# How and When You Should Measure CPU Overhead of eBPF Programs

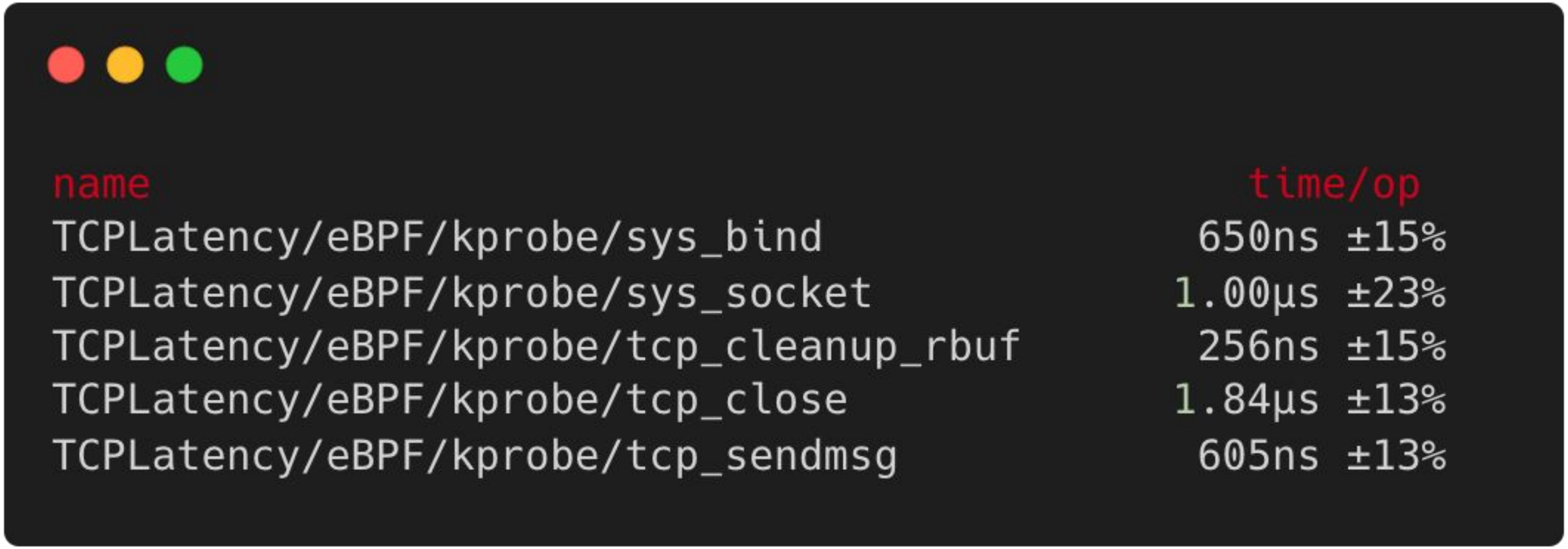
eBPF Summit

Bryce Kahle, Datadog

October 28, 2020

# Why should I profile eBPF programs?

# CI variance tracking



```
name                                               time/op
TCPLatency/eBPF/kprobe/sys_bind                    650ns ±15%
TCPLatency/eBPF/kprobe/sys_socket                 1.00µs ±23%
TCPLatency/eBPF/kprobe/tcp_cleanup_rbuf            256ns ±15%
TCPLatency/eBPF/kprobe/tcp_close                  1.84µs ±13%
TCPLatency/eBPF/kprobe/tcp_sendmsg                 605ns ±13%
```

