Modernize BPF for the next 10 years

Alexei Starovoitov

Meta

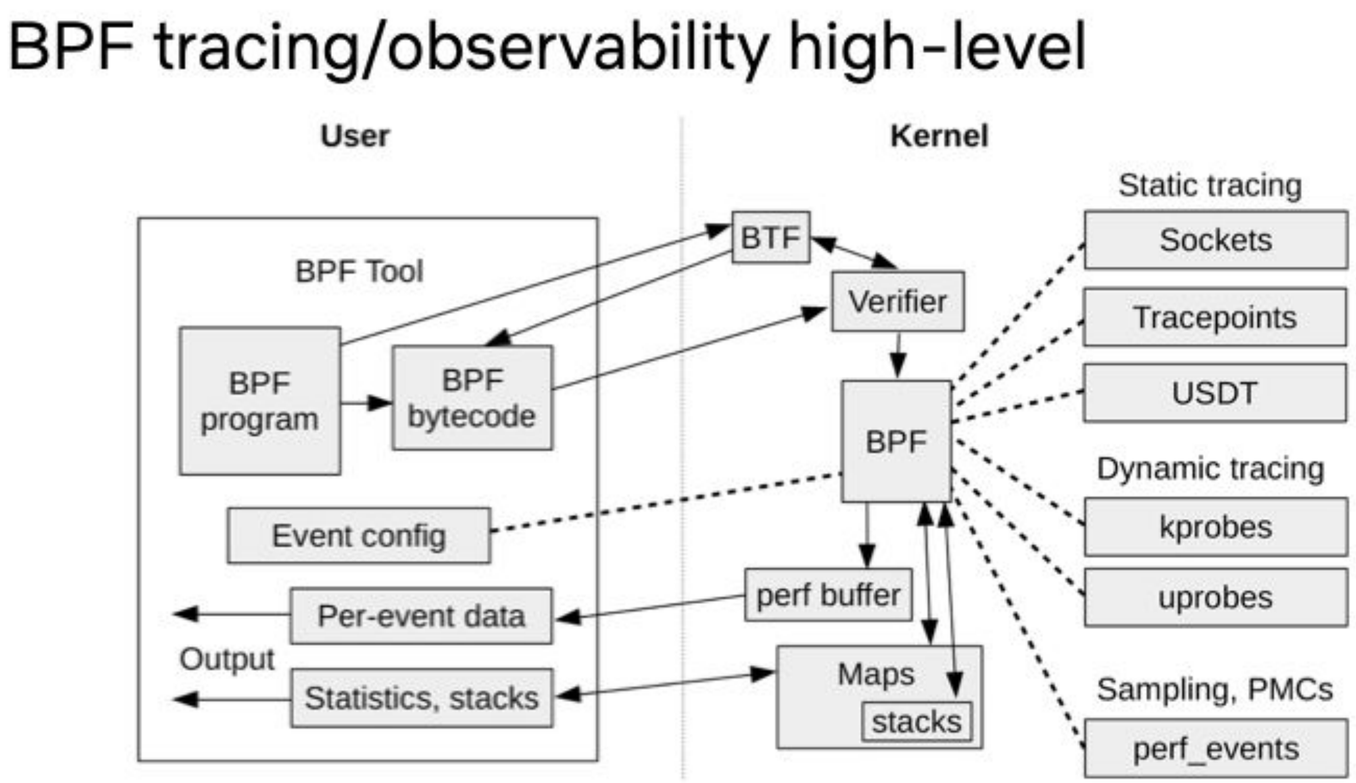
Agenda

- 10 years of BPF development in 10 minutes
- What comes next

2014... Initial use cases

Why: Programmable networking

Solution: BPF programs called when packet is received



Solution: kernel event triggers BPF program that collects stack trace, reads kernel memory, passes info to the tool

Tools: bpftrace, bcc, ...

BPF networking evolution

Problem: sk_buff adds overhead. Cannot reach 10G line rate while user space networking can.

