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

3 | Network stack implementations

4 | System calls, kernel, and userland

5 | Deep dive into Linux internals

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

INGRAIN by RED SIFT

Examination Criteria

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

1 | Send a PR, get a Pi

- ➢ Original contributions only
- ➢ Feel free to explore, and I’m happy to help
- ➢ We have a few bugs/compiler oddness
- ➢ 2/day max

2 | We’re hiring in Barcelona and London: peter@redsift.io

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

||

1 VirtualBox

1 vagrantup.com

1 Vagrantfile

||

1 Linux 4.19+

1 LLVM 9

1 Kernel source

1 Editor
1 Rust toolchain
30 kg patience

INGRAIN by RED SIFT

Using the Raspberry Pi

When you connect through Ethernet, you get an IP through DHCP.

Then:

```
ssh root@10.13.37.1
```

There's a NextCloud server running on `http://10.13.37.1/`

There is also a MariaDB running in Docker.

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

INGRAIN by RED SIFT

Instructions

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

Instructions

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

Instructions

jne load cmp cmp load cmp

6 1 3 5 2 4

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

7 jump

Linux <5.4 ==> 4096

Linux >=5.4 ==> 1 000 000

Eliminate the state

Computers are stateful, no matter how hard we try.

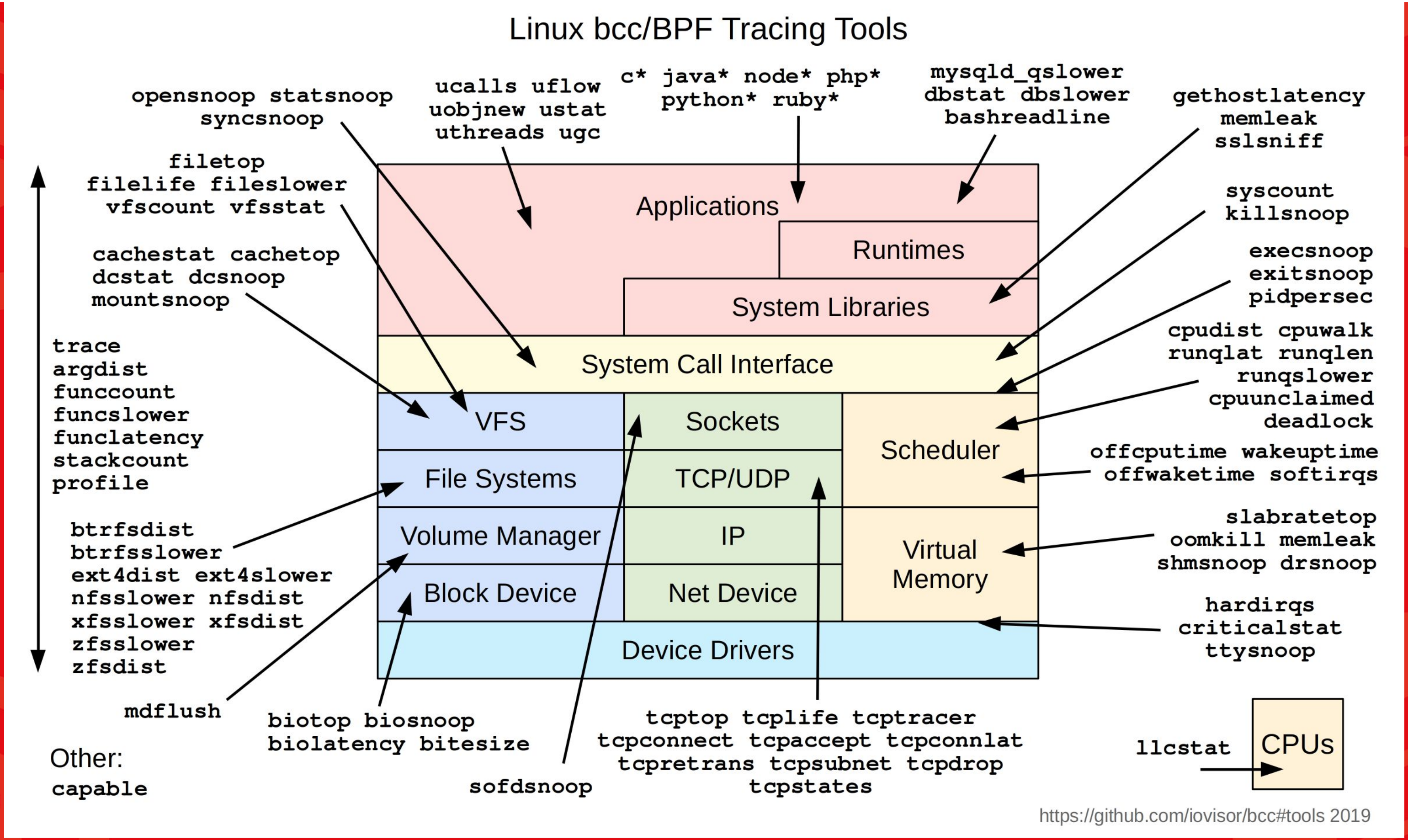
Docker **containers** can be stateless, but mostly aren't.