# Tools

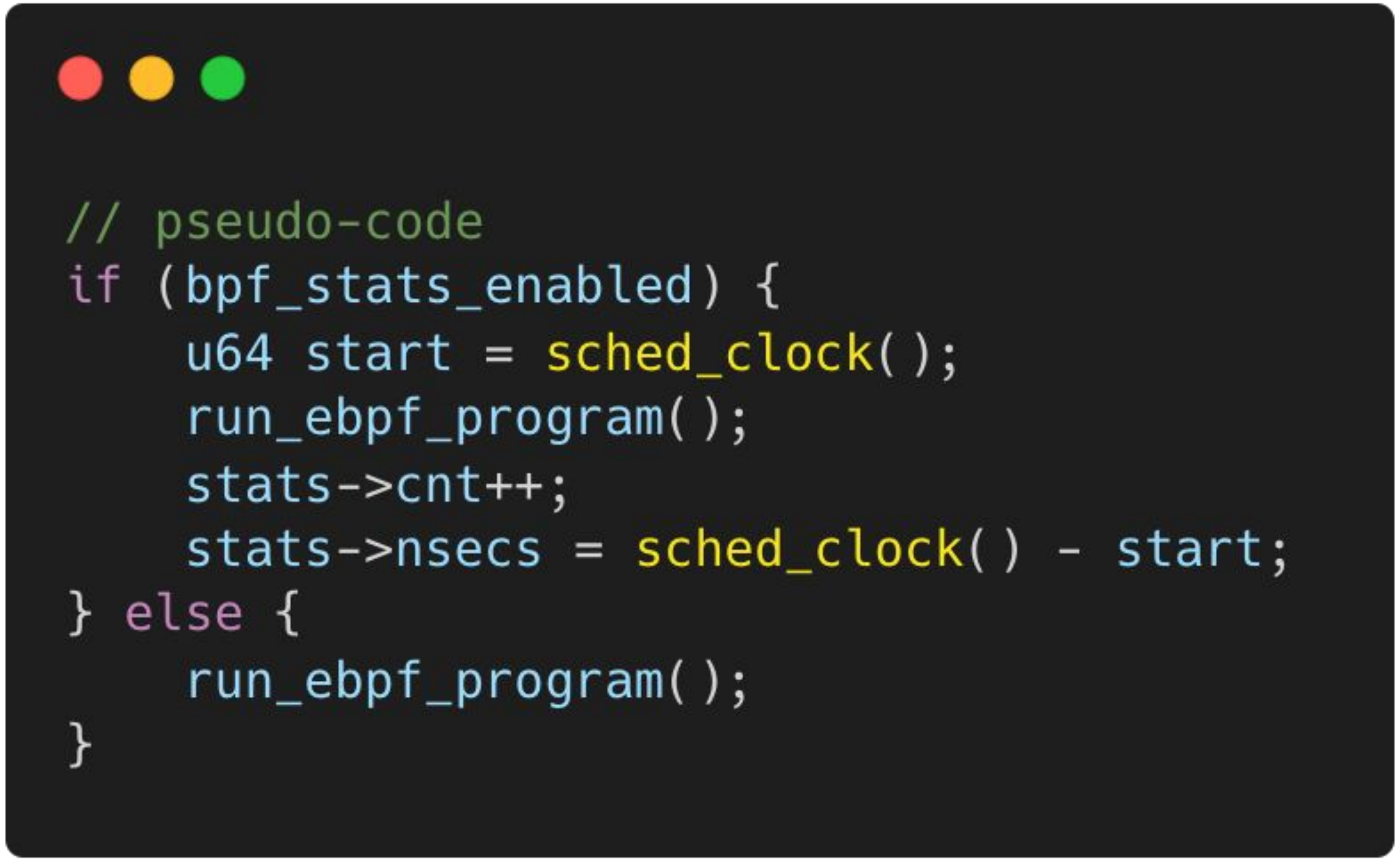
# kernel.bpf_stats_enabled

# kernel.bpf_stats_enabled sysctl

- Added in kernel v5.1 (off by default)
- Turns on stats collection for **all eBPF programs**
- exposes total `run_time_ns` and `run_cnt`
- Use cases:
  - Benchmarking + CI/CD
  - Sampling profiler in production

# How does it work?

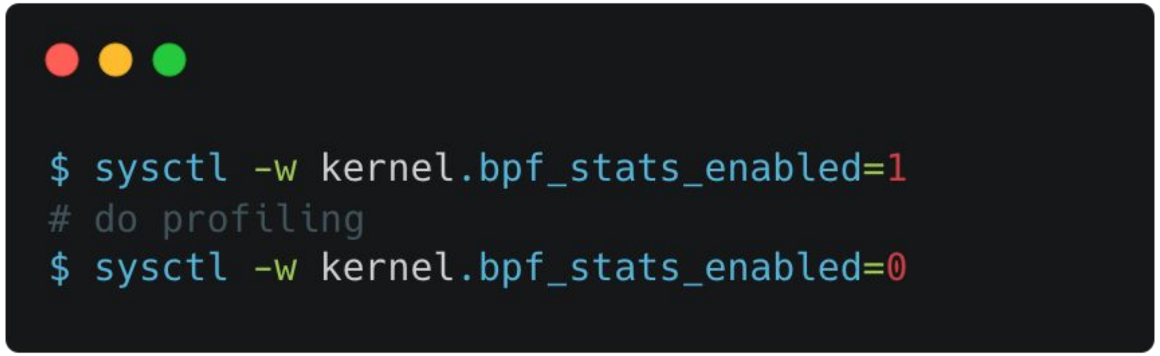
- Adds ~20ns of overhead per run



```
// pseudo-code
if (bpf_stats_enabled) {
    u64 start = sched_clock();
    run_ebpf_program();
    stats->cnt++;
    stats->nsecs = sched_clock() - start;
} else {
    run_ebpf_program();
}
```

