

Ah, you mean **eBPF** — the **extended Berkeley Packet Filter**. This is very different from EBFS. eBPF is a Linux kernel technology that lets you run sandboxed programs in the kernel safely. It's often used for **networking, observability, tracing, and security**.

In simple terms: eBPF programs are small, safe programs that the kernel can run in response to events, like network packets, syscalls, or tracepoints. You can attach eBPF programs to hooks in the kernel and gather data or enforce policies without changing kernel code.

Here's a very basic example:

---

### Example: Trace the `execve` system call using eBPF (C + BPF Compiler Collection)

```
// Filename: trace_exec.c
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

BPF_PERF_OUTPUT(events);

struct data_t {
    u32 pid;
    char comm[TASK_COMM_LEN];
};

int trace_exec(struct pt_regs *ctx) {
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

You'd typically compile this using **BCC (BPF Compiler Collection)** or **libbpf**, then attach it to the `execve` syscall. In Python, you can read the events:

```
from bcc import BPF

b = BPF(src_file="trace_exec.c")
b.attach_kprobe(event="sys_execve", fn_name="trace_exec")

def print_event(cpu, data, size):
    event = b["events"].event(data)
    print(f"PID {event.pid} executed {event.comm.decode()}")

b["events"].open_perf_buffer(print_event)

while True:
    b.perf_buffer_poll()
```

---

## Key Points:

- **Safe in kernel:** eBPF programs run in a sandbox; kernel won't crash even if your code is buggy.
  - **Use cases:** Profiling, observability, network packet filtering, security monitoring.
  - **Languages:** Mostly C, compiled to BPF bytecode. You can also use higher-level frameworks like **BCC**, **libbpf**, **bpftrace**, or even **Go** bindings.
- 

If you want, I can also show a **super simple example that counts network packets per process**, which is one of the classic eBPF use cases. It's easier to test on a live system.

Do you want me to show that?

Great! Here's a straightforward example: **counting network packets per process using eBPF** with Python and BCC. This is a common way to monitor which processes are generating network traffic.

---

### eBPF C code (`packet_count.c`)

```
#include <uapi/linux/ptrace.h>
#include <net/sock.h>
#include <linux/skbuff.h>
#include <bcc/proto.h>

BPF_HASH(packet_count, u32, u64);

int count_packets(struct __sk_buff *skb) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u64 *count = packet_count.lookup(&pid);
    if (count) {
        (*count)++;
    } else {
        u64 initial = 1;
        packet_count.update(&pid, &initial);
    }
    return 0;
}
```

- `packet_count` is a hash map keyed by process ID (`pid`) that stores the number of packets.
  - `count_packets` runs for each packet processed by the kernel hook.
- 

## Python script to attach the eBPF program

```

from bcc import BPF
import time

b = BPF(src_file="packet_count.c")
b.attach_kprobe(event="__netif_receive_skb", fn_name="count_packets")

print("Monitoring packets per process... Ctrl-C to exit.")

try:
    while True:
        time.sleep(5)
        print("\nPID      Packets")
        for pid, count in b["packet_count"].items():
            print(f"{pid.value:5} {count.value:8}")
except KeyboardInterrupt:
    print("Exiting.")

```

---

## How it works

1. `attach_kprobe` hooks into the kernel function `__netif_receive_skb`, which is called for each incoming packet.
  2. The eBPF program increments a counter in a map for the process ID that received the packet.
  3. The Python script reads and prints the map every 5 seconds.
- 

This is a **live network monitoring example** and requires **root** access. You can expand it to count packets per user, per interface, or even track bytes instead of packets.

I can also make a **version that only monitors TCP packets from a specific port** if you want a more focused example. Do you want me Great! Here's a more complete example that monitors **both incoming and outgoing TCP packets per process**, optionally filtered by a specific port like 80.

---

## eBPF C code (`tcp_io_count.c`)

```

#include <uapi/linux/ptrace.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <bcc/proto.h>

BPF_HASH(tcp_io_count, u32, u64);

int count_tcp_io(struct __sk_buff *skb) {
    u8 *cursor = 0;

```

```

struct ethernet_t *eth = cursor_advance(cursor, sizeof(*eth));
if (eth->type != 0x0800) // Only IPv4
    return 0;

struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
if (ip->nextp != IPPROTO_TCP)
    return 0;

struct tcp_t *tcp = cursor_advance(cursor, sizeof(*tcp));
u16 sport = tcp->src_port;
u16 dport = tcp->dst_port;