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

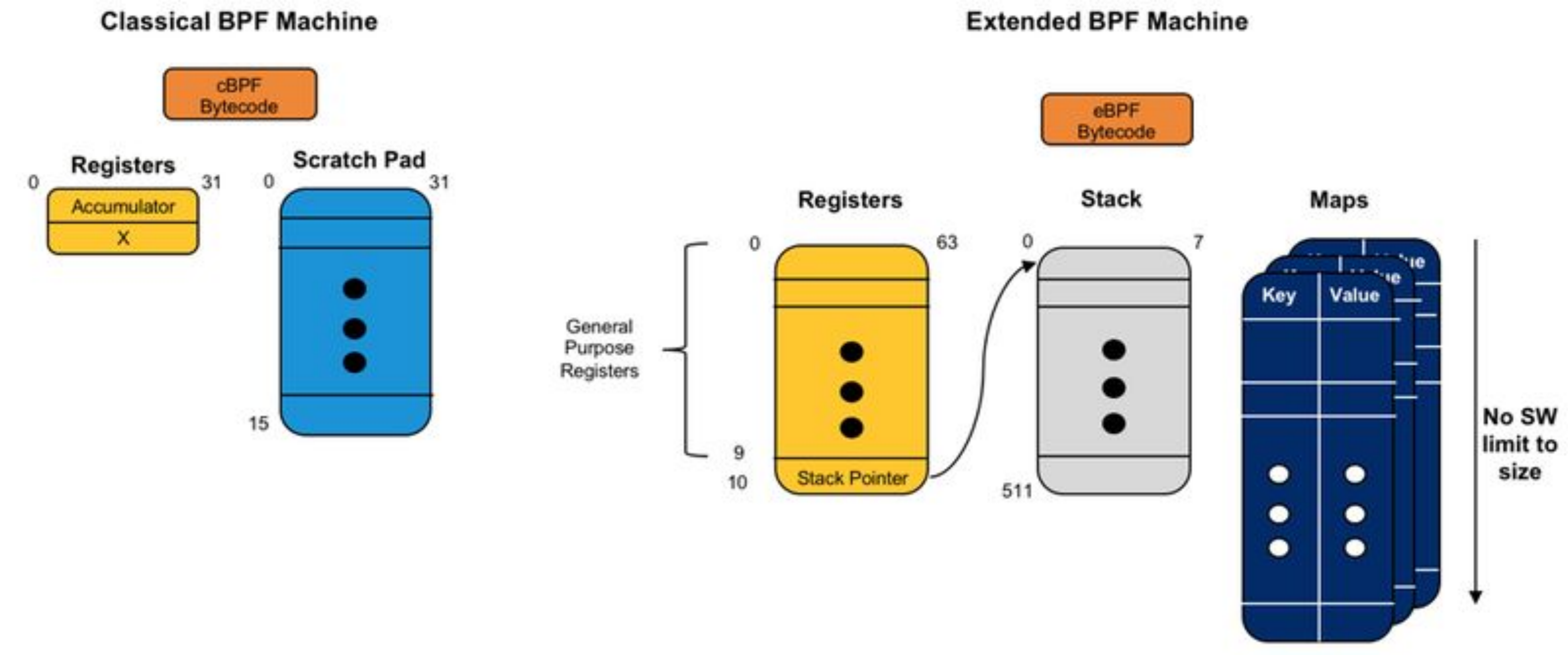
Kubernetes *StatefulSet* vs *Deployment* is about long-term state.

