



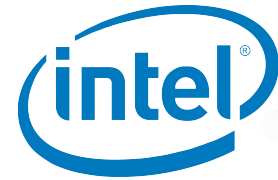
# BoF - What Can BPF Do For You?

Brenden Blanco  
Aug. 22, 2016

# Agenda

- A bit of history and project motivation
- An introduction to eBPF in the Linux kernel
- An introduction to the BCC toolkit
- Show how Clang/LLVM is integrated into BCC
- Demo how to use IO Visor+XDP for DDoS mitigation
- Demo how to use IO Visor to debug a live system
- Q+A

# Thank You to Sponsoring Members



# What we want

Started with building networking applications for SDN

An SDK to extend low-level infrastructure

But...

Don't want to become a kernel developer

# Compare to a server app framework (e.g. Node.js)

Recognize that writing multithreaded apps is hard

Syntax that mirrors thought process, not the CPU arch (events vs threads)

Don't sacrifice performance (v8 jit)

Make it easy to get code from the devs to deployment (npm)

Foster a community via sharing of code



# What do you need to write infrastructure apps

High performance access to data

Reliability...it must never crash

In-place upgrades

Debug tools

A programming language abstraction

# But there are restrictions

No custom kernels

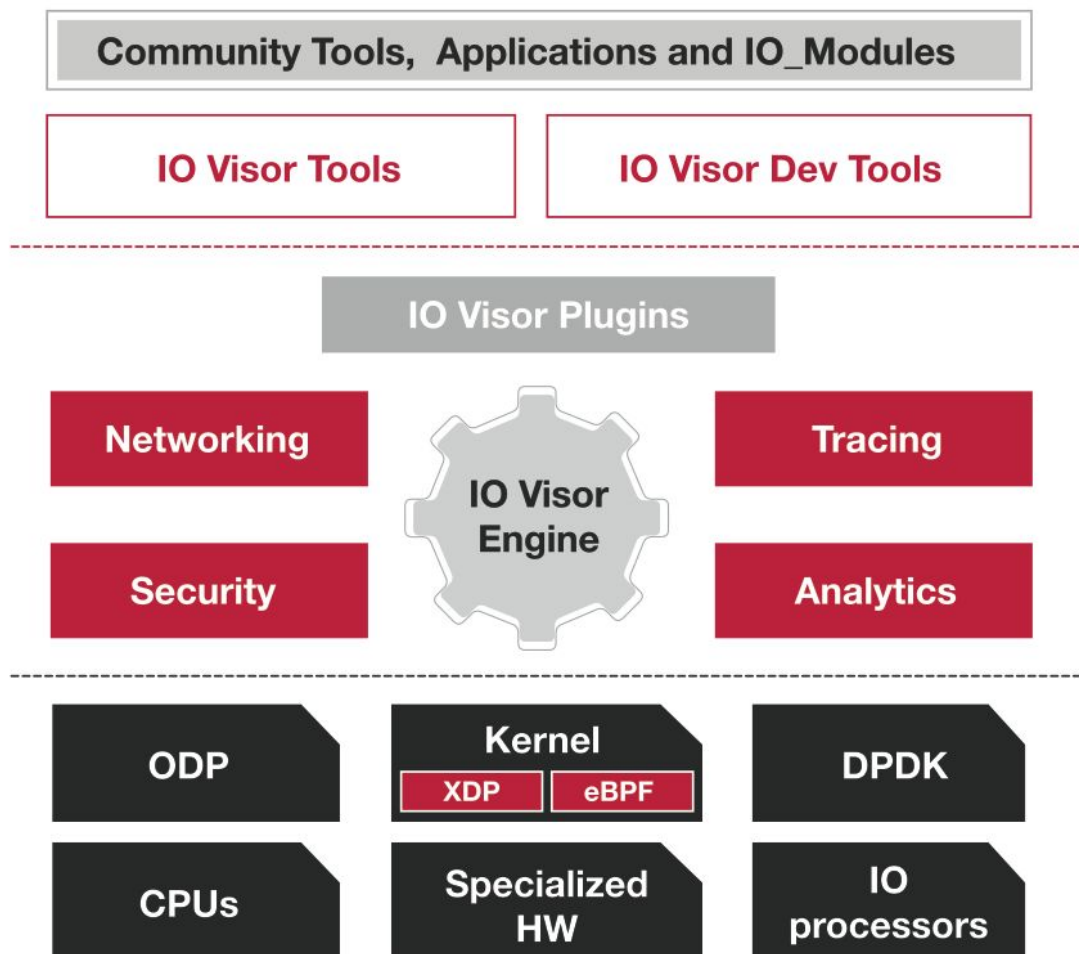
No custom kernel modules

No kernels with debug symbols

No reboots

(some of these are nice-to-haves)

