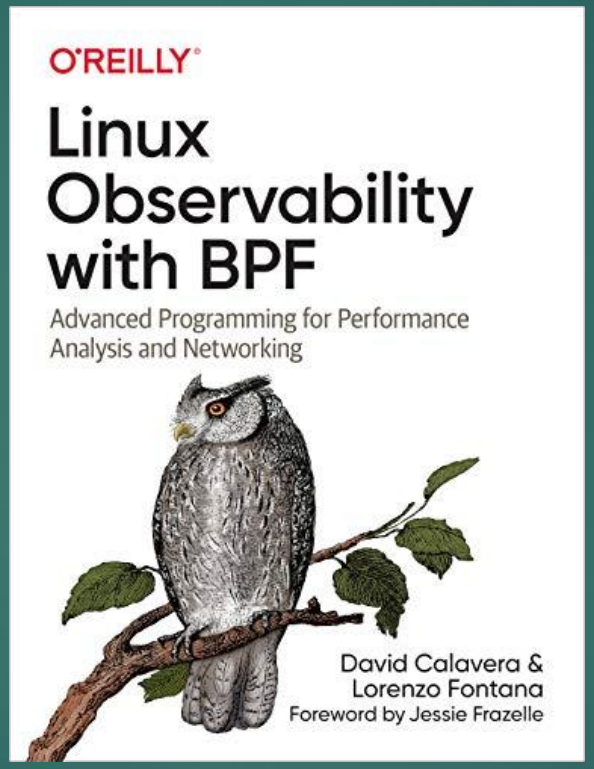
a vm wtf eBPF


O'REILLY
Linux Observability with BPF
Advanced Programming for Performance Analysis and Networking
David Calavera &
Lorenzo Fontana
Foreword by Jessie Frazelle

What is observability?

- Understanding what is happening?
  Everything is so much more complicated these days
- On Windows, there's ETW
  … but that feels like you're holding on to a fire hose
- Log your way to success? Flight recording (undo.io)?
- Where do you do the filtering?
- And you don't want to restart
- And what if you could modify the kernel's normal behaviour

Why am I interested?

- Understanding what a Kubernetes cluster was doing
  Brendan Gregg - https://github.com/brendangregg

- Cilium - https://github.com/cilium/cilium
  iptables not designed for the load

Installing the tooling

- sudo apt update
- sudo apt install build-essential git make libelf-dev clang strace tar bpfcc-tools linux-headers-$(uname -r) gcc-multilib

- cd /tmp
- git clone --depth 1 git://kernel.ubuntu.com/ubuntu/ubuntu-bionic.git
- sudo mv ubuntu-bionic /kernel-src
- cd /kernel-src/tools/lib/bpf
- sudo make && sudo make install prefix=/usr/local
- sudo mv /usr/local/lib64/libbpf.* /lib/x86_64-linux-gnu/

A simple first example

- https://github.com/bpftools/linux-observability-with-bpf

- cd linux-observability-with-bpf/code/chapter-2/hello_world
- make

- sudo ./monitor-exec

So what are the parts

- Byte code
- A byte code verifier
- A way to communicate between the kernel and user space
- A set of places to attach the byte code

- Then BCC
- Then Bpftool and Bpftrace and kubectl-trace
- And then flamegraphs

bpftool

- git clone https://github.com/torvalds/linux
- cd linux
- git checkout v5.1
- cd tools/bpf/bpftool/
- make && sudo make install

Part 1: Return of the bytecode

- sudo bpftool prog show
- sudo bpftool perf

- sudo bpftool prog dump xlated id 62

- https://github.com/iovisor/bpf-docs/blob/master/eBPF.md
- https://www.kernel.org/doc/Documentation/networking/filter.txt

Part 2: Byte code verifier

- Protect the kernel at all costs
- No loops
- Maximum number of instructions
- Extended over time to tail call other scripts

Part 3: Communication

- A rich set of maps

  hash/array/stack maps (global and per-cpu)

- And a virtual file system to persist them

  A bpf file system you can mount

- Maps can be initialized from user space

Part 4: A set of trigger points

- KProbes
- Tracepoints

  sudo ls -l /sys/kernel/debug/tracing/events
- User probes

  Hook user functions
- … networking….

  Lots of places to change the behaviour
  (DOS attacks and load balancing)

BCC

- Tooling around using eBPF

- Eg python wrappers
  cd linux-observability-with-bpf/code/chapter-4/uprobes

  cat example.py

Bpftool

- Let's you see what's happening

Bpftrace

- See this blog post https://theartofmachinery.com/2019/04/26/bpftrace_d_gc.html

- Hook user functions with a higher level DSL for expressing intent
- Print out maps in user friendly form (like histograms)

- https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md

BPF Heritage

- Fast networking filters

  1992 The BSD Packet Filter: A new architecture for User-level packet capture

- 2014 The extended BPF implementation

  A better instruction set

What, no packet filtering?

- Take a packet and modify and drop/continue/resend after modification
- Push the eBPF onto the card itself

#linux-insides

- Panagis posted the original paper here

Bpf is being used from boot

- sudo bpftool prog show
- sudo bpftool prog show --json | jq -c '.[] | [.id, .type, .loaded_at]'
- sudo bpftool prog dump xlated id 36

- sudo bpftool cgroup tree
- Sudo bpftool net

- sudo bpftool map show
- sudo bpftool map dump id 41