On getting tc classifier fully programmable with cls_bpf.

Daniel Borkmann
<daniel@iogearbox.net>
Noiro Networks / Cisco

netdev 1.1, Sevilla, February 12, 2016

Background, history.

- BPF origins as a generic, fast and 'safe' solution to packet parsing
- tcpdump → libpcap → compiler → bytecode → kernel interpreter
- Intended as early drop point in `AF_PACKET` kernel receive path
- JIT'able for x86_64 since 2011, ppc, sparc, arm, arm64, s390, mips
- BPF used today: networking, tracing, sandboxing

```
# tcpdump -i any -d ip
(000) ldh      [14]
(001) jeq      #0x800          jt 2 jf 3
(002) ret      #65535
(003) ret      #0
```

Classic BPF (cBPF) in a nutshell.

- 32 bit, available register: A, X, M[0-15], (pc)
- A used for almost everything, X temporary register, M[] stack
- Insn: 64 bit (`u16:code, u8:jt, u8:jf, u32:k`)
- Insn classes: ld, ldx, st, stx, alu, jmp, ret, misc
- Forward jumps, max 4096 instructions, statically verified in kernel
- Linux-specific extensions overload `ldb/ldh/ldw` with `k`← `off+x`
- `bpf_asm`: 33 instructions, 11 addressing modes, 16 extensions
- Input data/"context" (`ctx`), e.g. `skb`, `seccomp_data`
- Semantics of exit code defined by application

Extended BPF (eBPF) as next step.

- 64 bit, 32 bit sub-registers, available register: R0-R10, stack, (pc)
- Insn: 64 bit (`u8:code, u8:dst_reg, u8:src_reg, s16:off, s32:imm`)
- New insns: dw ld/st, mov, alu64 + signed shift, endian, calls, xadd
- Forward & backward* jumps, max 4096 instructions
- Generic helper function concept, several kernel-provided helpers
- Maps with arbitrary sharing (user space apps, between eBPF progs)
- Tail call concept for eBPF programs, eBPF object pinning
- LLVM eBPF backend: `clang -O2 -target bpf -o foo.o foo.c`
  - C → LLVM → ELF → tc → kernel (verification/JIT) → cls_bpf (exec)

eBPF, General remarks.

- Stable ABI for user space, like the case with cBPF
- Management via `bpf(2)` syscall through file descriptors
- Points to kernel resource $\rightarrow$ eBPF map / program
- No cBPF interpreter in kernel anymore, all eBPF!
- Kernel performs internal cBPF to eBPF migration for cBPF users
- JITs for eBPF: x86_64, s390, arm64 (remaining ones are still cBPF)
- Various stages for in-kernel cBPF loader
- Security (verifier, non-root restrictions, JIT hardening)

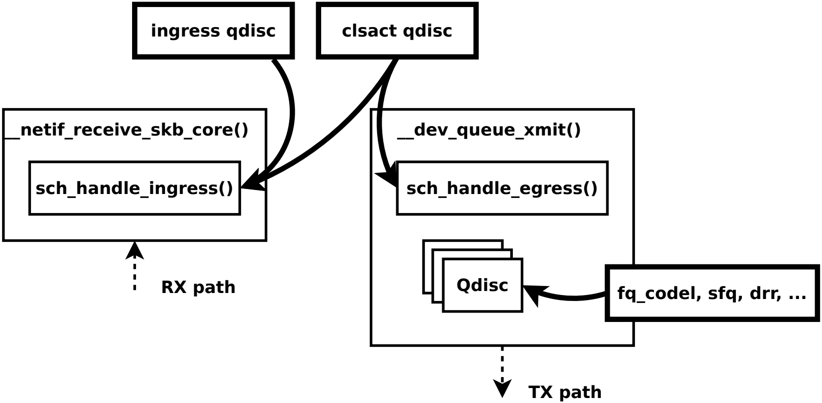
eBPF and cls_bpf.

- cls_bpf as cBPF-based classifier in 2013, eBPF support since 2015
- Minimal fast-path, just calls into `BPF_PROG_RUN()`
- Instance holds one or more BPF programs, 2 operation modes:
  - Calls into full tc action engine `tcf_exts_exec()` for e.g. `act_bpf`
  - Direct-action (DA) fast-path for immediate return after BPF run
- In DA, eBPF prog sets `skb->tc_classid`, returns action code
  - Possible codes: ok, shot, stolen, redirect, unspec
- tc frontend does all the setup work, just sends fd via netlink

eBPF and cls_bpf.