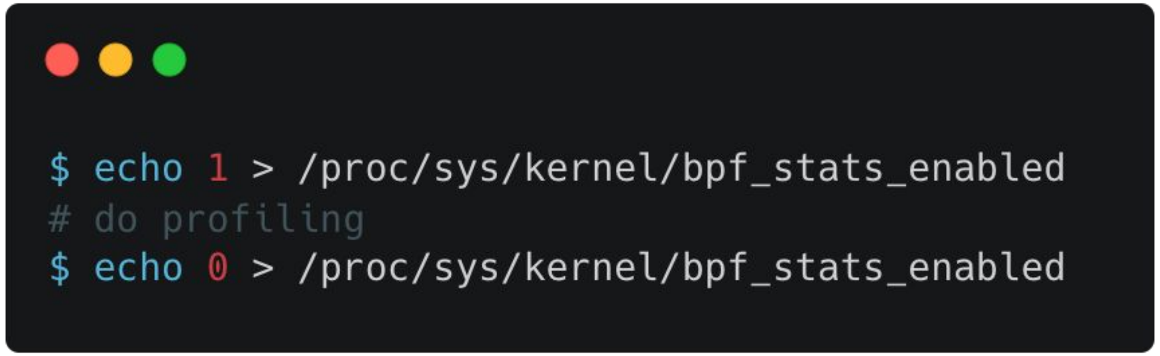
# Two ways to enable kernel eBPF stats

## sysctl



```
$ sysctl -w kernel.bpf_stats_enabled=1
# do profiling
$ sysctl -w kernel.bpf_stats_enabled=0
```

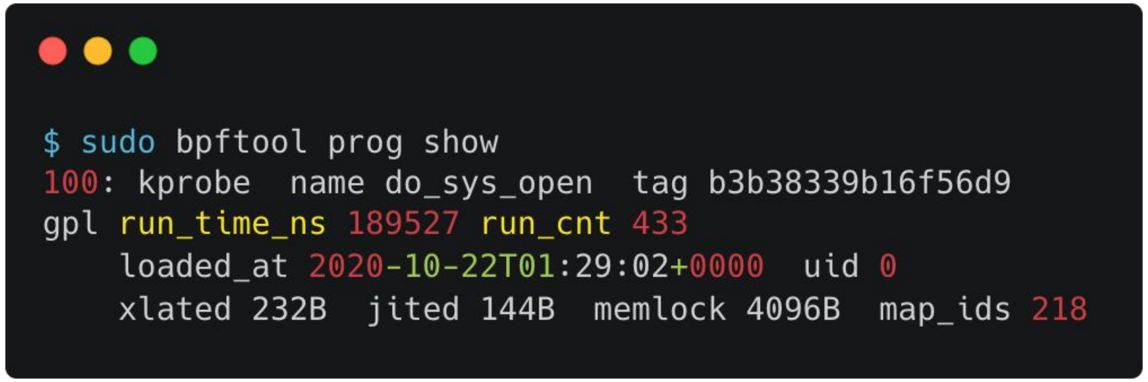
## procfs



```
$ echo 1 > /proc/sys/kernel/bpf_stats_enabled
# do profiling
$ echo 0 > /proc/sys/kernel/bpf_stats_enabled
```

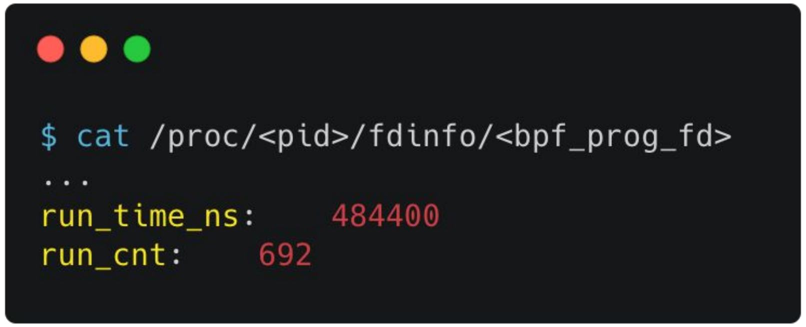
# Three ways to access kernel eBPF stats

