Falco's Discovery of the Modern eBPF World

Andrea Terzolo
Research fellow @ Polytechnic of Turin
Core Maintainer @ Falco

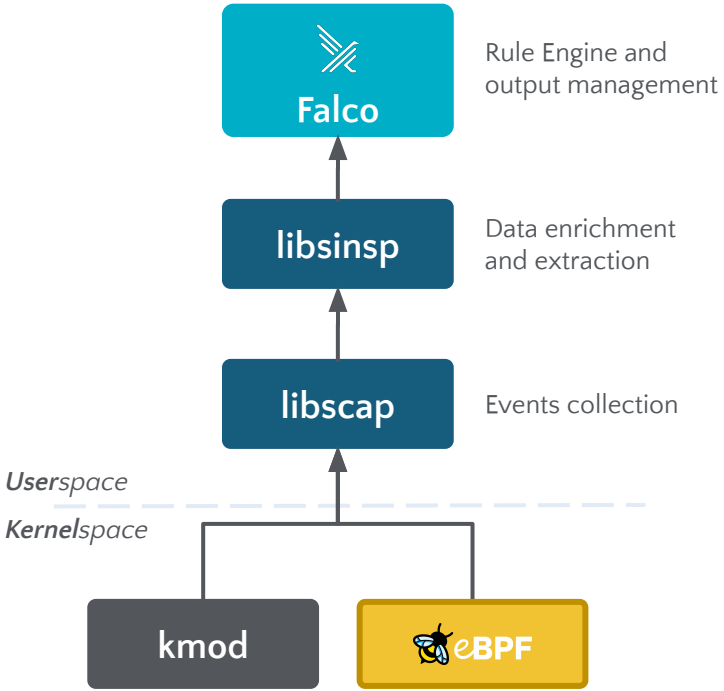
Jason Dellaluce
Open Source Engineer @ Sysdig
Core Maintainer @ Falco

eBPF Summit

@Andreagit97
@jasondellaluce

What is Falco?

- Falco collects events from your system and detects threats at runtime
- What about eBPF?
  - Safer alternative to our Kernel Module
    - Some envs don't allow a kmod at all
  - High system compatibility
    - Clang from 5.0 onwards
    - Kernels from 4.14 onwards
  - Access to a high number of events



What's Next?

