Making the Kernel's Networking Data Path Programmable with BPF and XDP.

Daniel Borkmann
<daniel@covalent.io>
Covalent IO

OSSNA17, September 12, 2017

What is BPF?

```
tcpdump -i eno1 -ddd icmp or icmp6
12
40 0 0 12
21 0 2 2048
48 0 0 23
21 6 7 1
21 0 6 34525
48 0 0 20
21 3 0 58
21 0 3 44
48 0 0 54
21 0 1 58
6 0 0 262144
6 0 0 0
```

What is BPF?

```
tcpdump -i eno1 -d icmp or icmp6
(000) ldh      [12]
(001) jeq      #0x800          jt 2 jf 4
(002) ldb      [23]
(003) jeq      #0x1            jt 10 jf 11
(004) jeq      #0x86dd         jt 5 jf 11
(005) ldb      [20]
(006) jeq      #0x3a           jt 10 jf 7
(007) jeq      #0x2c           jt 8 jf 11
(008) ldb      [54]
(009) jeq      #0x3a           jt 10 jf 11
(010) ret      #262144
(011) ret      #0
```

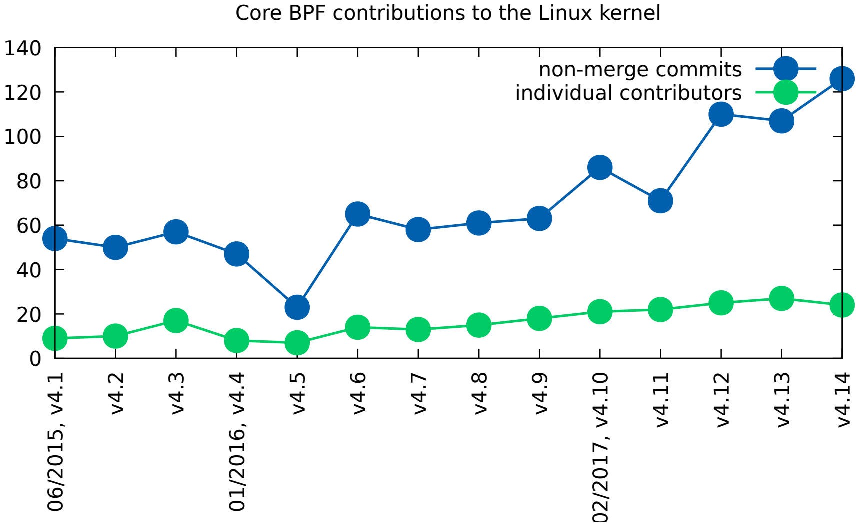
BPF back then.

- Original use-case: tcpdump filter for raw packet sockets
- Filtering as early as possible to avoid wasting resources
- Generic, fast and safe language for packet parsing
  - Protocols often complex and parsers buggy (→ CVEs) ...
  - Hard requirement to ensure stability when run in kernel space
- tcpdump → libpcap → BPF insns → kernel → verifier → interpreter

BPF nowadays.

- Instruction set and infrastructure heavily evolved over last years
- Two flavors of BPF: cBPF and eBPF
- BPF really stands for eBPF these days
    - Superset of cBPF features
    - Kernel migrates all cBPF to eBPF internally
- Not used exclusively in packet sockets anymore!
    - Networking: XDP, tc, socket progs, kcm, reuseport, ...
    - Tracing: kprobes, uprobes, tracepoints, ...
    - Security: seccomp, landlock

BPF nowadays.



BPF in a nutshell.

- 11 64bit registers, 32bit subregisters, up to 512bytes stack
- Instructions 64bit wide, max 4096 per program
- Core components of architecture
  - Read/write access to context
  - Helper function concept
  - Maps, arbitrary sharing
  - Tail calls
  - Object pinning
  - cBPF to eBPF translation
  - LLVM eBPF backend
- eBPF JIT backends implemented by archs
- Orchestration via `bpf(2)`, stable ABI (!)

BPF verifier.

- Providing a verdict for kernel whether safe to run
- Simulation of execution of all paths of the program
- Steps involved (extract):
  - Checking control flow graph for loops
  - Detecting out of range jumps, unreachable instructions
  - Tracking context access, initialized memory, stack spill/fills
  - Checking unpriviledged pointer leaks
  - Verifying helper function call arguments
  - Value and alignment tracking for data access (pkt pointer, map access)
  - Register liveness analysis for pruning
  - State pruning for reducing verification complexity
- Patching BPF programs at post-verification

BPF JITs.

- C → LLVM → BPF → loader → verifier → JIT → tc/XDP → offload
- JITs in kernel: x86_64, arm64, ppc64, mips64, s390x, sparc64, arm32
- Full instruction set supported by all 64 bit JITs
- BPF registers mapped to CPU registers 1:1
- BPF calling convention for helpers allows for efficient mapping
  - R0 → return value from helper call
  - R1 - R5 → argument registers for helper call
  - R6 - R9 → callee saved, preserved on helper call
- /proc/kallsyms exposure of JIT image as symbol for stack traces
- Generic constant blinding for JITs

user space, kernel space

BPF LLVM backend.