// Filter only TCP port 80 (incoming or outgoing)
if (sport != 80 && dport != 80)
    return 0;

u32 pid = bpf_get_current_pid_tgid() >> 32;
u64 *count = tcp_io_count.lookup(&pid);
if (count) {
    (*count)++;
} else {
    u64 initial = 1;
    tcp_io_count.update(&pid, &initial);
}

return 0;
}

```

---

## Python script to attach it

```

from bcc import BPF
import time

b = BPF(src_file="tcp_io_count.c")
b.attach_kprobe(event="__netif_receive_skb", fn_name="count_tcp_io")

print("Monitoring TCP packets on port 80 (incoming/outgoing) per process...
Ctrl-C to exit.")

try:
    while True:
        time.sleep(5)
        print("\nPID      TCP Packets on port 80")
        for pid, count in b["tcp_io_count"].items():
            print(f"{pid.value:5} {count.value:8}")
except KeyboardInterrupt:
    print("Exiting.")

```

---

## How this version works

1. It hooks into \_\_netif\_receive\_skb for all packet activity.

2. It checks both **source port** and **destination port** for filtering (so both inbound and outbound on port 80).
3. It keeps a **per-process count** using an eBPF hash map.
4. The Python script prints counts every 5 seconds.

This approach gives a **complete TCP port 80 activity overview** per process.

If you want, I can also show a **more efficient version that uses XDP for higher performance**, which can handle thousands of packets per second without dropping events. That's closer to how production eBPF network monitoring works.

Do you want me to do that?

to do that?

Perfect! Let's step it up. Using **XDP (eXpress Data Path)** with eBPF is much faster than kprobes because it processes packets **directly at the network driver level**, before the kernel network stack. This is ideal for high-performance packet counting or filtering.

Here's a simple example to **count TCP packets on port 80 per process** using XDP.

---

### eBPF C code for XDP (`xdp_tcp_count.c`)

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <bpf/bpf_helpers.h>

struct bpf_map_def SEC("maps") tcp_count_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(__u32),
    .value_size = sizeof(__u64),
    .max_entries = 1024,
};

SEC("xdp")
int xdp_count_tcp(struct xdp_md *ctx) {
    void *data_end = (void *) (long) ctx->data_end;
    void *data = (void *) (long) ctx->data;

    struct ethhdr *eth = data;
    if ((void*)eth + sizeof(*eth) > data_end)
        return XDP_PASS;

    if (eth->h_proto != __constant_htons(ETH_P_IP))
        return XDP_PASS;

    struct iphdr *ip = data + sizeof(*eth);
```

```

if ((void*)ip + sizeof(*ip) > data_end)
    return XDP_PASS;

if (ip->protocol != IPPROTO_TCP)
    return XDP_PASS;

struct tcphdr *tcp = (void*)ip + ip->ihl * 4;
if ((void*)tcp + sizeof(*tcp) > data_end)
    return XDP_PASS;

__u16 sport = __builtin_bswap16(tcp->source);
__u16 dport = __builtin_bswap16(tcp->dest);

if (sport != 80 && dport != 80)
    return XDP_PASS;

__u32 pid = bpf_get_current_pid_tgid() >> 32;
__u64 *count = bpf_map_lookup_elem(&tcp_count_map, &pid);
__u64 one = 1;
if (count)
    *count += 1;
else
    bpf_map_update_elem(&tcp_count_map, &pid, &one, BPF_ANY);

return XDP_PASS;
}

char _license[] SEC("license") = "GPL";

```

---

## Python script to attach XDP program

```

from bcc import BPF
import ctypes
import time
import sys

if len(sys.argv) < 2:
    print("Usage: sudo python3 xdp_tcp_monitor.py <interface>")
    exit(1)

iface = sys.argv[1]

b = BPF(src_file="xdp_tcp_count.c", cflags=["-O2"])
fn = b.load_func("xdp_count_tcp", BPF.XDP)
b.attach_xdp(iface, fn, 0)

print(f"Monitoring TCP packets on port 80 via XDP on interface {iface}...
Ctrl-C to stop")

try:
    while True:
        time.sleep(5)
        print("\nPID      TCP Packets on port 80")
        tcp_map = b.get_table("tcp_count_map")
        for pid, count in tcp_map.items():

```

```
        print(f"{{pid.value:5} {count.value:8}}")
except KeyboardInterrupt:
    print("Detaching XDP program...")
    b.remove_xdp(iface, 0)
```

---

## Notes

1. **XDP attaches at the NIC driver**, so it's extremely fast.
  2. This example still filters TCP port 80 packets, counting per PID.
  3. Requires **root** and a supported Linux kernel ( $\geq 4.8$ ).
  4. `bpf_get_current_pid_tgid()` gives the current PID, though in XDP it may not always be fully accurate for packets coming from other processes. For exact per-process accounting, **tc or socket-level eBPF** might be more precise.
- 

If you want, I can create a **hybrid approach**: XDP for speed + socket-level BPF to accurately map packets to processes. That's what production systems like **Cilium** or **Katran** do.