# IO Visor Project, What is in it?



- A set of development tools, **IO Visor Dev Tools**
- A set of **IO Visor Tools** for management and operations of the IO Visor Engine
- A set of Applications, Tools and open **IO Modules** build on top of the IO Visor framework
- A set of possible use cases & applications like **Networking, Security, Tracing & others**



# Hello, World! Demo

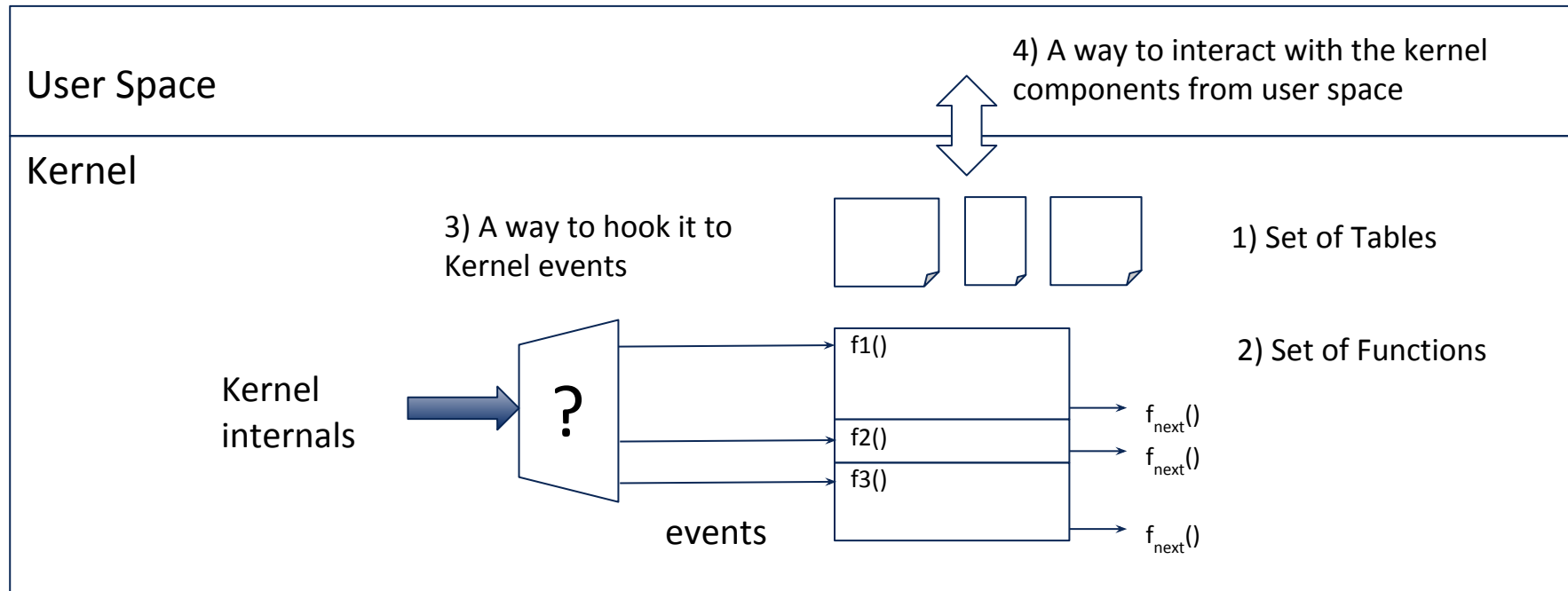
```
#!/usr/bin/python
import bcc
b = bcc.BPF(text="""
int kprobe__sys_clone(void *ctx) {
    bpf_trace_printk("Hello, World!\\n");
    return 0;
}
""")
b.trace_print()
```