## bpftool prog show



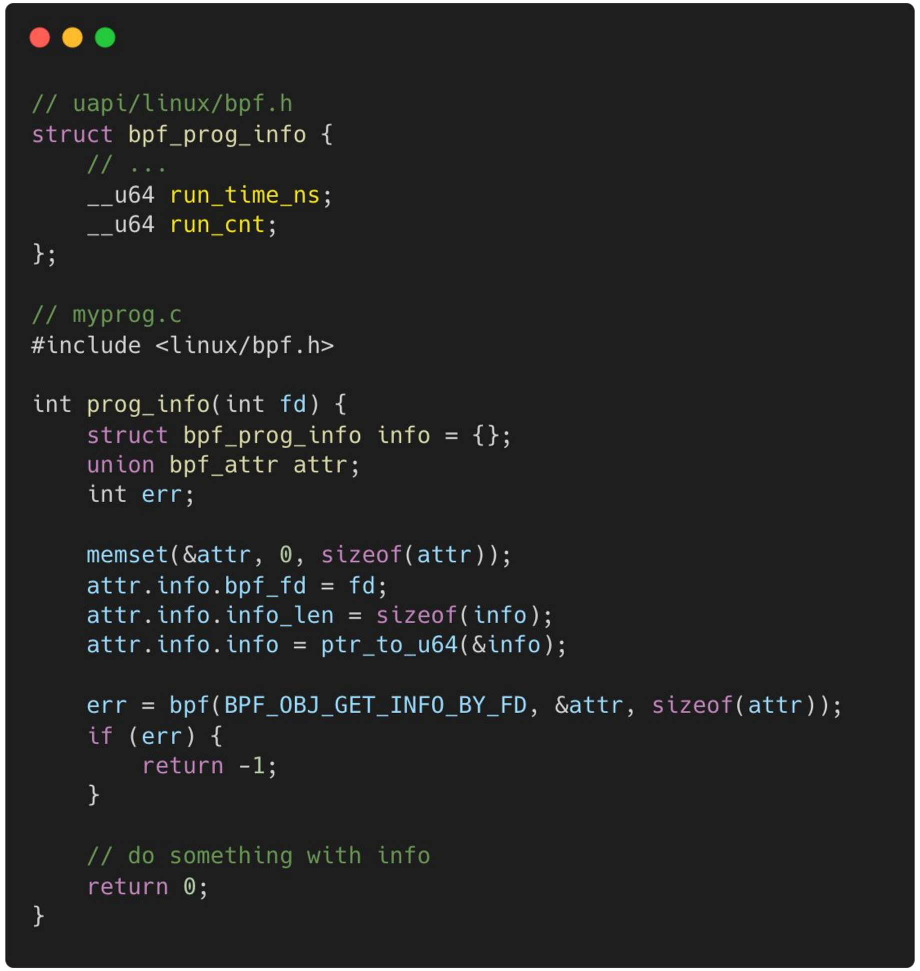
```
$ sudo bpftool prog show
100: kprobe  name do_sys_open  tag b3b38339b16f56d9
gpl run_time_ns 189527 run_cnt 433
    loaded_at 2020-10-22T01:29:02+0000  uid 0
    xlated 232B  jited 144B  memlock 4096B  map_ids 218
```

## procfs



```
$ cat /proc/<pid>/fdinfo/<bpf_prog_fd>
...
run_time_ns:    484400
run_cnt:    692
```

## bpf syscall BPF_OBJ_GET_INFO_BY_FD



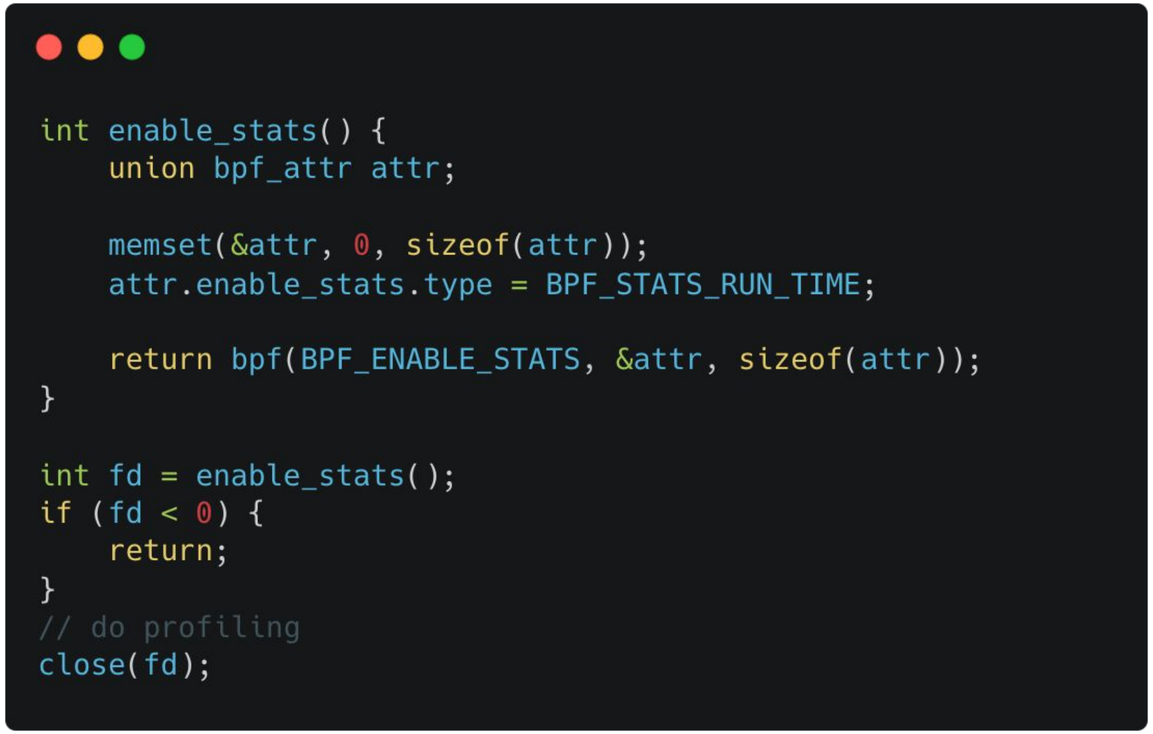
```
// uapi/linux/bpf.h
struct bpf_prog_info {
    // ...
    __u64 run_time_ns;
    __u64 run_cnt;
};

// myprog.c
#include <linux/bpf.h>

int prog_info(int fd) {
    struct bpf_prog_info info = {};
    union bpf_attr attr;
    int err;

    memset(&attr, 0, sizeof(attr));
    attr.info.bpf_fd = fd;
    attr.info.info_len = sizeof(info);
    attr.info.info = ptr_to_u64(&info);

    err = bpf(BPF_OBJ_GET_INFO_BY_FD, &attr, sizeof(attr));
    if (err) {
        return -1;
    }

    // do something with info
    return 0;
}
```