Solution: Run BPF programs in NIC driver aka XDP

Tools: katran, cilium

BPF tracing evolution

Problem: tracing progs read kernel data structures that change in different kernels.

Tools have to include kernel headers and compile bpf prog on the server where it will run.

Solution: BTF and CO-RE

- Annotate vmlinux with type info (BTF)
- Teach clang to generate symbolic field access instead of fixed offset
- Teach libbpf and the kernel to relocate BPF ELF object while loading

BPF usability evolution

Problem: Natural way to write programs in C is to use global variables that compilers put into .data/.bss/.rodata sections. To user space such sections are opaque array maps of one element and user space needs "layout information" to interact with such variables.

Solution: BTF and skeleton

- Compilers annotate .data/.bss/.rodata sections with BTF (information about types and offsets of all variables)
- libbpf/bpftool generate "skeleton" from bpf ELF object which is .h file with structs/names matching the section layout picked by the compiler
- User space #include "my_prog.skel.h"

```
skel = my_prog__open_and_load();
skel->bss->my_var;
```

__attribute__((always_inline)) is no longer necessary.

The verifier supports global and static functions.

```
static ~~__always_inline~~ __u32 get_packet_hash(struct packet_description *pckt,
                                                 bool ipv6)
{
    // ..
}

static ~~__always_inline~~ bool get_packet_dst(struct real_definition **real,
                                               struct packet_description *pckt,
                                               struct vip_meta *vip_info,
                                               bool is_ipv6)
{
      __u32 hash = get_packet_hash(pckt, is_ipv6) % RING_SIZE;
      // ...
}

static ~~__always_inline~~ int parse_icmpv6(void *data, void *data_end, __u64 off,
                                            struct packet_description *pckt)
{
```

```
// in 2014 loops were not supported
#pragma clang loop unroll(full)
for (i = 0; i < 100; i++) {


// in 2019 bounded loops were added
for (i = 0; i < 100; i++) {


// in 2021 bpf_loop() helper was added
bpf_loop(10000, callback_fn, NULL, 0);


// in 2023 open coded iterators were added
bpf_for(i, 0, 10000) {


// in 2024 cond_break was added
for (i = 0; i < 10000; i++) {
        cond_break;

for (;;) {
        cond_break;
```

BPF kernel interface evolution (bpf -> kernel)

Why: helper functions (bpf_map_lookup_elem(), bpf_probe_read_kernel(), ...) have hard coded IDs and fixed UAPI.

Solution: Stop adding helpers, introduce kfunc mechanism.

kfunc is an unstable interface between bpf programs and the kernel.

The kernel modules can define their own kfuncs.

```
git grep "FN(" include/uapi/linux/bpf.h      211 helpers
git grep '^__bpf_kfunc\>'                    164 kfuncs
```

BPF kernel interface evolution (kernel -> bpf)

Why: XDP, TC, other networking hooks are stable UAPI.

There are existing callback mechanisms like tcp_congestion_ops that kernel modules can use.

Solution: Introduce struct-ops.

struct-ops mechanism allows a set of bpf programs register as callbacks.

Initially implemented to plug into tcp_congestion_ops and allow TCP congestion control to be implemented as BPF programs.

Not an UAPI.

Upcoming struct-ops users: sched-ext, hid-bpf, fuse-bpf, qdisc-bpf.

Key challenge: translating kernel (cpu native) calling convention to BPF and vice versa

sched-ext demands efficient data sharing

Why: BPF schedulers need access to cpumask, task_struct, ...

Solution: Introduce kptr: a pointer to kernel data structure.

- bpf rb tree and bpf link list
- bpf_obj_new
- Exclusive and shared ownership

Towards arbitrary algorithms and data structures

Why: hash, array, rb tree, link list were added. Never ending race to add algorithms.