- skb metadata:
  - Read/write: mark, priority, tc_index, cb[5], tc_classid
  - Read: len, pkt_type, queue_mapping, protocol, vlan_*, ifindex, hash
- Tunnel metadata:
  - Read/write: tunnel key for IPv4/IPv6 (dst-meta by vxlan, geneve, gre)
- Helpers:
  - eBPF map access (lookup/update/delete)
  - Tail call support
  - Store/load payload (multi-)bytes
  - L3/L4 csum fixups
  - skb redirection (ingress/egress)
  - Vlan push/pop and tunnel key
  - `trace_printk` debugging
  - `net_cls` cgroup classid
  - Routing realms (`dst->tclassid`)
  - Get random number/cpu/ktime

cls_bpf, Invocation points.



cls_bpf, Example setup in 1 slide.

```
$ clang -O2 -target bpf -o foo.o foo.c

# tc qdisc  add dev em1 clsact
# tc qdisc show dev em1
[...]
qdisc clsact ffff: parent ffff:fff1

# tc filter add dev em1 ingress bpf da obj foo.o sec p1
# tc filter add dev em1 egress  bpf da obj foo.o sec p2

# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 foo.o:[p1] direct-action

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 foo.o:[p2] direct-action

# tc filter del dev em1 ingress pref 49152
# tc filter del dev em1 egress  pref 49152
```

tc frontend.

- Common loader backend for `f_bpf`, `m_bpf`, `e_bpf`
- Walks ELF file to generate program fd, or fetches fd from pinned
- Setup via ELF object file in multiple steps:
  - Mounts bpf fs, fetches all ancillary sections
  - Sets up maps (fd from pinned or new with pinning)
  - Relocations for injecting map fds into program
  - Loading of actual eBPF program code into kernel
  - Setup and injection of tail called sections
- Grafting of existing prog arrays
- Dumping trace pipe

tc eBPF examples, minimal module.

```
$ cat >foo.c <<EOF
   #include "bpf_api.h"

   __section_cls_entry
   int cls_entry(struct __sk_buff *skb)
   {
           /* char fmt[] = "hello prio%u world!\n"; */
           skb->priority = get_cgroup_classid(skb);
           /* trace_printk(fmt, sizeof(fmt), skb->priority); */
           return TC_ACT_OK;
   }

   BPF_LICENSE("GPL");
EOF

$ clang -O2 -target bpf -o foo.o foo.c
# tc filter add dev em1 egress bpf da obj foo.o
# tc exec bpf dbg             # -> dumps trace_printk()

# cgcreate -g net_cls:/foo
# echo 6 > foo/net_cls.classid
# cgexec -g net_cls:foo ./bar  # -> app ./bar xmits with priority of 6
```

tc eBPF examples, map sharing.

```
#include "bpf_api.h"

BPF_ARRAY4(map_sh, 0, PIN_OBJECT_NS, 1);
BPF_LICENSE("GPL");

__section("egress") int egr_main(struct __sk_buff *skb)
{
        int key = 0, *val;
        val = map_lookup_elem(&map_sh, &key);
        if (val)
                lock_xadd(val, 1);
        return BPF_H_DEFAULT;
}

__section("ingress") int ing_main(struct __sk_buff *skb)
{
        char fmt[] = "map val: %d\n";
        int key = 0, *val;
        val = map_lookup_elem(&map_sh, &key);
        if (val)
                trace_printk(fmt, sizeof(fmt), *val);
        return BPF_H_DEFAULT;
}
```

tc eBPF examples, tail calls.

```
#include "bpf_api.h"

BPF_PROG_ARRAY(jmp_tc, JMP_MAP, PIN_GLOBAL_NS, 1);
BPF_LICENSE("GPL");

__section_tail(JMP_MAP, 0) int cls_foo(struct __sk_buff *skb)
{
        char fmt[] = "in cls_foo\n";
        trace_printk(fmt, sizeof(fmt));
        return TC_H_MAKE(1, 42);
}

__section_cls_entry int cls_entry(struct __sk_buff *skb)
{
        char fmt[] = "fallthrough\n";
        tail_call(skb, &jmp_tc, 0);
        trace_printk(fmt, sizeof(fmt));
        return BPF_H_DEFAULT;
}

$ clang -O2 -DJMP_MAP=0 -target bpf -o graft.o graft.c
# tc filter add dev em1 ingress bpf obj graft.o
```

Code and further information.

- Take-aways:
  - Writing eBPF programs for `tc` is really easy
  - Stable ABI, fully programmable for specific use-cases
  - Native performance when JITed!
- Code:
  - Everything upstream in kernel, iproute2 and llvm!
  - Available from usual places, e.g. `https://git.kernel.org/`
- Some further information:
  - Examples in iproute2's `examples/bpf/`
  - `Documentation/networking/filter.txt`
  - Man pages `bpf(2)`, `tc-bpf(8)`

Appendix / Backup.

eBPF, Helper functions.