# BPF_ENABLE_STATS

Added in kernel v5.8

FD-based alternative to sysctl

Handles multiple concurrent profilers



```c
int enable_stats() {
    union bpf_attr attr;

    memset(&attr, 0, sizeof(attr));
    attr.enable_stats.type = BPF_STATS_RUN_TIME;

    return bpf(BPF_ENABLE_STATS, &attr, sizeof(attr));
}

int fd = enable_stats();
if (fd < 0) {
    return;
}
// do profiling
close(fd);
```

# bpftool prog profile

# bpftool prog profile

- Added in kernel v5.7
- Uses hardware perf counters
- Available metrics:
  - `cycles, instructions, l1d_loads, llc_misses`
- Used for more in-depth profiling

# bpftool prog run / BPF_PROG_TEST_RUN

# bpftool prog run

- Added in kernel v4.12
- Only for specific program types
- Specify how many times to repeat
- Control input data and/or context. Examine output data/context.
- Use cases:
  - Unit testing
  - Debugging

# bpftool prog run

## Feature Support

| Program Type | Input Data | Input Context | Output Data | Output Context | Repeat |
|---|---|---|---|---|---|
| BPF_PROG_TYPE_SOCKET_FILTER | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_SCHED_CLS | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_SCHED_ACT | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_CGROUP_SKB | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_LWT_IN | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_LWT_OUT | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_LWT_XMIT | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_LWT_SEG6LOCAL | ✅ | ✅ | ✅ | ✅ | ✅ |
| BPF_PROG_TYPE_XDP | ✅ | 🚫 | ✅ | 🚫 | ✅ |
| BPF_PROG_TYPE_FLOW_DISSECTOR | ✅ | ✅ | ✅ | ✅ | ✅ |