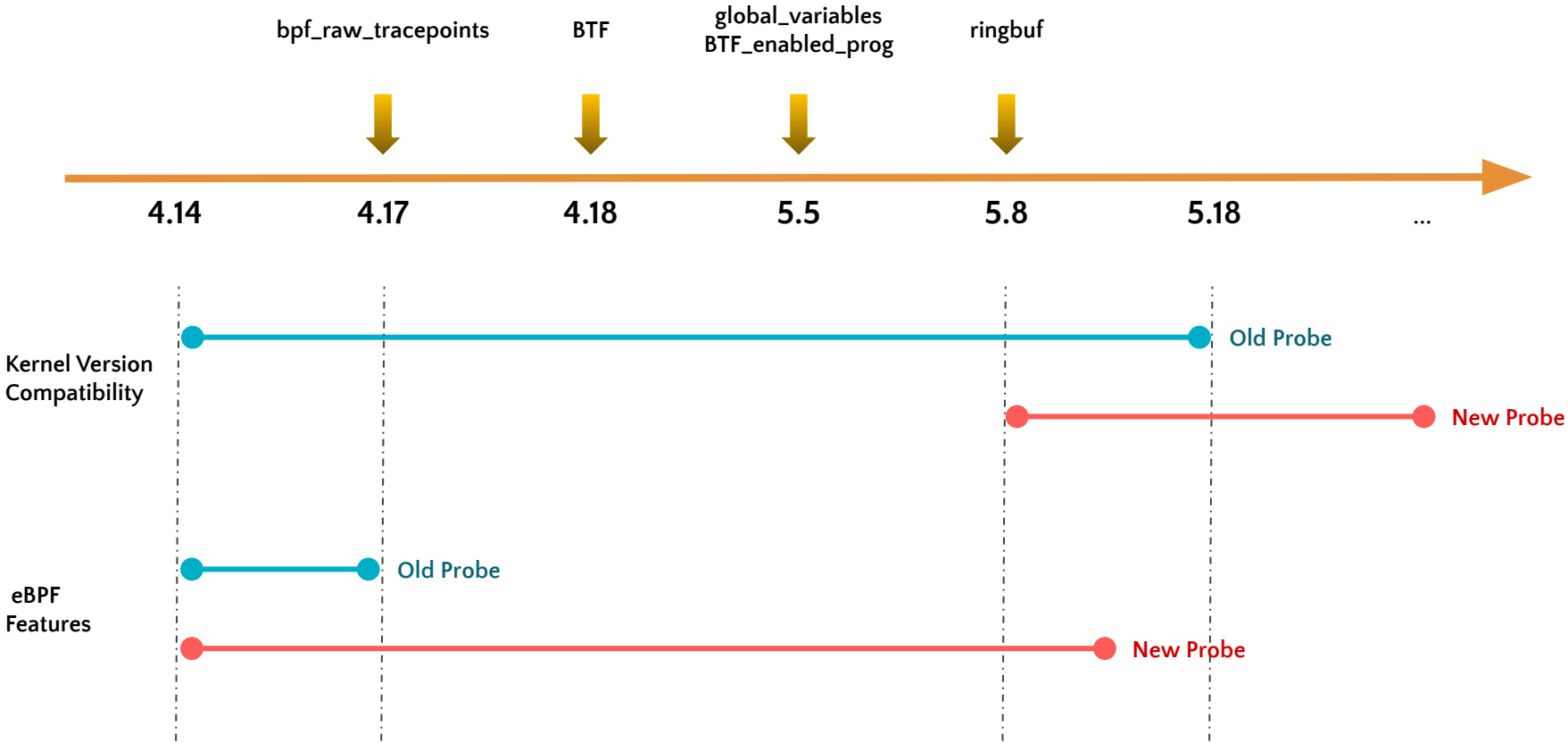
Pain Points

- **Portability** – compiling probe for many kernels
  - Pre-builts are costly
  - Kernel verifier still complains
- **Uniformity** – features availability changes in different kernels
  - We need many `#ifdef`
- **Size** – code grows and is harder to maintain

Solutions

- **CO-RE?**
  - Great for portability 😄
  - No good for size or uniformity ☹️
- **Use modern eBPF helpers?**
  - We'd lose older kernels support
- **Build another probe from scratch?**
  - Keep the old one for old kernels
  - Get the best of both worlds

Overview of our idea

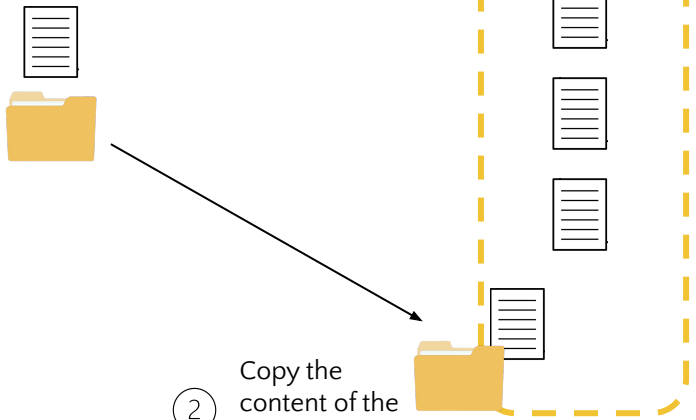


BPF ringbuf vs BPF perfbuf

PERF BUFFER

`bpf_perf_event_output()`

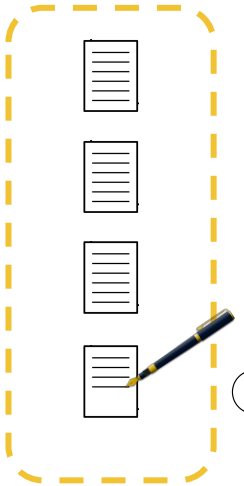
1. Collect our data into an auxiliary BPF map



2. Copy the content of the map into the perf buffer

RING BUFFER

`bpf_ringbuf_reserve()`



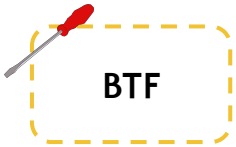
1. Write our data directly into the buffer

What If the size is not known at compile time?