# BPF

# What are BPF Programs?

In a very simplified way:

A safe, runtime way to extend Linux kernel capabilities  
Functions, Maps, Attachment Points, Syscall



# More on BPF Programs

Berkeley Packet Filters around since 1990, extensions started Linux 3.18

Well, not really a program (no pid)...an event handler

A small piece of code, executed when an event occurs

In-kernel virtual machine executes the code

Assembly instruction set

See 'man 2 bpf' for details

# The eBPF Instruction Set

## Instructions

- 10x 64bit registers
- 512B stack
- 1-8B load/store
- conditional jump
- arithmetic
- function call

## Helper functions

- forward/clone/drop packet
- load/store packet data
- load/store packet metadata
- checksum (incremental)
- push/pop vlan
- access kernel mem (kprobes)

## Data structures

- lookup/update/delete
  - in-kernel or from userspace
- hash, array, ...



# BPF Kernel Hook Points

A program can be attached to:

- kprobes or uprobes

- socket filters (original tcpdump use case)

- seccomp

- tc filters or actions, either ingress or egress

- XDP (*NEW*)

# BPF Verifier

A program is declared with a type (kprobe, filter, etc.)

Only allows permitted helper functions

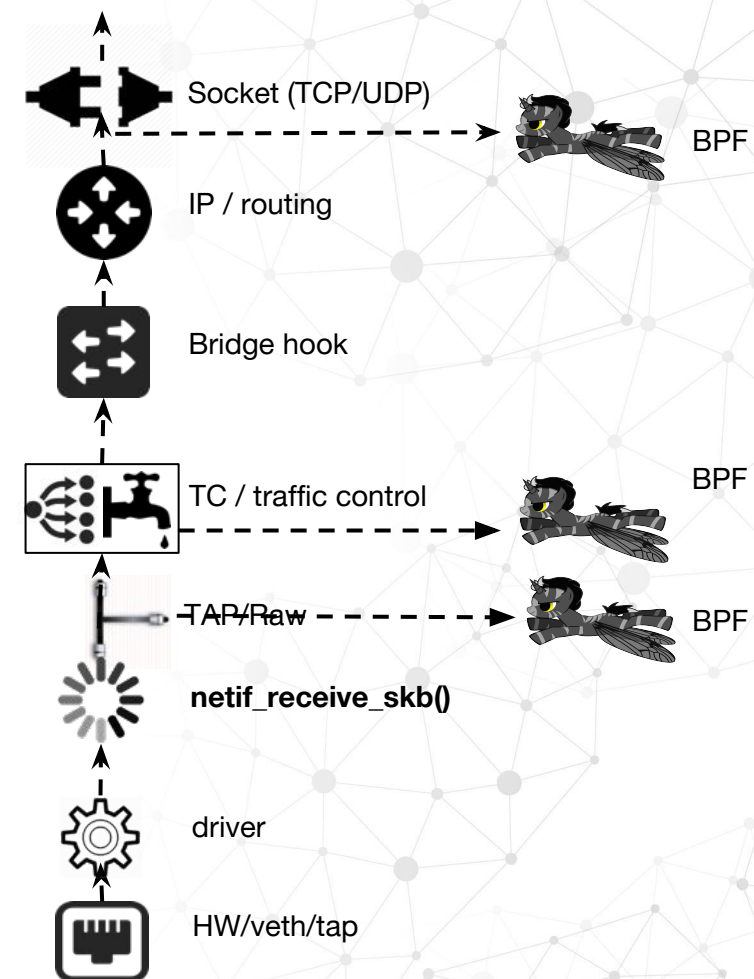
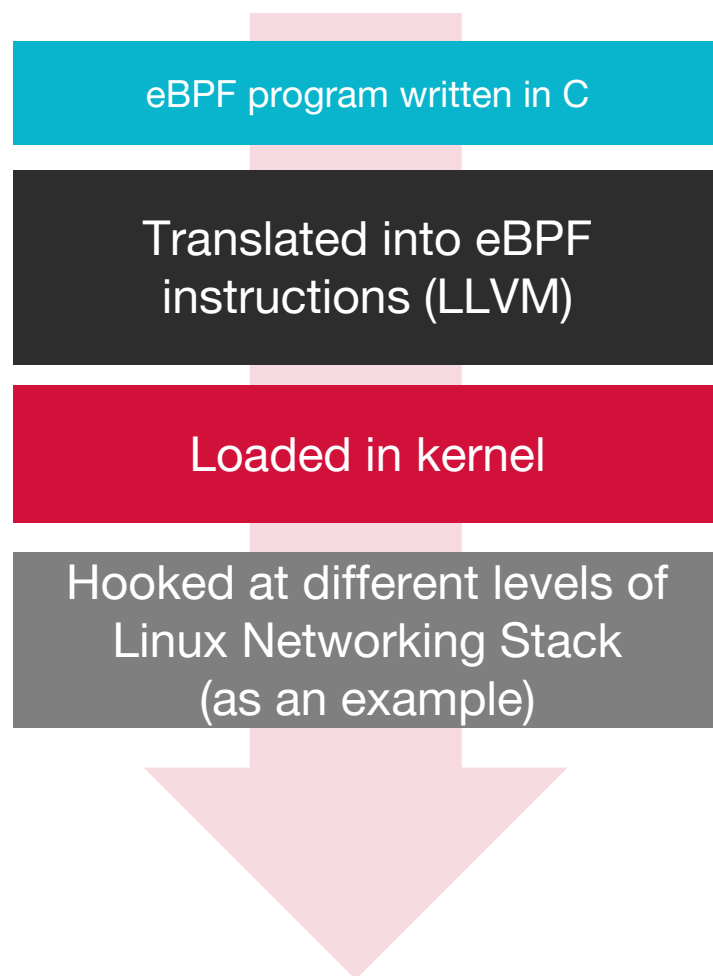
Kernel parses BPF instructions into a DAG