DATADOG

# ebpfbench - Go library for eBPF benchmarking

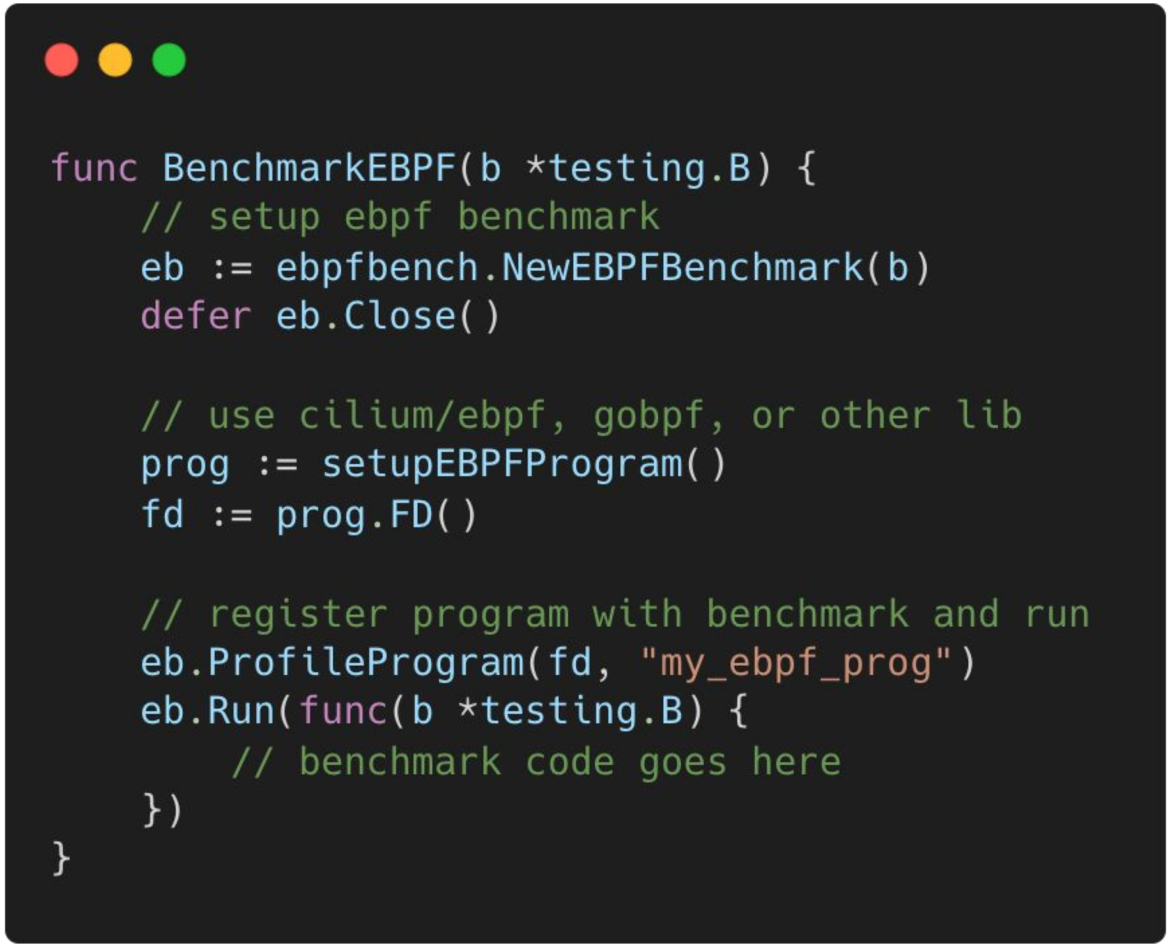
https://github.com/DataDog/ebpfbench

# ebpfbench

API Augments `testing.B`

Outputs results in go benchmark format

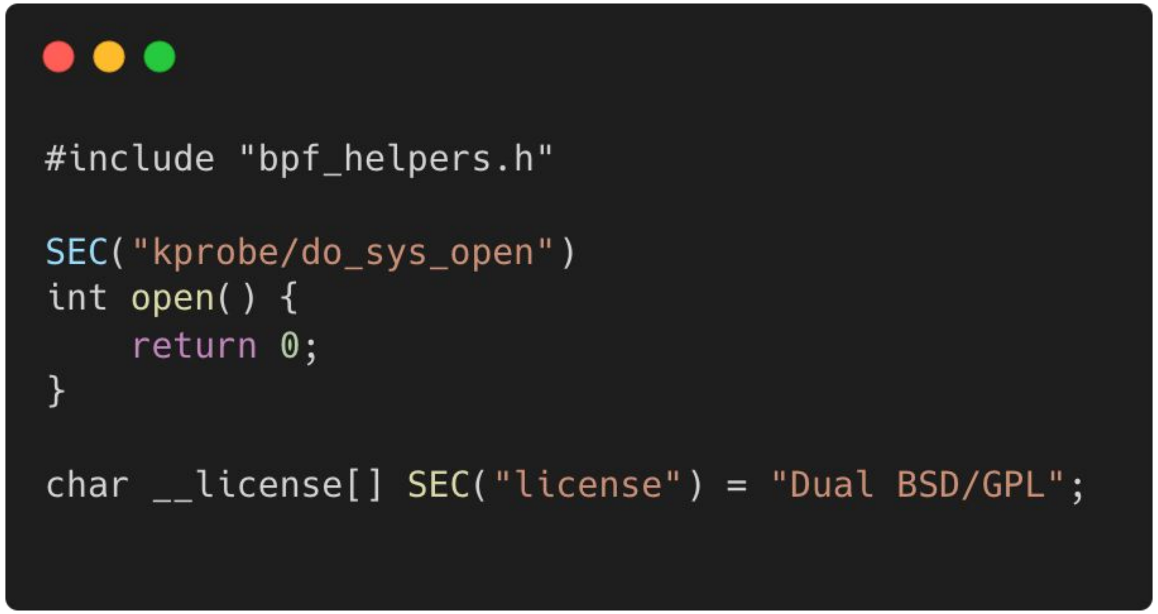
Can be used with `benchstat` and other tools



```go
func BenchmarkEBPF(b *testing.B) {
    // setup ebpf benchmark
    eb := ebpfbench.NewEBPFBenchmark(b)
    defer eb.Close()

    // use cilium/ebpf, gobpf, or other lib
    prog := setupEBPFProgram()
    fd := prog.FD()

    // register program with benchmark and run
    eb.ProfileProgram(fd, "my_ebpf_prog")
    eb.Run(func(b *testing.B) {
        // benchmark code goes here
    })
}
```