Solution: Introduce bpf_arena, cond_break to implement any algorithm as bpf program.

bpf_arena - sparse shared memory region between bpf prog and user space.
Different __attribute__((address_space)) from LLVM pov.
Dynamic memory access sandboxing.

cond_break - runtime termination of loops

Towards BPF libraries

Demonstrated hash table, regex, string ops, frag alloc as bpf progs.

string manipulations are next.

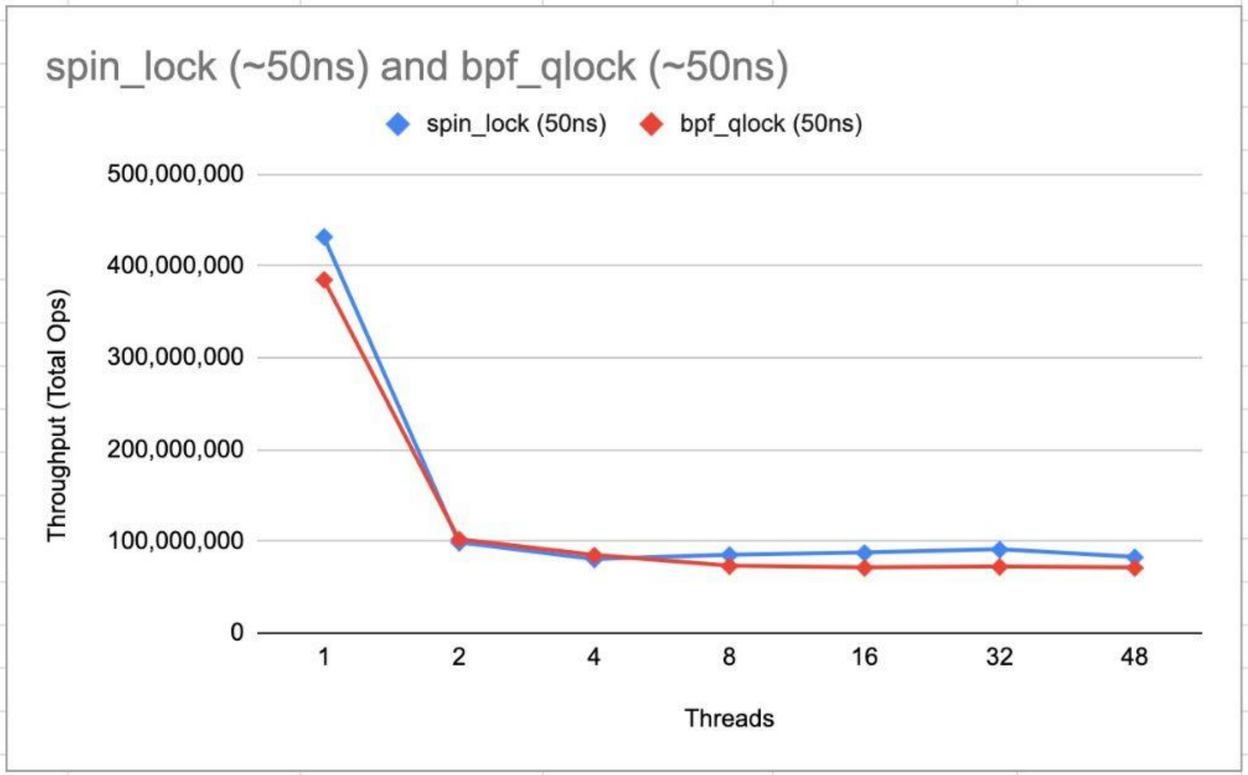
Need an infrastructure to share BPF code as libraries.
C/C++ didn't do it well. Dependency management is a pain.
Need to learn from rust/python.
Distribute libraries as source only.