- Since LLVM 3.7: `clang -O2 -target bpf -c foo.c -o foo.o`
- Enabled by default with all major distributions
  - Registered targets: `llc --version`
  - `llc`'s BPF `-march` options: bpf, bpfeb, bpfel
  - `llc`'s BPF `-mcpu` options: generic, v1, v2, probe
- Assembler output through `-S` supported
- `llvm-objdump` for disassembler and code annotations (via DWARF)
- Annotations correlate directly with kernel verifier log
- Outputs ELF file with maps as relocation entries
  - Processed by BPF loaders (e.g. iproute2) and pushed into kernel

Restricted C for BPF.

- BPF has slightly different environment for C
  - Helper functions and program context available
  - Program entry points specified by sections
  - One or more entry points in a single object file possible
  - Library functions all get inlined, no notion of function calls (yet)
  - No global variables, no loops (yet) unless unrolled by pragma
  - No const strings or data structures
  - LLVM built-in functions usually available and inlined
  - Partitioning processing path with tail calls
  - Limited stack space up to 512 bytes
- C example walkthrough: tools/testing/selftests/bpf/test_l4lb.c

XDP basics.

- DoS mitigation, forwarding/load balancing, monitoring, preprocessing
- Framework for running BPF programs in driver's RX path
  - Ensures packets are linear, read/writeable
  - 256 bytes headroom for custom encap
  - Atomic replacement of BPF progs during runtime
  - Post-processing of 5 verdicts from BPF prog
    - pass, drop, tx, redirect, aborted
  - Tailored for high-performance close to line-rate
    - XDP LB against IPVS up to 10x better with similar features[1]
- Hook runs at earliest possible point by definition (!)
  - No skb alloc yet, no GRO, etc

[1] Droplet: DDoS countermeasures powered by BPF + XDP

XDP and the kernel.

- Works in concert with the kernel and its infrastructure (!)
- Advantages of XDP
  - Reuses upstream kernel drivers and tooling
  - Same security model as kernel for accessing hardware
  - Allows for flexible structuring of workloads
  - Punting to stable, efficient TCP/IP stack already available
  - No need for crossing boundaries when punting to sockets
  - No third party code/licensing required to use it
  - Shipped everywhere since kernel 4.8

XDP operation modes.

- Offloaded XDP
  - nfp
  - Limited offloading through JIT for NIC
  - `ip link set dev eno1 xdpoffload obj prog.o`
- Native XDP
  - mlx4, mlx5, ixgbe, i40e, nfp, bnxt, thunder, qede, virtio_net, tun
  - Further 10G/40G driver support growing
  - `ip link set dev eno1 xdp obj prog.o`
  - `ip link set dev eno1 xdpdrv obj prog.o`
- Generic XDP
  - All netdevices supported
  - For experimentation purposes, run from stack
  - `ip link set dev eno1 xdpgeneric obj prog.o`

XDP context and helpers (extract).

```
struct xdp_buff {
        void *data;
        void *data_end;
        void *data_hard_start;
};
```

- Direct read/write on `xdp->data`
- Possibility to adjust offset of `xdp->data`
- Generic meta data transfer from XDP to skb (soon)
- Event output through lockless, per-CPU mmap'ed perf ring buffer
  - Direction kernel → user space, e.g. sampling, notifications
  - Ring buffer slot fully programmable
  - Full or truncated packet can be appended

XDP demo.

- Two test workstations, back to back connected
  - Xeon E3-1240, 3.4Ghz (no DDIO), Supermicro X10SLM-F, 16G RAM
  - Spec'ed out for silence and 10G tests approx 4yrs ago
  - ixgbe, nfp for testing
- pktgen attack with random pkts in 10.0.0.0/8, 11.5Mio pps generated
- L3 filter with 16Mio blacklist entries, all of 10.0.0.0/8 as /32s
- Testing latency/throughput for allowed flows
- Demo: part 1, part 2

Thanks!

- BPF/XDP → programmable, high performance networking data path
- Code and more information
  - BPF/XDP core: `https://git.kernel.org` → kernel, iproute2 tree
  - Cilium project: `https://github.com/cilium/cilium`
    - BPF & XDP for containers
    - OSSNA Cilium booth 501
- Further BPF related talks at OSSNA
  - **Cilium - Container Security and Networking Using BPF and XDP**
    - Thomas Graf, Wednesday, 2:50pm @ Diamond Ballroom 6
  - **Performance Analysis Superpowers with Linux BPF**
    - Brendan Gregg, Wednesday, 11:50am @ Diamond Ballroom 3
  - **Our Experiences Deploying Kubernetes with IPv6**
    - André Martins, Wednesday, 2:00pm @ Diamond Ballroom 6