obj prog.o
  ```
- valgrind: upcoming version (3.14) with support for `bpf()` system call
  ```
  valgrind bpftool prog show
  ```
- ...

Next Steps for eBPF Tooling

NETRONOME

Netronome remains involved! We do or intend to...

- Improve components
  - Error messages
  - Existing tool set
  - Documentation
- Improve packaging
  - bpftool
  - libbpf
- Help keep tools up-to-date
- Create new tools?
  - Thinking about ways to run eBPF in a debugger
  - Maybe some work to do on the side of libpcap

We are not alone: eBPF community increasing, more and more activity!

Summary

NETRONOME

eBPF programs do not run in user space: debugging is not trivial

But:

- Tooling is getting better and better: more tools, more complete
- Possible to dump the instructions at all the stages of the process (llvm-obdjump, bpftool)
- Possible to get some output (`bpf_trace_printk()`, perf event maps) at runtime
- Debugging offloaded programs is nearly the same as for programs on the host

More resources

NETRONOME

- Netronome's *eBPF – Getting Started Guide*
  https://www.netronome.com/documents/305/eBPF-Getting_Started_Guide.pdf
- Partial F.A.Q for verifier output: Kernel documentation (filter.txt)
  https://www.kernel.org/doc/Documentation/networking/filter.txt
- Netronome's resources on eBPF
  https://www.netronome.com/technology/ebpf/
- Netronome's sample eBPF applications
  https://github.com/Netronome/bpf-samples
- Kernel source code
  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/
- Documentation on eBPF helper functions, generated from kernel comments
  https://github.com/iovisor/bpf-docs/blob/master/bpf_helpers.rst

NETRONOME

Thank you!