Fallback

`bpf_ringbuf_output()`

BTF-enabled programs



BPF debug info

Verifier performs smart checks on the programs loaded.

To take advantage of these new features we need dedicated BPF programs

`BPF_PROG_TYPE_TRACING`

Direct memory access, **NO** more need of ~~`bpf_probe_read()`~~ helpers!

BPF_PROG_TYPE_RAW_TRACEPOINT

Simple `RAW_TRACEPOINT` program

```
SEC("raw_tp/sys_exit")
int catch_syscall_exit_event(struct sys_exit_args *ctx)
{
    struct task_struct *t = bpf_get_current_task();
    struct file * f;
    int err = BPF_CORE_READ_INTO(&f, t, mm, exe_file);
    if(err < 0)
    {
        return 1;
    }
    return 0;
}
```

BPF Bytecode

```
int catch_syscall_exit_event(struct sys_exit_args * ctx):
; struct task_struct *t = (struct task_struct *)bpf_get_current_task();
  0: (85) call bpf_get_current_task#-61936
  1: (b7) r1 = 2192
  2: (0f) r0 += r1
  3: (bf) r1 = r10
;
  4: (07) r1 += -16
; int err = BPF_CORE_READ_INTO(&f, t, mm, exe_file);
  5: (b7) r2 = 8
  6: (bf) r3 = r0
  7: (85) call bpf_probe_read_kernel#-64432
  8: (b7) r1 = 944
  9: (79) r3 = *(u64 *)(r10 -16)
 10: (0f) r3 += r1
 11: (bf) r1 = r10
;
 12: (07) r1 += -8
; int err = BPF_CORE_READ_INTO(&f, t, mm, exe_file);
 13: (b7) r2 = 8
 14: (85) call bpf_probe_read_kernel#-64432
 15: (18) r1 = 0x80000000
; int err = BPF_CORE_READ_INTO(&f, t, mm, exe_file);
 17: (5f) r0 &= r1
;
 18: (77) r0 >>= 31
; }
 19: (95) exit
```

BPF_PROG_TYPE_TRACING

Simple TRACING program

```
SEC("tp_btf/sys_exit")
int catch_syscall_exit_event(struct sys_exit_args *ctx)
{
    struct task_struct *t = bpf_get_current_task_btf();
    struct file * f = t->mm->exe_file;
    if(!f)
    {
        return 1;
    }
    return 0;
}
```

BPF Bytecode

```
int catch_syscall_exit_event(struct sys_exit_args * ctx):
; struct task_struct *t = bpf_get_current_task_btf();
   0: (85) call bpf_get_current_task_btf#-61936
; struct file * f = t->mm->exe_file;
   1: (79) r1 = *(u64 *)(r0 +2192)
; struct file * f = t->mm->exe_file;
   2: (79) r1 = *(u64 *)(r1 +944)
   3: (b7) r0 = 1
; if(!f)
   4: (15) if r1 == 0x0 goto pc+1
   5: (b7) r0 = 0
; }
   6: (95) exit
```

IMPROVEMENTS

- Better performance, reduced bytecode
- Code readability and maintainability, C-like code

BPF GLOBAL VARIABLES

Array with just one key/value pair that contains all global data.


```
{
   "key": ["0x00","0x00","0x00","0x00"
   ],
   "value": ["0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00",
             "0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00","0x00"],
   "formatted": {
       "value": {
           ".bss": [{
                   "glob_val_1": 0
               },{
                   "glob_val_2": 0
               },{
                   "glob_val_3": 0
               },{
                   "glob_val_4": 0
               },{
                   "glob_val_5": 0
               },{
                   "glob_val_6": 0
               }
           ]
       }
   }
}
```