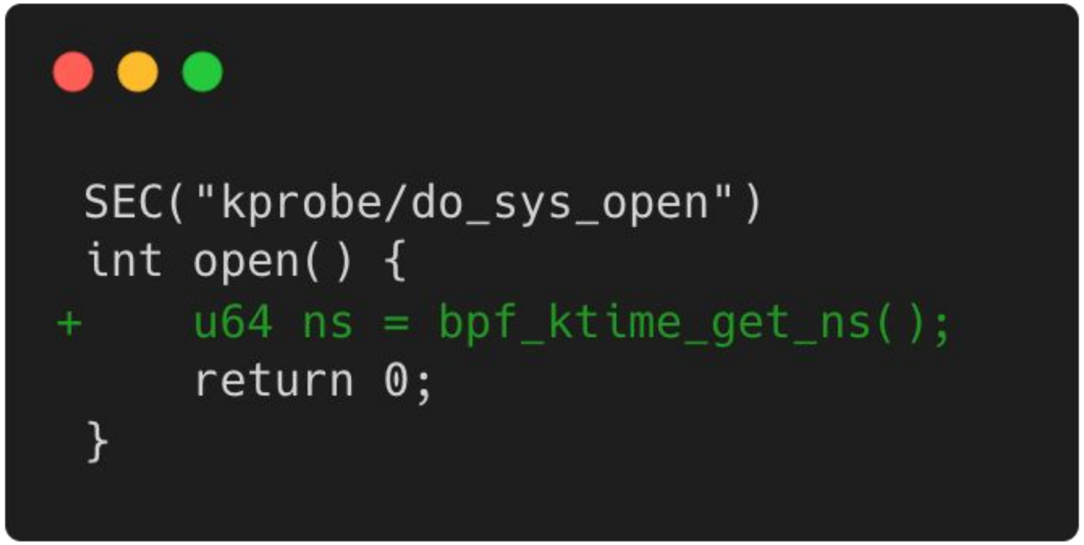
# kprobe runtime comparison programs

## no-helper



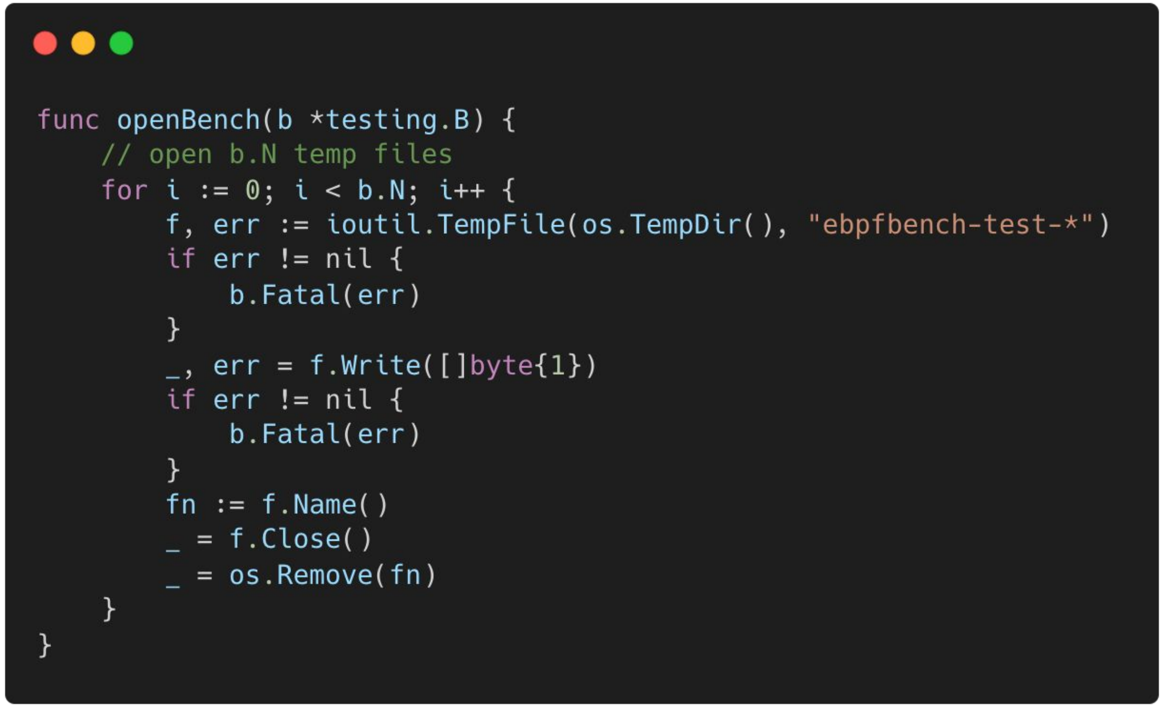
```c
#include "bpf_helpers.h"

SEC("kprobe/do_sys_open")
int open() {
    return 0;
}

char __license[] SEC("license") = "Dual BSD/GPL";
```

