

Rust <3 eBPF

Linux system monitoring with Rust

---

# Welcome to Operating Systems 101

Course material: <https://github.com/redsift/ingraind-rf-bcn>

What we'll cover:

- 1 | Theory
- 2 | Diagrams
- 4 | Network stack implementations
- 5 | System calls, kernel, and userland
- 6 | Deep dive into Linux internals

We'll use Rust and eBPF for an interactive experience.

# Let's get started

Deets over here: <https://github.com/redsift/ingraind-rf-bcn>

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

1 Raspberry Pi

1 Ethernet cable

NOT IN  
BETA

||

1 VirtualBox

1 [vagrantup.com](https://vagrantup.com)

1 Vagrantfile

||

1 Linux 4.19+

1 LLVM 9

1 Kernel source

1 Editor  
1 Rust toolchain  
30 kg patience

---

# How do we observe computers?

## Basic monitoring tools

Operating systems move away from batch processing and into the interactive real-time world

**1990s**



## Functional performance metrics

Clunky mysterious black boxes are plugged into your infrastructure for device-centric alerts

**2000s**



## Network monitoring

Automated security information and event management monitor code performance, even at the infrastructure level

**2010s**



## Full observability

Gather operational metrics straight from the kernel, Docker, or other management systems. Apps can expose more metrics that are specific to them.

**2020s**

---

# Why we do monitoring

We need a way to tell

- Which code paths are running, so we **log**
- What's eating CPU and/or RAM, so we look at **performance metrics**
- Who's accessing the system, and what they're doing, so we collect **security metrics**
- How reliable/resilient our setup is, so we monitor **traffic shape**
- **Observe the state our system's in**

## eBPF

A virtual machine inside the Linux kernel.

Maximum 4096 instructions/program

Not Turing-complete

Stateless (only in maps)

Google, Facebook, Cloudflare

## Rust

A programming language (dah)

Kinda why we're here

Mozilla, Microsoft, Facebook, Cloudflare

Aims to be "safe"

Used for systems/network programming

Used for security-related work

People rewrite everything in them

Red Sift

---

# Instructions

```
if (!(eth_proto == ETH_P_IP && ip_proto == IPPROTO_TCP))  
{  
    goto IGNORE;  
}
```

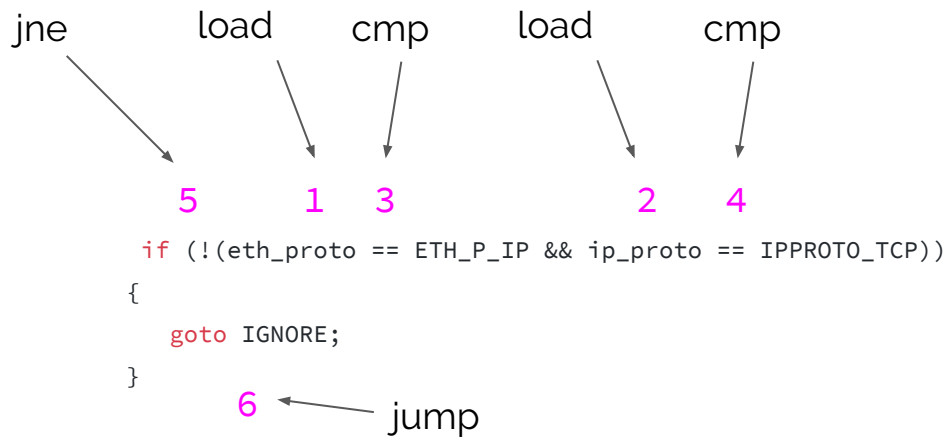
---

# Instructions

```
5      1 3      2 4
if (!(eth_proto == ETH_P_IP && ip_proto == IPPROTO_TCP))
{
    goto IGNORE;
}
6
```



# Instructions



Linux <5.4 ==> 4096

Linux >=5.4 ==> 1 000 000

---

# Eliminate the state

**Computers** are stateful, no matter how hard we try.

Docker **containers** can be stateless, but mostly aren't.

Your **application code** can run stateless, if you use some database as transient memory.