Disallows: back edges, unreachable blocks, illegal insns, finite execution

No memory accesses from off-stack, or from unverified source

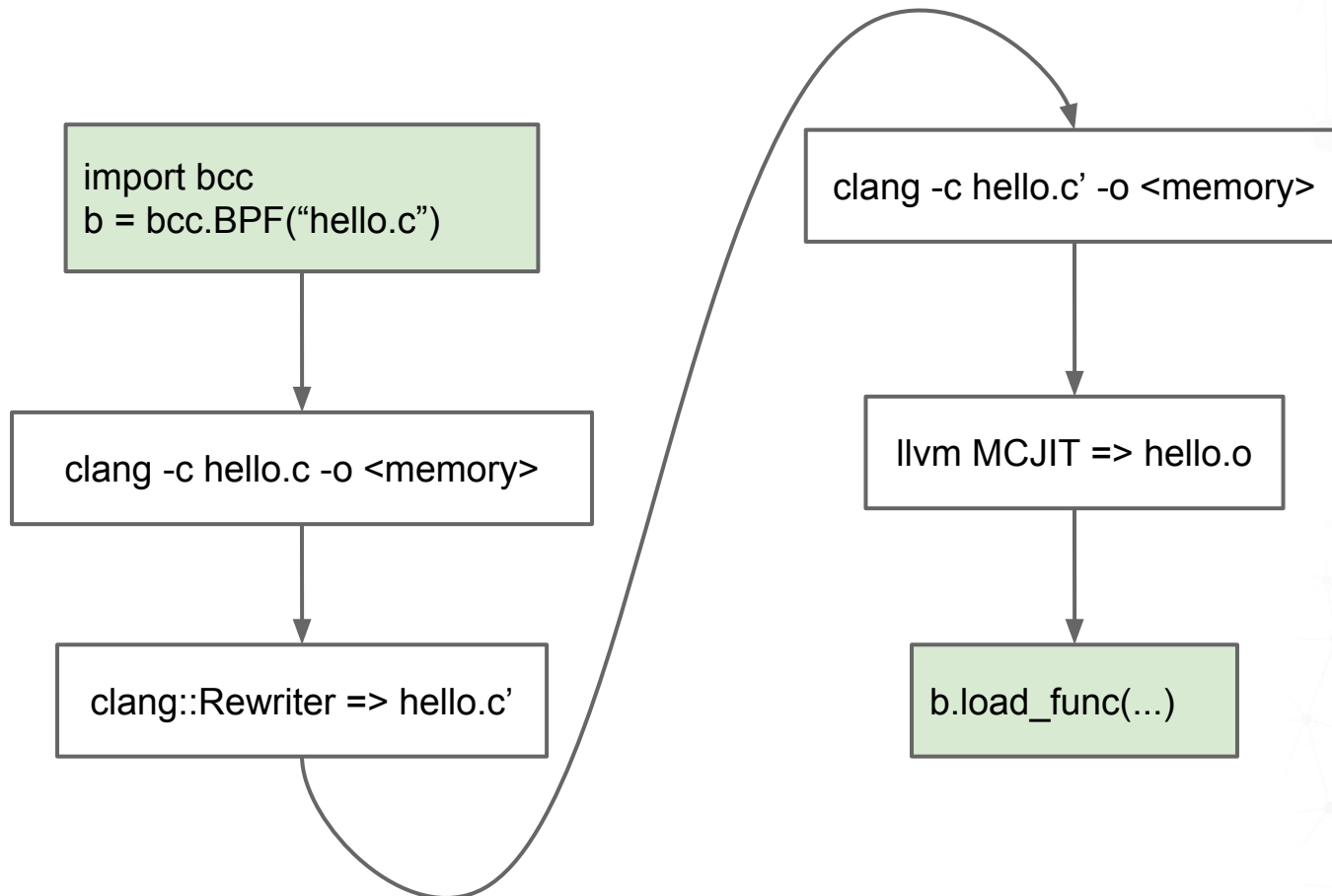
Program ok? => JIT compile to native instructions (x86\_64, arm64, s390)