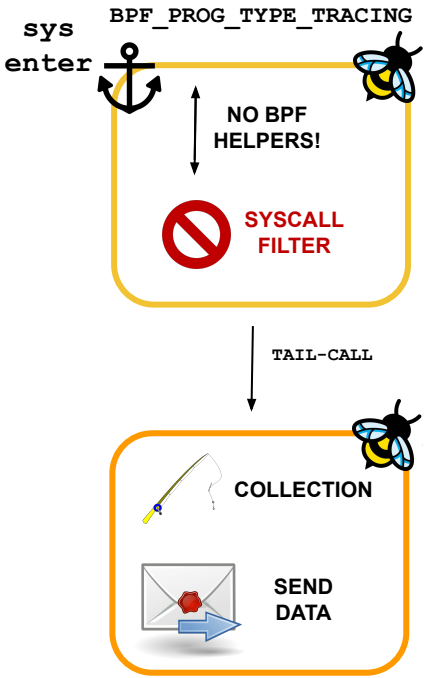

IMPROVEMENTS

- BPF programs can access global variables without BPF helpers.
- Userspace can access global variables without the `bpf` syscall.

Architecture comparison

sys_enter capture flow

MODERN ARCHITECTURE



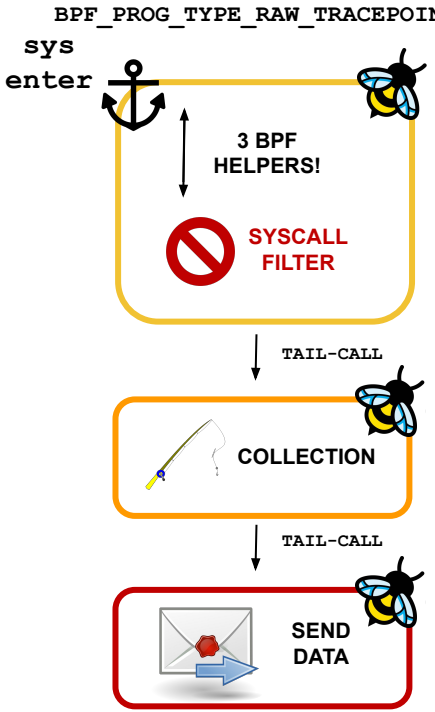
Global variables + BTF-enabled progs allow us to filter syscalls without calling helpers

RING BUFFER

reserve approach with fixed-size events.

output approach with variable-size events

OLD ARCHITECTURE



The BTF absence force us to use 3 bpf helpers.

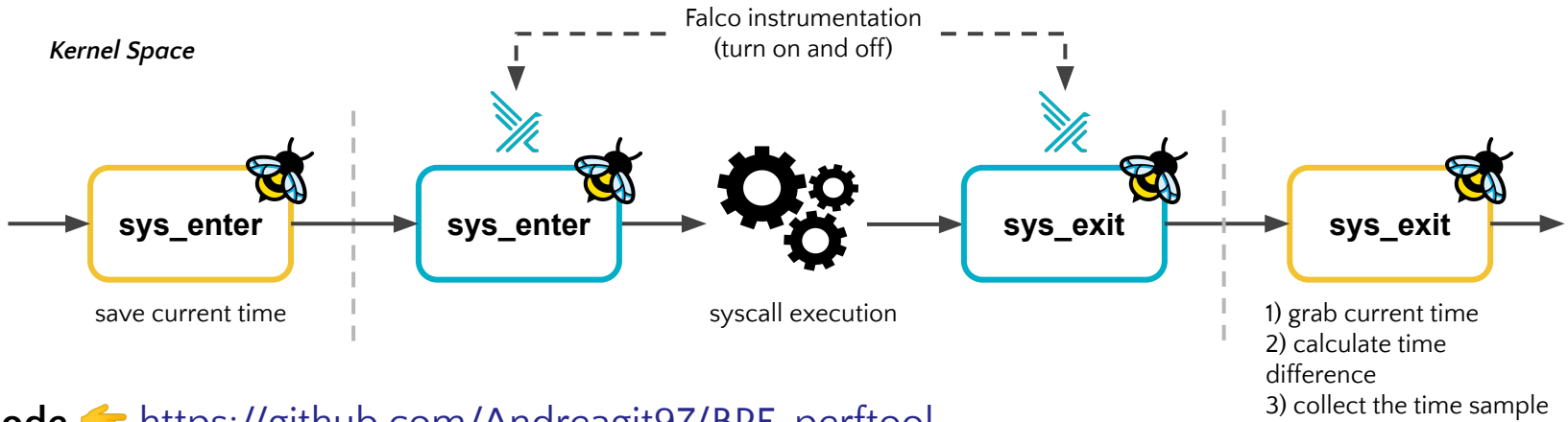
We need a dedicated BPF program to collect data