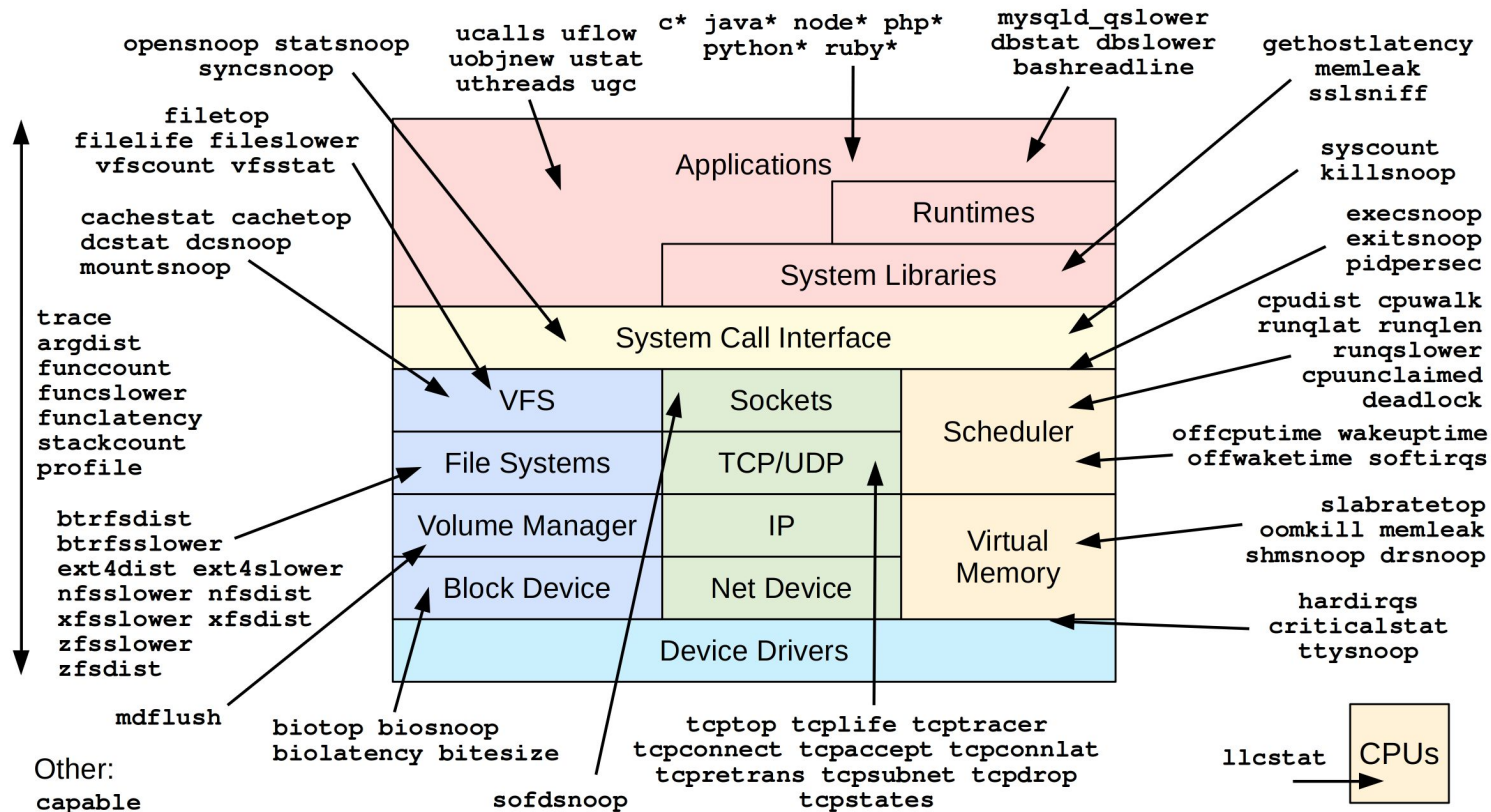
**Kubernetes** *StatefulSet* vs *Deployment* is about long-term state.



Distributed systems want to **decouple** application logic and state.

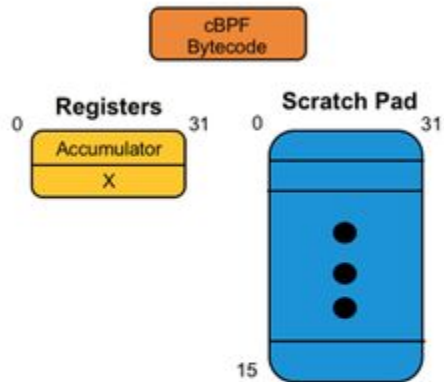
Consensus by blockchain/Paxos.