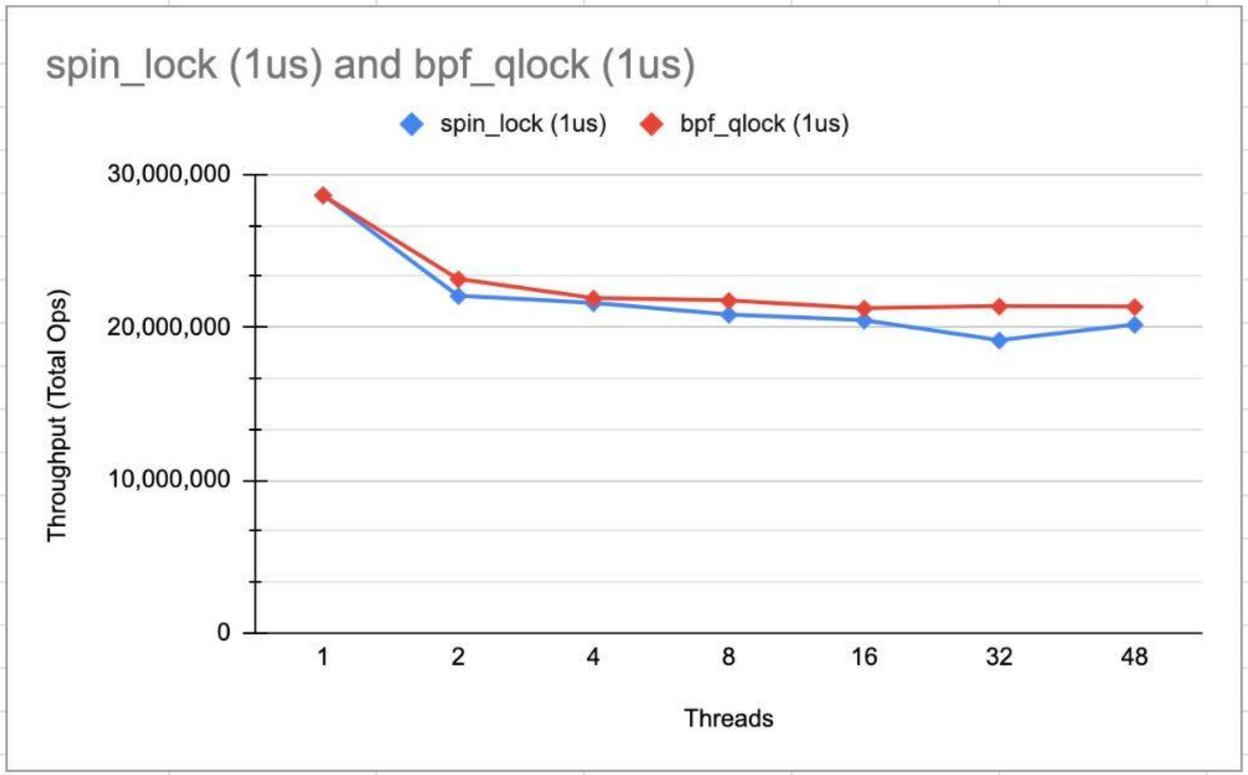
Towards arbitrary locks

Why: bpf_spin_lock() is severely restricted. One at a time. No calls while holding.
Bugs in the verifier and BPF infra can cause deadlocks.

Need: Another line of defence, so that locks taken by programs or core infra never affect the kernel.
Take multiple locks, detect deadlocks.
Call helpers and kfuncs while holding locks.
Locks need to be resilient to tampering.

Towards arbitrary locks





BPF is Turing complete

Implement BPF interpreter as BPF program:

```
switch (opcode) {
case BPF_ALU | BPF_ADD | BPF_K:
  ...
```

Expands to 100s of "if" branches.

Need support for jumptable and indirect goto instruction.

Evolving BPF ISA

Why: bpf_tail_call() is a workaround for a lack of indirect calls. Cumbersome to use.

- It's a tail call, not an indirect call.

Solution: Introduce indirect call instruction (likely opcode = BPF_JMP | BPF_CALL | BPF_X).

- Support in GCC and LLVM already exists.
- Call targets = global functions.
- The concept of "address of the instruction" is missing in the verifier.
- Necessary for static_branch support too.