# Developer Workflow



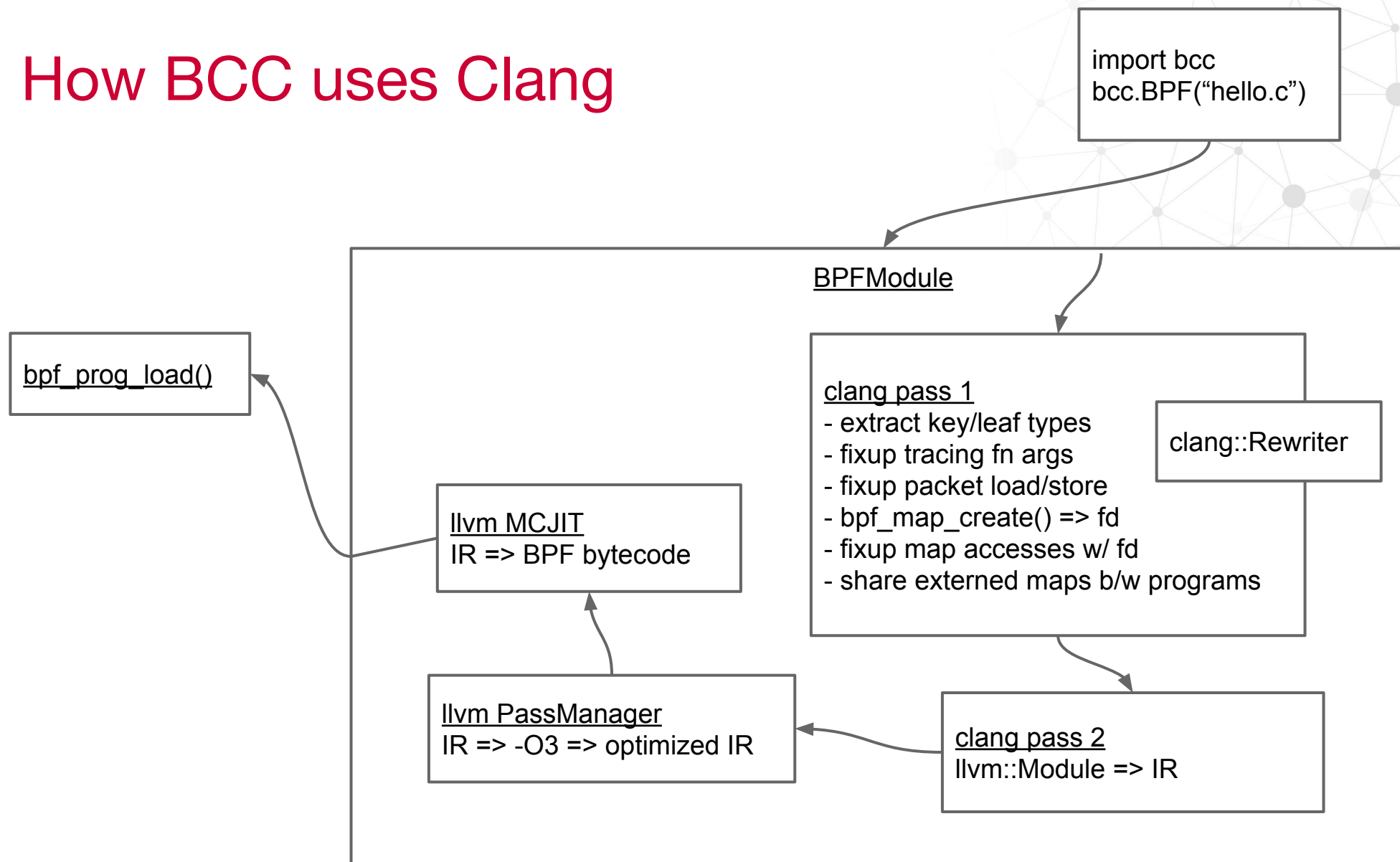
# Using Clang and LLVM in BCC

# How BCC uses Clang





# How BCC uses Clang



# Rewrite Sample #1

```
#include <uapi/linux/ptrace.h>
int do_request(struct pt_regs *ctx, int req) {
    bpf_trace_printk("req ptr: 0x%x\n", req);
    return 0;
}
```

```
#include <uapi/linux/ptrace.h>
int do_request(struct pt_regs *ctx, int req) {
    ({
        char _fmt[] = "req ptr: 0x%x\n";
        bpf_trace_printk_(_fmt, sizeof_(fmt), ((u64)ctx->di));
    });
    return 0;
}
```

# Rewrite Sample #2

```
#include <linux/sched.h>
#include <uapi/linux/ptrace.h>

int count_sched(struct pt_regs *ctx,
                struct task_struct *prev) {
    pid_t p = prev->pid;
    return p != -1;
}
```

# Rewrite Sample #2

```
#include <linux/sched.h>
#include <uapi/linux/ptrace.h>

int count_sched(struct pt_regs *ctx,
                struct task_struct *prev) {
    pid_t p = ({
        pid_t _val;
        memset(&_val, 0, sizeof(_val));
        bpf_probe_read(&_val, sizeof(_val),
                      ((u64)ctx->di) + offsetof(struct task_struct, pid));
        _val;
    });
    return p != -1;
}
```

# Rewrite Sample #3

```
#include <bcc/proto.h>
struct IPKey { u32 dip; u32 sip; };
BPF_TABLE("hash", struct IPKey, int, mytable, 1024);
int recv_packet(struct __sk_buff *skb) {
    struct IPKey key;
    u8 *cursor = 0;
    struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet));
    struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
    key.dip = ip->dst;
    key.sip = ip->src;
    int *leaf = mytable.lookup(&key);
    if (leaf)
        *(leaf)++;
    return 0;
}
```



# Rewrite Sample #3

```
#include <bcc/proto.h>
struct IPKey { u32 dip; u32 sip; };
BPF_TABLE("hash", struct IPKey, int, mytable, 1024);
int recv_packet(struct __sk_buff *skb) {
    struct IPKey key;
    u8 *cursor = 0;
    struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet));
    struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
    key.dip = bpf_dext_pkt(skb, (u64)ip+16, 0, 32);
    key.sip = bpf_dext_pkt(skb, (u64)ip+12, 0, 32);
    int *leaf = bpf_map_lookup_elem((void *)bpf_pseudo_fd(1, 3), &key);
    if (leaf)
        *(leaf)++;
    return 0;
}
```