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

Consensus by blockchain/Paxos.



Linux bcc/BPF Tracing Tools
opensnoop statsnoop
syncsnoop
ucalls uflow
uobjnew ustat
uthreads ugc
c* java* node* php*
python* ruby*
mysqld_qslower
dbstat dbslower
bashreadline
gethostlatency
memleak
sslsniff
filetop
filelife fileslower
vfscount vfsstat
cachestat cachetop
dcstat dcsnoop
mountsnoop
syscount
killsnoop
execsnoop
exitsnoop
pidpersec
Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers
trace
argdist
funccount
funcslower
funclatency
stackcount
profile
cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock
offcputime wakeuptime
offwaketime softirqs
btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower
zfsdist
slabratetop
oomkill memleak
shmsnoop drsnoop
hardirqs
criticalstat
ttysnoop
mdflush
biotop biosnoop
biolatency bitesize
tcptop tcplife tcptracer
tcpconnect tcpaccept tcpconnlat
tcpretrans tcpsubnet tcpdrop
tcpstates
llcstat
CPUs
Other:
capable
sofdsnoop
https://github.com/iovisor/bcc#tools 2019



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

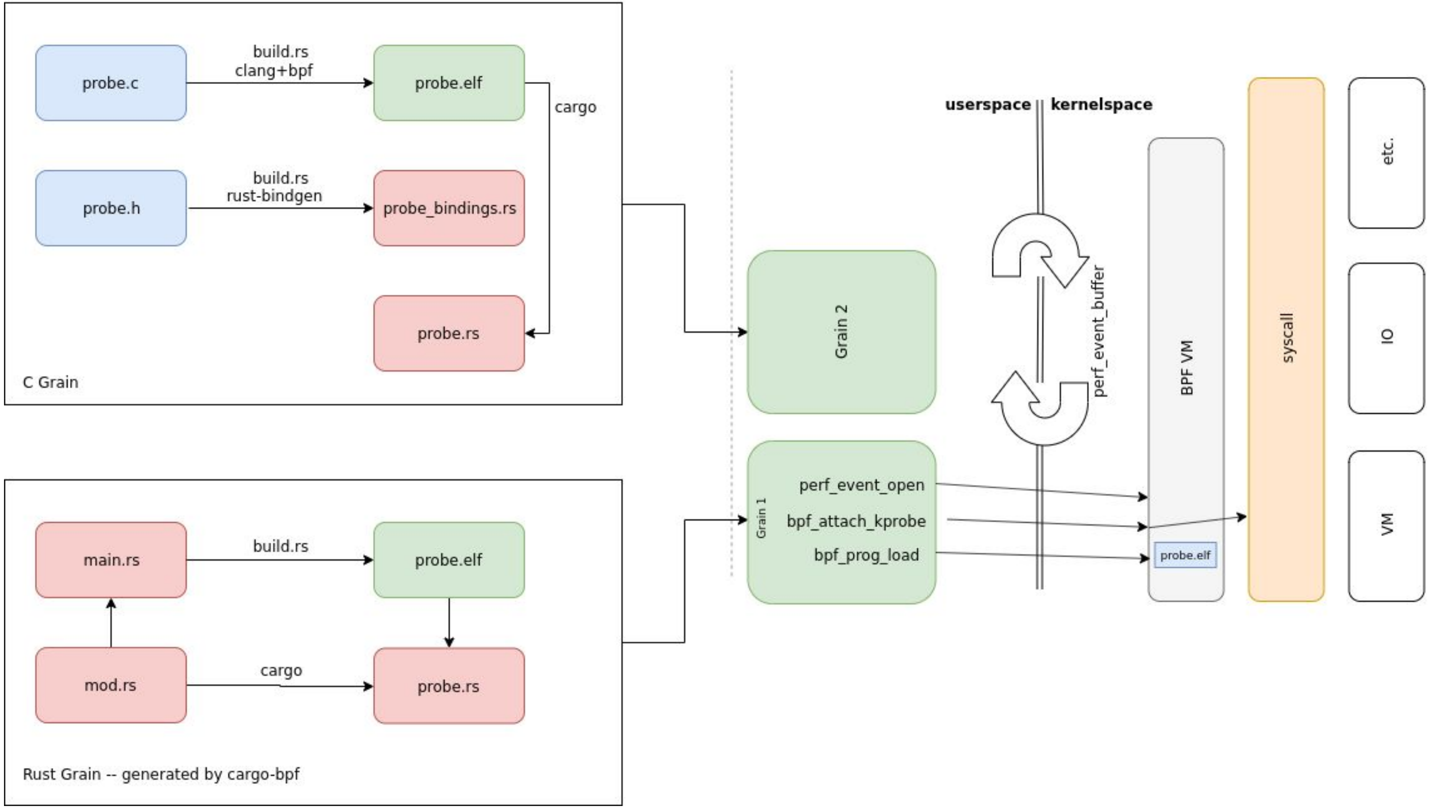
Introducing InGRAINd

An eBPF framework written in Rust

- ➢ Supports eBPF programs written in Rust or C
- ➢ Compile once, deploy everywhere
- ➢ High performance metrics aggregator
- ➢ Filtering/rewrite pipeline and configuration interface

Build

Runtime



INGRAIN by RED SIFT

https://github.com/redsift/ingraind-rf-bcn

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

I'm in the Matrix

```
