XDP

Linos Giannopoulos

March 28, 2019

XDP in practice:
integrating XDP into our DDoS mitigation pipeline

**Gilberto Bertin**
Cloudflare Ltd.
London, UK
gilberto@cloudflare.com

Abstract

To absorb large DDoS (distributed denial of service) attacks, the Cloudflare DDoS mitigation team has developed a solution based on kernel bypass and classic BPF. This allows us to filter network packets in userspace, skipping the usual packet processing done by Netfilter and the Linux network stack. This approach has solved performance issues that were experienced whilst handling large packet floods using solely the vanilla Linux kernel features.

In this paper we will first introduce our current architecture and then discuss a proposed solution based on XDP and eBPF. We will explain how XDP can be used in our infrastructure and which parts of our system need to be rewritten and adapted to make use of it. We will then conclude with the issues we have experienced so far with XDP.
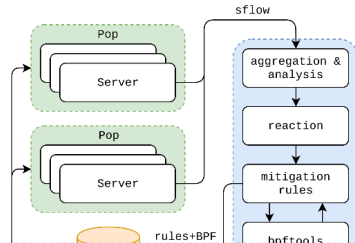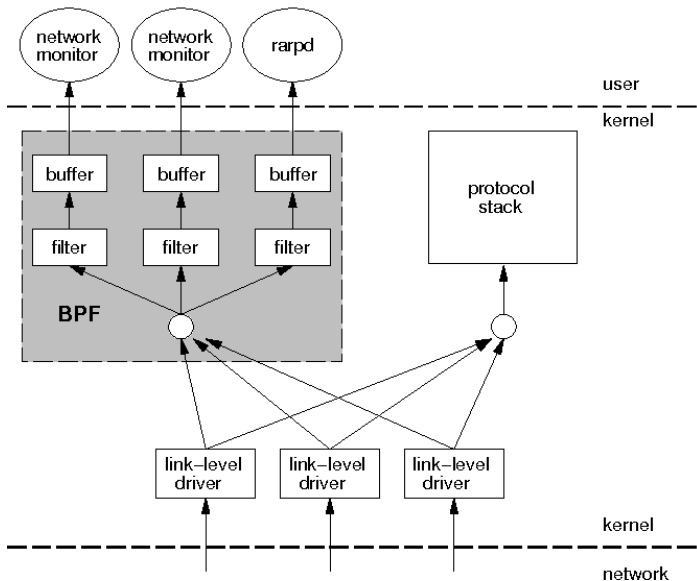
- cBPF
- eBPF
- XDP
- DDoS Mitigation Pipeline

cBPF

Berkeley Packet Filter

- *The BSD Packet Filter: A New Architecture for User-level Packet Capture*
- Avoid needless copying of packets
- Register-based VM in-kernel
- Filter packets as early as possible

eBPF

eBPF

What changed

- Extended ISA for modern CPUs
- More registers (from 2 to 10)
- JIT compiler bytecode to native code

New concepts

- eBPF Maps shared between user and kernel space
- eBPF Verifier

- Hash
- Array
- Tail Call Array
- Per-CPU Hash/Array
- Stack Trace

- LRU (per-CPU) Hash
- Longest-Prefix Matching Trie
- Array/Hash of Maps
- Net device Map
- Socket Map
- ...

eBPF Verifier

We essentially run userspace programs in the kernel.

How can we ensure an eBPF program won't cause a kernel panic or read/write arbitrary kernel memory?

eBPF Verifier

We essentially run userspace programs in the kernel.

How can we ensure an eBPF program won't cause a kernel panic or read/write arbitrary kernel memory?

Restrictions:
- 4096 instructions per program
- No loops
- No unreachable instructions
- Cannot read uninitialized registers
- Cannot read any arbitrary memory
- No out-of-bounds jumps
- Everything needs to be inlined (no function calls or shared library calls)
- Only calls to BPF Helpers (BPF to BPF is also allowed on newer kernel versions)

(some) eBPF Helper Functions

uapi/linux/bpf.h

```
#define __BPF_FUNC_MAPPER(FN)   \
  FN(map_lookup_elem),    \
  FN(map_update_elem),    \
  FN(map_delete_elem),    \
  ...
  FN(ktime_get_ns),   \
  FN(trace_printk),   \
  FN(get_prandom_u32),    \
  FN(skb_store_bytes),    \
  FN(l3_csum_replace),    \
  FN(tail_call),      \
  FN(xdp_adjust_head),    \
  FN(xdp_adjust_meta),    \
  ...
```
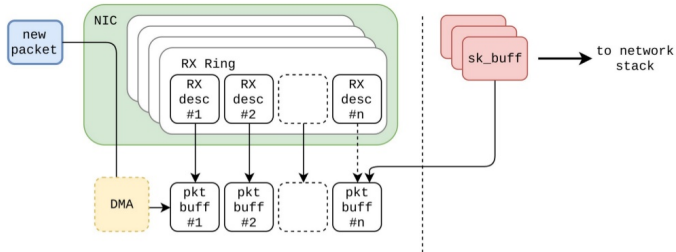
Some use cases:

- Tracing (kprobes/uprobes/tracepoints)[1]
- Networking (tc/XDP)
- Security (seccomp/IDS/DDoS)

---

[1]https://www.youtube.com/watch?v=w8nFRoFJ6EQ

XDP

NIC and kernel packet buffers

eXpress Data Path

- High-performance packet processing
- Runs eBPF programs at the earliest possible point (before sk_buffer allocations)
- In concert with the kernel (no userspace bypass or out-of-tree kernel modifications)
- Driver dependant
- Verdicts: XDP_DROP, XDP_PASS, XDP_TX, XDP_ABORTED, XDP_REDIRECT