# Using BCC for Tracing

# Tracing Demo

<https://github.com/iovisor/bcc>

<http://www.brendangregg.com/blog>

# XDP for Networking

# What is XDP?

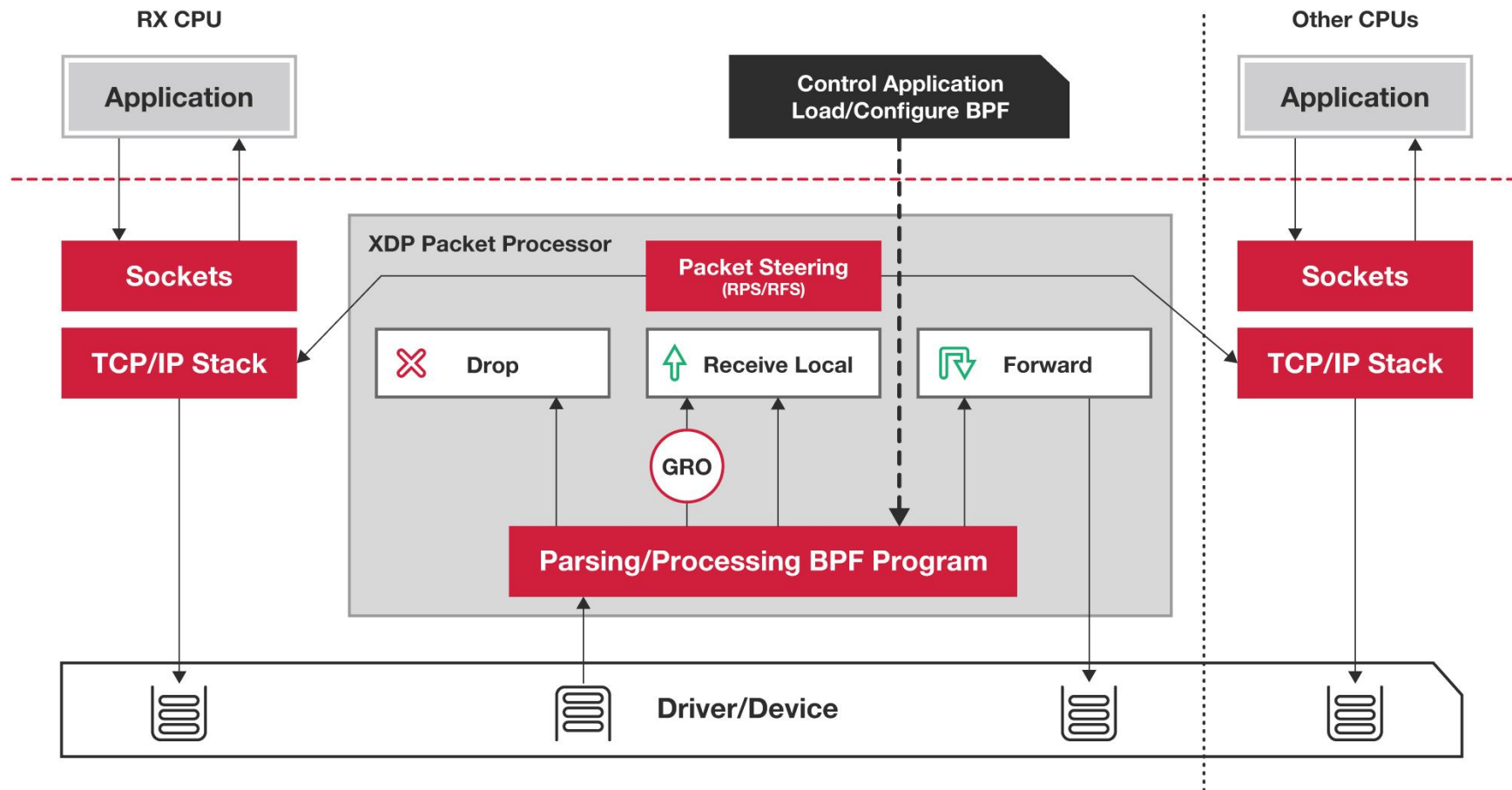
- A programmable, high performance, specialized application, packet processor in the networking data path
- Bare metal packet processing at lowest point in the SW stack
- Use cases include
  - Pre-stack processing like filtering to do DOS mitigation
  - Forwarding and load balancing
  - Batching techniques
  - Flow sampling, monitoring