$ cargo install cargo-bpf

$ git clone -b v1.0 https://github.com/redsift/ingraind

$ 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

INGRAIN by RED SIFT

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.

Ethernet header	IP header	TCP header	HTTP header	Payload

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

What this looks like in C

```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;
```

…
…

INGRAIN by RED SIFT

My C is getting Rusty

```
#[xdp("block_http")]
pub extern "C" fn probe(ctx: *mut xdp_md) -> XdpAction {
   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,
   };
```

INGRAIN by RED SIFT

My C is getting Rusty

```
#[xdp("block_http")]
pub extern "C" fn probe(ctx: *mut xdp_md) -> XdpAction {
    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

INGRAIN by RED SIFT

String match to make a decision?

No school like old school

```rust
let http = ['H', 'T', 'T', 'P', '/', '1', '.', '1'];
let iters = http.len();

let mut decision = 1;
if let Some(header) = data.slice(iters) {
    for i in 0..8 {
        if header[i] != http[i] as u8 {
            decision = 0;
        }
    }
} else {
    decision = 0;
};
```

```rust
if decision == 1 {
    XdpAction::Drop
} else {
    XdpAction::Pass
}
```

INGRAIN by RED SIFT

Running the XDP probe

```
$ cd ingraind-probes

$ cargo bpf build block_http

$ cargo bpf load -i eth0 ingraind-probes/target/release/bpf-programs/block_http/block_http.elf
```

Try reaching the VM over http://10.13.37.1

It will just time out.