## add helper call

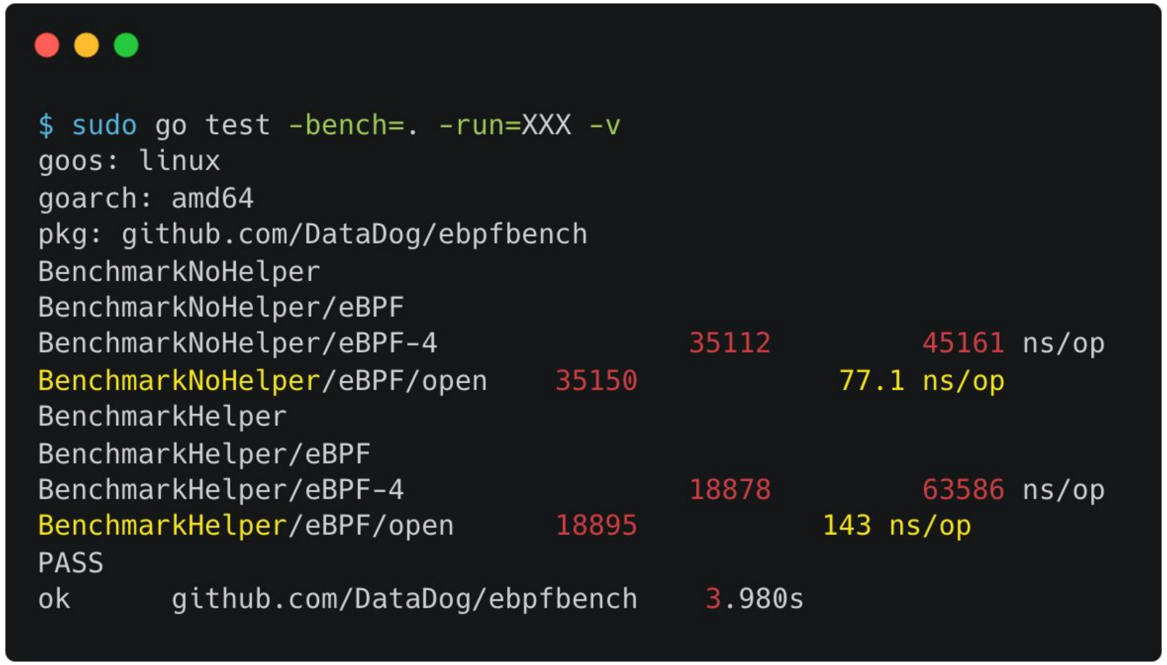


```diff
 SEC("kprobe/do_sys_open")
 int open() {
+    u64 ns = bpf_ktime_get_ns();
     return 0;
 }
```

# benchmark program and results



```go
func openBench(b *testing.B) {
    // open b.N temp files
    for i := 0; i < b.N; i++ {
        f, err := ioutil.TempFile(os.TempDir(), "ebpfbench-test-*")
        if err != nil {
            b.Fatal(err)
        }
        _, err = f.Write([]byte{1})
        if err != nil {
            b.Fatal(err)
        }
        fn := f.Name()
        _ = f.Close()
        _ = os.Remove(fn)
    }
}
```



```
$ sudo go test -bench=. -run=XXX -v
goos: linux
goarch: amd64
pkg: github.com/DataDog/ebpfbench
BenchmarkNoHelper
BenchmarkNoHelper/eBPF
BenchmarkNoHelper/eBPF-4               35112             45161 ns/op
BenchmarkNoHelper/eBPF/open    35150            77.1 ns/op
BenchmarkHelper
BenchmarkHelper/eBPF
BenchmarkHelper/eBPF-4                 18878             63586 ns/op
BenchmarkHelper/eBPF/open      18895           143 ns/op
PASS
ok      github.com/DataDog/ebpfbench    3.980s
```

# Thank you!

DATADOG