Do you want me to show that hybrid version?

Perfect! Let's outline a **hybrid approach** combining **XDP for high-speed packet capture** with **socket-level eBPF** for accurate per-process mapping. This is essentially how production systems like Cilium or Katran track per-process network usage at line-rate.

---

## Approach Overview

1. **XDP Layer**
    - o Attached to the network interface for **fast packet filtering**.
    - o Counts TCP packets per port quickly, but **does not know the process** reliably because it runs before the kernel socket layer.
  2. **Socket-level BPF / tc Layer**
    - o Attached at the socket or traffic control (tc) layer.
    - o Associates packets with **actual processes** using `skb->sk` or `bpf_get_current_pid_tgid()`.
    - o Reads XDP counters or increments its own accurate counters.
  3. **User-space Aggregation**
    - o Python/BCC program reads both maps: XDP map for speed, socket map for accuracy.
    - o Combines counts per PID or per port.
-

## Hybrid eBPF C Code Snippets

### XDP Program (fast packet filter):

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <bpf/bpf_helpers.h>

struct bpf_map_def SEC("maps") xdp_tcp_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(__u16), // TCP port
    .value_size = sizeof(__u64), // packet count
    .max_entries = 1024,
};

SEC("xdp")
int xdp_tcp_filter(struct xdp_md *ctx) {
    void *data_end = (void *) (long) ctx->data_end;
    void *data = (void *) (long) ctx->data;

    struct ethhdr *eth = data;
    if ((void*)eth + sizeof(*eth) > data_end) return XDP_PASS;
    if (eth->h_proto != __constant_htons(ETH_P_IP)) return XDP_PASS;

    struct iphdr *ip = data + sizeof(*eth);
    if ((void*)ip + sizeof(*ip) > data_end) return XDP_PASS;
    if (ip->protocol != IPPROTO_TCP) return XDP_PASS;

    struct tcphdr *tcp = (void*)ip + ip->ihl * 4;
    if ((void*)tcp + sizeof(*tcp) > data_end) return XDP_PASS;

    __u16 dport = __builtin_bswap16(tcp->dest);
    if (dport != 80) return XDP_PASS; // filter port 80

    __u64 zero = 0, *count;
    count = bpf_map_lookup_elem(&xdp_tcp_map, &dport);
    if (count) *count += 1;
    else bpf_map_update_elem(&xdp_tcp_map, &dport, &zero, BPF_ANY);

    return XDP_PASS;
}
```

### Socket-level BPF Program (accurate PID mapping):

```
#include <uapi/linux/ptrace.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <bcc/proto.h>

BPF_HASH(pid_tcp_map, u32, u64);

int count_pid_tcp(struct __sk_buff *skb) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
```

```

    u64 *count = pid_tcp_map.lookup(&pid);
    u64 one = 1;
    if (count) *count += 1;
    else pid_tcp_map.update(&pid, &one, BPF_ANY);
    return 0;
}

```

---

## Python Aggregation Example

```

from bcc import BPF
import time

# Load XDP
b_xdp = BPF(src_file="xdp_tcp_count.c", cflags=["-O2"])
fn_xdp = b_xdp.load_func("xdp_tcp_filter", BPF.XDP)
b_xdp.attach_xdp("eth0", fn_xdp, 0)

# Load socket-level BPF
b_sock = BPF(src_file="socket_tcp_count.c")
b_sock.attach_kprobe(event="__netif_receive_skb", fn_name="count_pid_tcp")

print("Hybrid monitoring started. Ctrl-C to stop.")
try:
    while True:
        time.sleep(5)
        print("\nPort 80 packets (XDP map):")
        xdp_map = b_xdp.get_table("xdp_tcp_map")
        for port, count in xdp_map.items():
            print(f"Port {port.value}: {count.value} packets")

        print("\nPer-PID TCP packets (socket map):")
        pid_map = b_sock.get_table("pid_tcp_map")
        for pid, count in pid_map.items():
            print(f"PID {pid.value}: {count.value} packets")

except KeyboardInterrupt:
    b_xdp.remove_xdp("eth0", 0)
    print("Stopped monitoring.")

```

---

## Advantages of This Hybrid Approach

- High performance:** XDP filters massive packet streams before they hit the kernel.
- Accuracy:** Socket-level BPF maps packets to the correct process.
- Flexible aggregation:** User-space Python combines both fast counts and per-process data.
- Production-ready pattern:** This is essentially how modern CNI plugins and network monitoring tools work.

If you want, I can create a **ready-to-run single Python script + eBPF C files** that implements this hybrid approach fully, so you could literally run it on a Linux system with minimal setup.

Do you want me to do that?