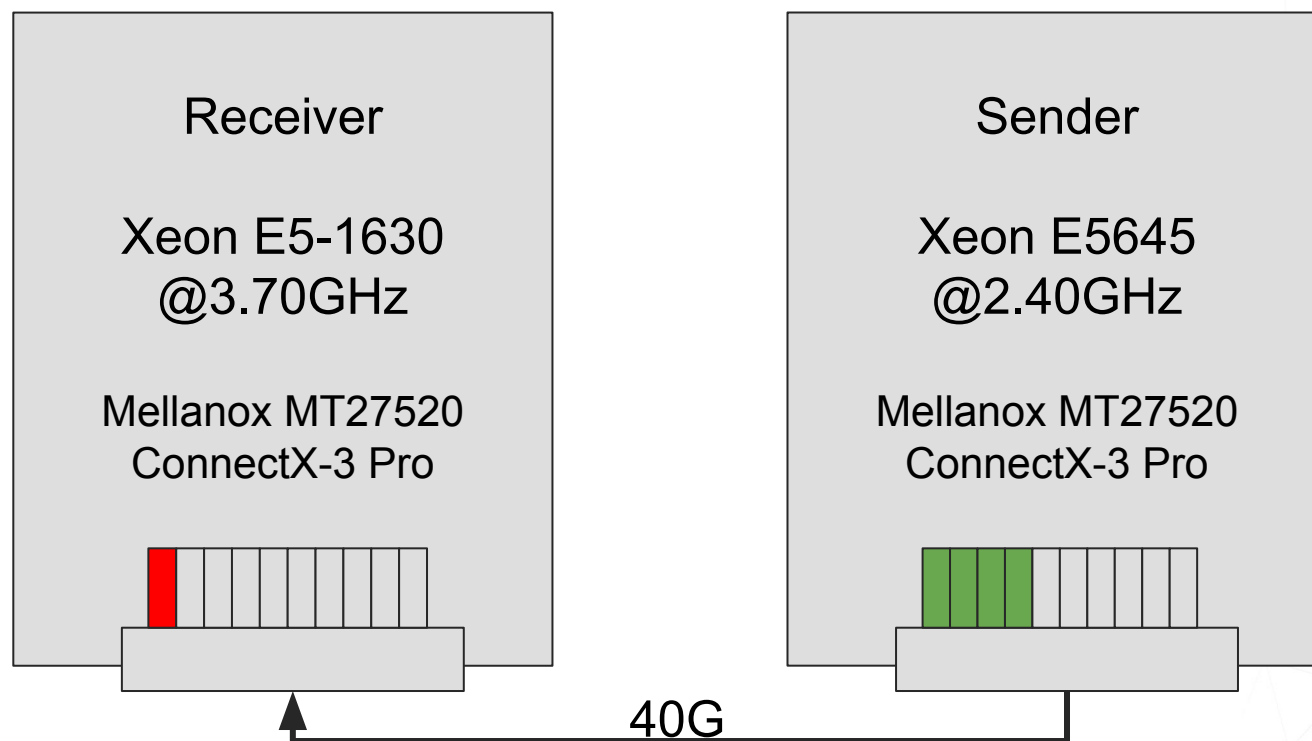
# XDP Properties

- XDP is **designed for high performance**. It uses known techniques and applies selective constraints to achieve performance goals
- XDP is also **designed for programmability**. New functionality can be implemented on the fly without needing kernel modification
- XDP is **not kernel bypass**. It is an integrated fast path in the kernel stack
- XDP **does not replace the TCP/IP stack**. It augments the stack and works in concert
- XDP **does not require any specialized hardware**. Less-is-more principle for networking hardware

# eXpress Data Path (XDP)



# XDP Benchmark Setup



# Thank You!

# Learn More and Contribute

<https://iovisor.org>

<https://github.com/iovisor>

#iovisor irc.oftc.net

@IOVisor