Evolving BPF ISA

Why: Certain algorithms require efficient bit manipulation.

Solution: Introduce find first 0/1 bit, count bits, rotate

__attribute__((no_caller_saved_registers))

Why: Several hot-path helpers and kfuncs have no arguments and trivial implementation, but compilers are forced to save precious R0-R5 BPF registers across the call.

bpf_rcu_read_lock() is not free, while it's a nop in preempt=none kernels.

Solution: Introduce attribute((no_csr)) and corresponding support in the kernel and libbpf

r0 (ret), r1-r5 (caller saved), r6-r9 (callee), r10

Why: BPF registers match x86 regs, but arm64, riscv, s390 have a bunch more.
  x86 vs BPF performance gap is tiny on x86, but bigger on other architectures.

1. Introduce virtual registers for compilers and register allocator in the kernel ?
2. Allow compilers to use r11, r12, ..., reject such bpf progs on x86, allow on arm64 ?
3. "Fat" BPF ELF files with code compiled for 10 registers and for 20 ?
4. The verifier tracks spill/fill of registers. Convert stack load/store to register copy ?

Pass 6+ arguments

Why: BPF registers r1-r5 are used to pass arguments. Functions with 6+ arguments are not supported.

1. Use r11 to pass 6th argument, r12 for 7th. Convert to stack access in JITs ?
2. Use BPF stack to pass 6th, 7th arguments ?

Key challenge: translating kernel (cpu native) calling convention to BPF and vice versa

Easy: prog calling another prog

Hard: prog calling kfunc

Compile the kernel to BPF ISA

Why: vmlinux is a black box. BPF ISA allows binary to be analyzed

Solution: Once BPF supports indirect jumps, indirect calls, and 6+ arguments
the large part of the kernel can be compiled to BPF ISA.
Support for passing and returning struct by value maybe necessary.

alloca()

Why: BPF program stack is limited to 512 bytes.

Solution:

Step 1: Divided stack (not to be confused with split or segmented stack)

Kernel stack is used only to call and return. Prog stack is in separate memory.

Step 2: Allow BPF progs dynamically grow such stack.

alloca() is much faster than bpf_mem_alloc() and can be guaranteed not to fail.

Cancellable programs

Why: Program runtime is statically limited. Not safe enough.

Solution: Timeouts, watchdog with cancellation of programs.

- Exceptions and bpf_throw.
- Fast execute.

Userspace observability

Why: Kernel is written in C and annotated with BTF which makes for great observability with BPF.
User space is mostly C++ with large dwarf, python, etc.

Solution:

- Fast uprobes.
- Add USDTs.
- Language specific stack walkers (C++, python, hack, java, ...)

1 million processed instructions limit

Why: Any verifier or compiler change can make BPF program hit 1M instruction limit.
Horrible user experience. Complicates kernel upgrades.

Workaround:
Submit your progs as selftests or plain ELF objects. BPF CI will run them through veristat.
If your prog needs 100k insn it's in a danger zone. Tweak it to use global functions
and good loops.

Partial solution:
If the verifier is making forward progress don't limit it to 1M.
If it's not making progress then change verification strategy (widening of scalars, ...)

BPF subsystem as a kernel module

Why: Long tail of kernel versions makes it challenging to develop BPF programs that work across all.

Solution: Make BPF subsystem upgradable independent of base line kernel.

CONFIG_BPF=m

rmmod; insmod - doable, but not practical, since BPF progs are loaded.

Live upgrade is challenging, since data structures, addresses of kfuncs change.

Checkpoint all progs, update BPF core, restore all progs.

Upgrade verifier only.

What's next for BPF in the kernel?

Tracing/observability and programmable networking are largely solved.

bpf-lsm and security is a big growth area.

Another growth area is a scheduler (both tasks and packets).

BPF's Why Statement

To innovate

To enable others to innovate

To challenge what's possible