# Linux bcc/BPF Tracing Tools

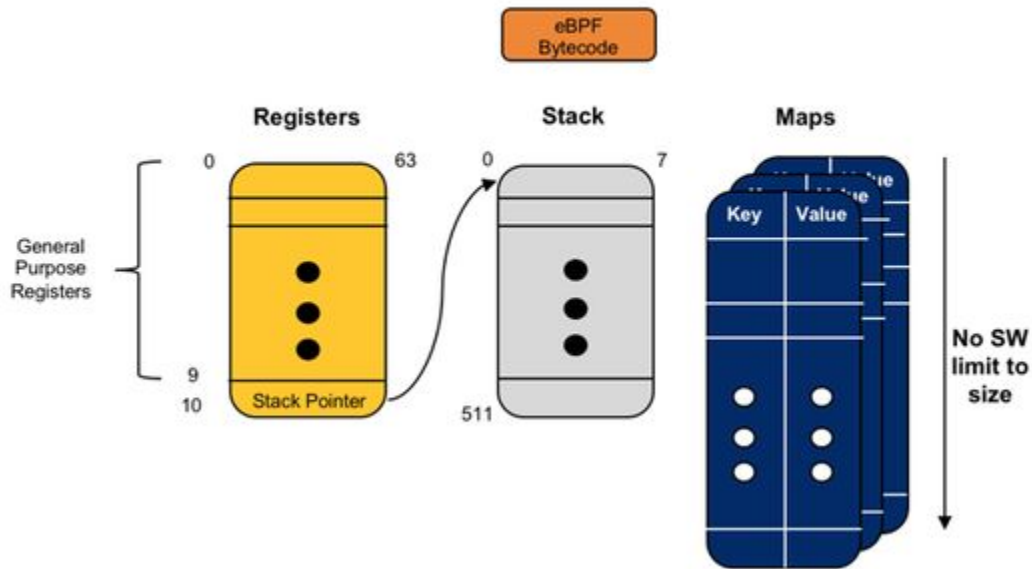


<https://github.com/iovisor/bcc#tools> 2019

## Classical BPF Machine

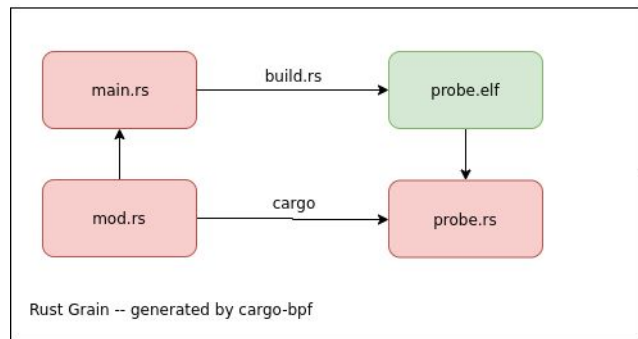
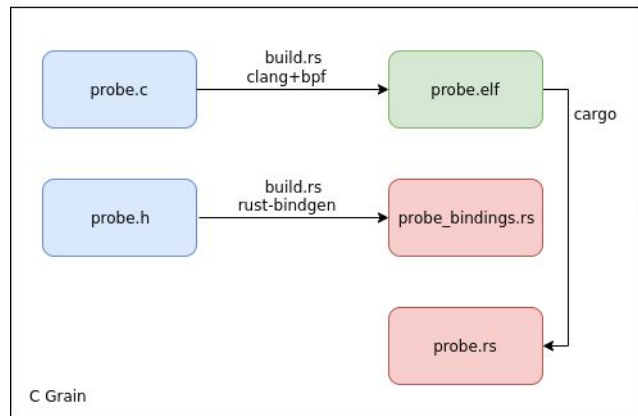


## Extended BPF Machine

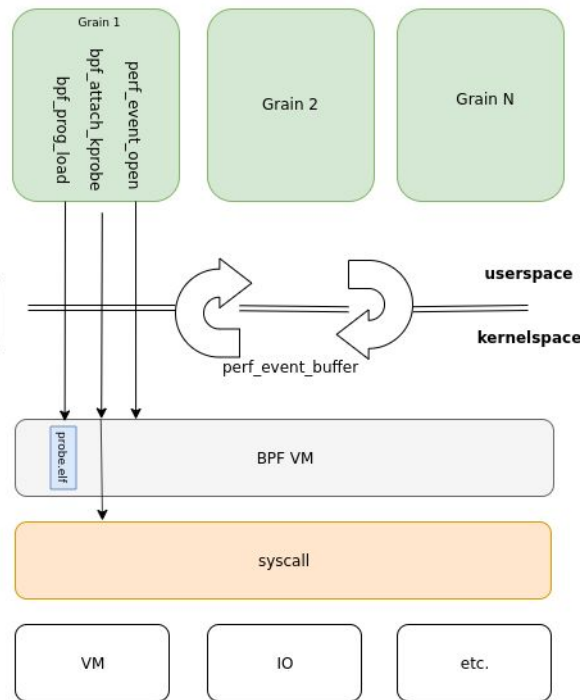


Source: <https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/>

## Build



## Runtime



---

# Why Rust over C?

- 1 | We can use Rust code in the kernel!
- 2 | It's so much easier to write (albeit unsafe) Rust
- 3 | Generate code like there's no tomorrow
- 4 | We can add compile-time check before trying to load into the kernel
- 5 | Directly share code between kernel and user-space
- 6 | Swiftly move between high level and low level abstractions
- 7 | Better tooling