- Signature: `u64 foo(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)`
- Calling convention:
  - R0 → return value
  - R1-R5 → function arguments
  - R6-R9 → callee saved
  - R10 → read-only frame pointer
- Specification for verifier, example:

```
static const struct bpf_func_proto foo_proto = {
        .func           = foo,
        .gpl_only       = false,
        .ret_type       = RET_INTEGER,
        .arg1_type      = ARG_CONST_MAP_PTR,
        .arg2_type      = ARG_PTR_TO_MAP_KEY,
        .arg3_type      = ARG_PTR_TO_MAP_VALUE,
        .arg4_type      = ARG_ANYTHING,
};
```

eBPF, Helper functions.

- eBPF program
  - Populates `R1 - R5` depending on specification
  - `BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_foo)`
  - Reads out `R0` if needed
  - Can only use core kernel provided `BPF_FUNC_*` helpers
- Kernel space
  - eBPF verification step
  - Mapping of `BPF_FUNC_*` (`insn->imm`) to `struct bpf_func_proto`
  - Call fixup: `insn->imm = fn->func - __bpf_call_base;`
  - Invocation: `R0 = (__bpf_call_base + insn->imm)(R1, ..., R5);`
  - JITing rather straight forward, x86_64 → 1:1 mapping to HW registers

eBPF, Maps.

- Lightweight key/value store for keeping state
  - Generic, efficient data structures
    - Array, hash table, (per CPU variants soon)
  - Application-specific data structures
    - Program array, perf event array
- Map creation only from user space → `bpf(2)`
- Map access for lookup, update, delete:
  - User space application → `bpf(2)` with fd
  - eBPF program → helper functions

eBPF, Maps.

- eBPF loader/program
  - Map mostly used in `R1` as type `ARG_CONST_MAP_PTR`
  - Loader fetches map fd via `bpf(2)`
  - Rewrites instruction `BPF_LD_MAP_FD(BPF_REG_1, fd)`
  - Expands to double `bpf_insn` `BPF_LD | BPF_IMM | BPF_DW`
  - First part holds `.src_reg = BPF_PSEUDO_MAP_FD, .imm = fd`
- Kernel space
  - eBPF verification step
  - Recognizes `BPF_PSEUDO_MAP_FD` keyword
  - Fetches real map from process fd table
  - Stores actual map pointer in `BPF_LD | BPF_IMM | BPF_DW`

eBPF, Tail calls.

- Idea: allow eBPF programs to call other eBPF programs
- No return to old program, same stack frame used (think of long jump)
- Consists of 2 components:
  - Program array map, populated by user space with eBPF fds
  - eBPF helper: `bpf_tail_call(ctx, &jmp_table, index)`
- Kernel caches actual pointers to map, updates `xchg()`'ed
- Kernel translates `BPF_FUNC_tail_call` into instructions
- Fall-through when lookup failed, otherwise `insn = prog->insnsi`
- Powerful concept for live eBPF program updates, dispatching protocol parsers, etc

eBPF, Object pinning.

- Everything being tied to fds → thus, tied to program livetime
- Makes f.e. eBPF map sharing cumbersome
- Option 1: UDS
  - File descriptor passing, works in general with eBPF fds
  - Requires deploying extra daemon for each application
- Option 2: small special purpose fs (utilized by tc)
  - Maps/programs can be pinned via `bpf(2)` as fs node
  - Picked up via `bpf(2)` again, point to same map/program
  - No difference to "normal" created `bpf(2)` fds
  - fs per mountns, supports bind-mounts, hard links, etc

eBPF, Security.

- Aim for BPF is to be "safe" as in "cannot crash the kernel" ;)
- Primary job of the verifier, eBPF one more complex
  - Checks for cyclic prog flow, uninitialized mem, dead code, types, etc
- `CONFIG_DEBUG_SET_MODULE_RONX` on x86_64, arm, arm64, s390
  - Locks down an entire eBPF program as RO for its lifetime
  - When JITed, locks module memory as RO and randomizes start address
  - Near future: constant blinding to mitigate JIT spraying
  - JIT switch: `sysctl net.core.bpf_jit_enable`
- eBPF restricted for unprivileged programs (socket filters)
  - Very few helpers allowed (map access, tail calls, and few others)
  - Restrictions on pointers (no arithmetic, passing to helpers, etc)
  - Once switch: `sysctl kernel.unprivileged_bpf_disabled`

eBPF, LLVM.

- And most importantly: `clang -O2 -target bpf -o foo.o foo.c`
- eBPF progs written in "restricted C", other frontends possible (P4)
- Compiled to eBPF insns by LLVM (since 3.7), outputs ELF file
  - `clang -O2 -target bpf -c foo.c -S -o -`
  - `readelf -a foo.o`, `readelf -x ... foo.o`
- ELF file → container for map specs, program code, license, etc
- Holds everything for "loaders" like `tc` to get it into kernel
- Typical workflow, example:
  - C → LLVM → ELF → tc → kernel (verification/JIT) → cls_bpf (exec)