INGRAIN by RED SIFT

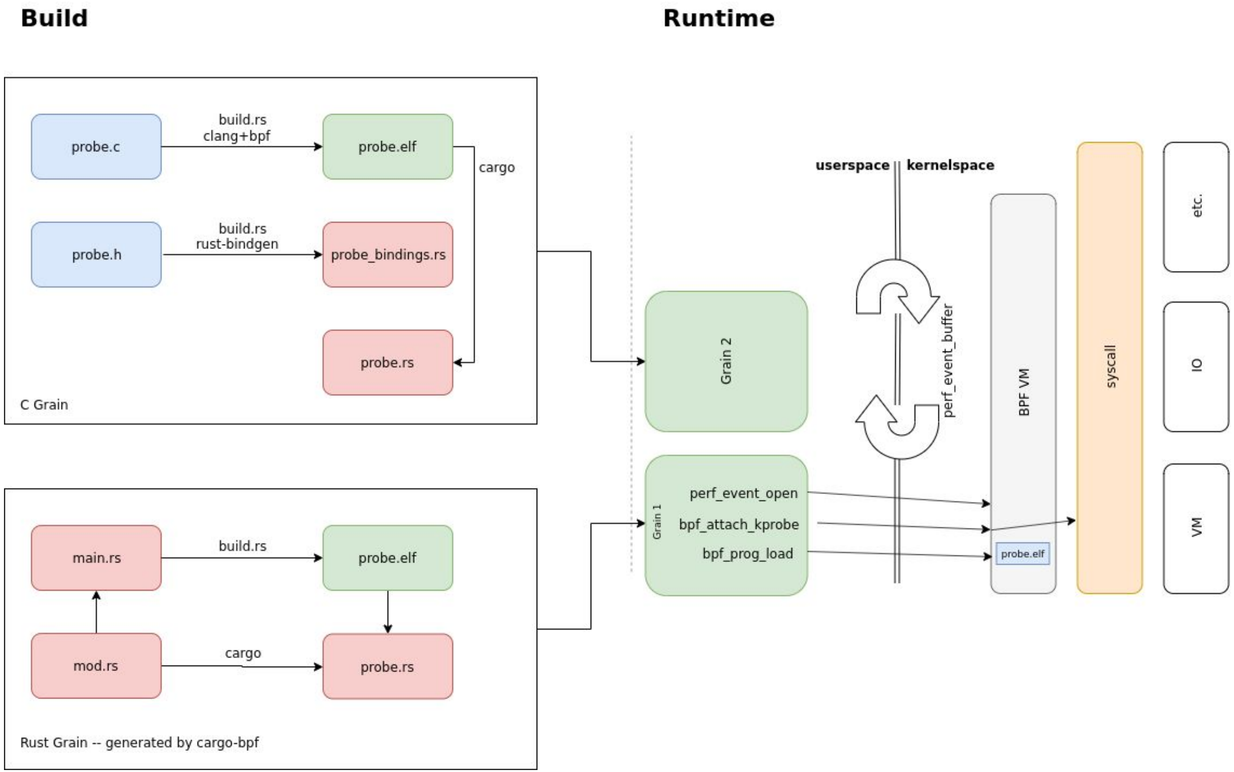
Let's stretch a bit

The XDP probe is now blocking traffic over any port that contains the `HTTP/1.1` string in clear text.

We didn't even need ingraind to run, just the probe! What is this magic?

It's just a bunch of system calls, and binding an XDP program to an interface.

Back to the theory



INGRAIN by RED SIFT

Communicating with userspace

Communicating with userspace is done through eBPF maps

- ➢ There are several types of maps: key-value stores, perf maps, etc
- ➢ Userspace and kernel space both have calls to access these
- ➢ We need some boilerplate
- ➢ InGRAINd takes care of the boring things, so we can rely on copy-pasta

Let's extract some metrics

Write an eBPF program that blocks HTTP traffic, but also reports some things to our backend.

main.rs

```
#[map("events")]
static mut events: PerfMap<HTTPBlocked> =
PerfMap::with_max_entries(1024);

#[xdp("block_http")]
pub extern "C" fn probe(ctx: *mut xdp_md) -> XdpAction
{
...
   if decision == 1 {
       let event = HTTPBlocked {
           saddr: ip.saddr,
           daddr: ip.daddr,
           sport: 0,
       };
       unsafe { events.insert(&ctx, event, 0) };
   }
```

mod.rs

```
#[repr(C)]
#[derive(Debug)]
pub struct HTTPBlocked {
    pub saddr: u32,
    pub daddr: u32,
    pub sport: u32,
}
```

INGRAIN by RED SIFT

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

5 | InGRAINd only needs the sources and toolchain once

6 | InGRAINd can cross-compile for embedded architectures

INGRAIN by RED SIFT