---

# InGRAINd vs BCC

- 1 | InGRAINd targets only Rust
- 2 | BCC only supports C, plus bindings to get data into other languages
- 3 | bpftrace is a really cool language built on BCC
- 4 | BCC needs a C compiler and kernel sources wherever you run it
- 6 | InGRAINd only needs the sources and toolchain once
- 7 | InGRAINd can cross-compile for embedded architectures

---

# I'm in the Matrix

```
$ cargo install cargo-bpf
```

```
$ git clone -b ingraind-probes https://github.com/alessandro/ingraind
```

BETA

```
$ cargo build --release
```

```
...
```

```
...
```

```
...
```

```
< build fails because you are missing X >
```



# Writing our first XDP probe

```
$ cd ingraind-probes
```

```
$ cargo bpf add block-http
```

```
$ ls -lR src
```

```
...
```

```
src/block-http:
```

```
total 8
```

```
-rw-r--r-- 1 p2501 p2501 1099 Oct 29 17:03 main.rs
```

```
-rw-r--r-- 1 p2501 p2501  124 Oct 29 17:03 mod.rs
```

This is the only file we need to edit



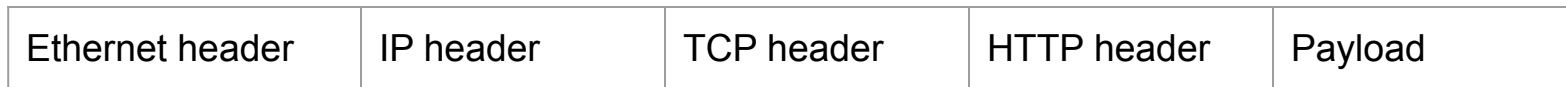
We'll get back to this one



# Intermission

We want to detect HTTP, which is in TCP, which is in IP, which is in 802.3 AKA Ethernet.

Linux gives our XDP probe the whole memory buffer, but then it's up to us to climb the protocols.



This is where we detect the "HTTP/1.1" version string

# How this looks like in C

```
u16 eth_proto = load_half(skb, offsetof(struct ethhdr, h_proto));
u8 ip_proto = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));

/* Skip non-802.3 protocols.
 */
if (!(eth_proto == ETH_P_IP && ip_proto == IPPROTO_TCP))
{
    goto IGNORE;
}

u8 iphlen = (load_byte(skb, ETH_HLEN) & 0x0F) << 2;
u8 tcplen = ((load_byte(skb, ETH_HLEN + iphlen + 12)) >> 4) << 2;

u8 http = ETH_HLEN + iphlen + tcplen;

...
...
```

# My C is getting Rusty

```
#[xdp("block_http")]
pub extern "C" fn probe(ctxp: *mut xdp_md) -> XdpAction {
    let ctx = unsafe { *ctxp };
    let (ip, transport) = match (ctx.ip(), ctx.transport()) {
        (Some(i), Some(t @ Transport::TCP(_))) => (unsafe { *i }, t),
        _ => return XdpAction::Pass
    };

    let data = match ctx.data() {
        Some(data) => data,
        None => return XdpAction::Pass
    };

    ...

    ...

}
```

# My C is getting Rusty

```
#[xdp("block_http")]
pub extern "C" fn probe(ctxp: *mut xdp_md) -> XdpAction {
    let ctx = unsafe { *ctxp };
    let (ip, transport) = match (ctx.ip(), ctx.transport()) {
        (Some(i), Some(t @ Transport::TCP(_))) => (unsafe { *i }, t),
        _ => return XdpAction::Pass
    };

    let data = match ctx.data() {
        Some(data) => data,
        None => return XdpAction::Pass
    };

    ...

    ...

}
```

← Ignore if not TCP/IP traffic

← &mut [u8]

---

# String match to make a decision?



There is no string

---

# No school like old school

```
// the array knows its own length through .len()
let http = ['H', 'T', 'T', 'P', '/', '1', '.', '1'];

// this is basically a memcmp manually
for i in 0..http.len() {
    if data[i] != http[i] {
        return XdpAction::Pass;
    }
}

return XdpAction::Reject;
```