PERF BUFFER

We cannot reserve space, we need to use an extra memory copy every-time

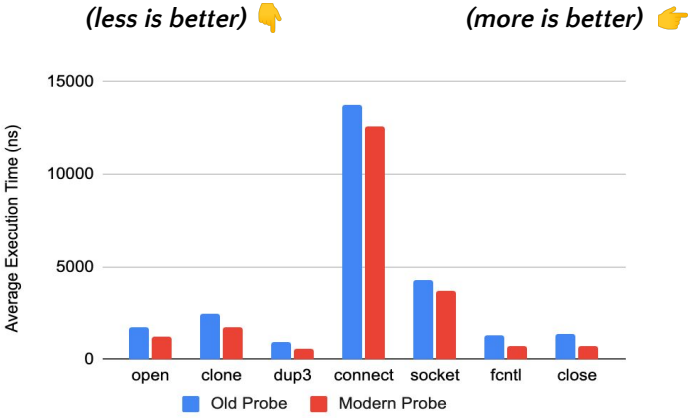
Performance: methodology

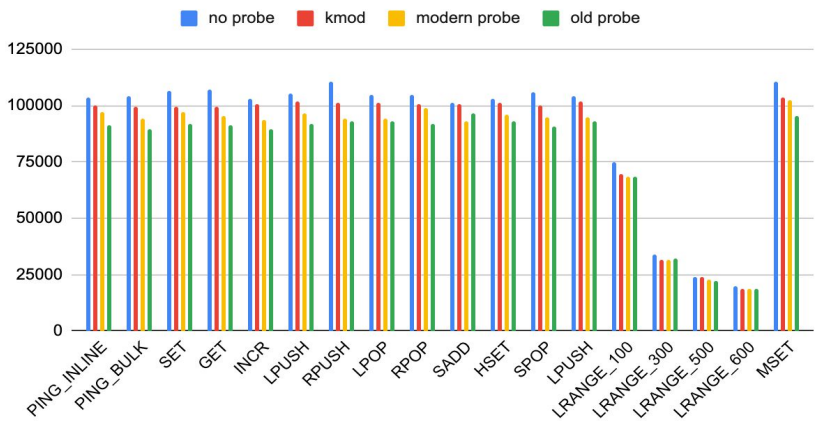
- **Userspace level –** run a benchmark (*redis-benchmark*) **with** and **without** kernel instrumentation
- **Kernel level –** attach programs **before** and **after** syscall execution and then compute the average time
  - **With** and **without** Falco instrumentation, for both **old** and **modern** eBPF probes



Code 👉 https://github.com/Andreagit97/BPF-perftool

Performance: observations

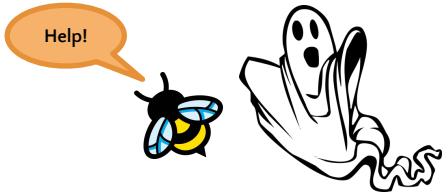




- Modern probe has ~**40% less syscall execution overhead** than old probe
- **Benchmark performance ~3% faster** with modern probe than with old probe
  - ~5% slowdown with modern probe, ~8% with old probe (confirms ~40% speedup)
- Modern probe over syscall execution **overhead still ~20% higher than kmod**
  - Just a **~1.2% benchmark performance difference** (we're getting close! 😄)

Developer experience

- Don't be scared by the old BPF development!
  - Smarter Verifier
  - Modern BPF features seen in this presentation



- Test it always! Even in BPF!
  - `libbpf` + `googletest`
  - Our testing framework 👇
    https://github.com/falcosecurity/libs/tree/master/test/modern_bpf

- Writing BPF programs is not like writing C programs!
  - Keep the flow linear
  - Avoid castles of inline methods

Thank you!

andrea.terzolo@polito.it

jasondellaluce@gmail.com