Use cases:

- Firewalling (Cillium[1])
- DDoS
- Load balancing (Katran[2])
- Monitoring

---

[1] https://cilium.io/
[2] https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/
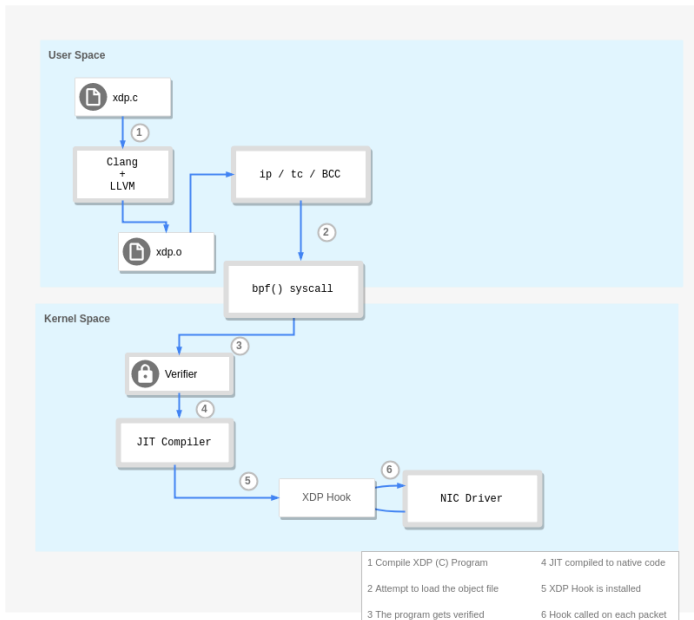
```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#define SEC(NAME) __attribute__((section(NAME), used))

SEC("dropper")
int xdp(struct xdp_md *ctx)
{
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    struct iphdr *ip = data + sizeof(struct ethhdr);

    // +1 is sizeof(struct iphdr)
    if (ip + 1 > data_end)
        return XDP_ABORTED;

    if(ip->saddr == 59017107){ // Binary format of "147.135.132.3"
        return XDP_DROP;
    }
    return XDP_PASS;
}
```

What about IPtables?

Test bench

- 10GbE Intel NIC
- Generate 14Mpps tiny UDP packets (pktgen)
- Randomized source IP/Port
- Traffic is steered towards a single CPU
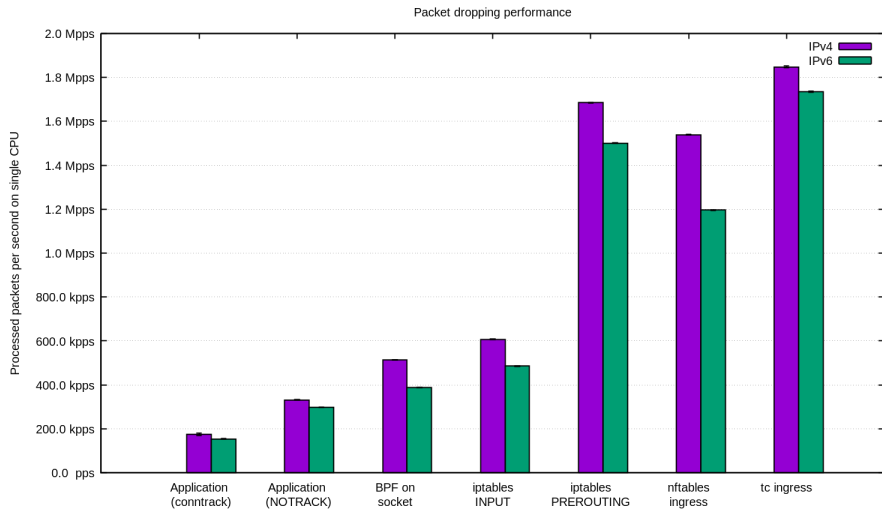- Measure number of packets handled by the kernel on that CPU

Test bench

- 10GbE Intel NIC
- Generate 14Mpps tiny UDP packets (pktgen)
- Randomized source IP/Port
- Traffic is steered towards a single CPU
- Measure number of packets handled by the kernel on that CPU

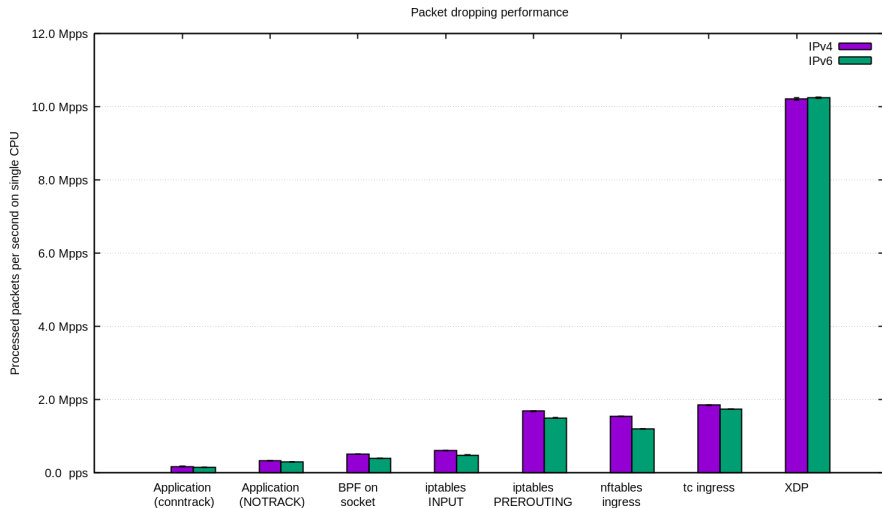*How fast can we drop packets?*

Packet dropping performance
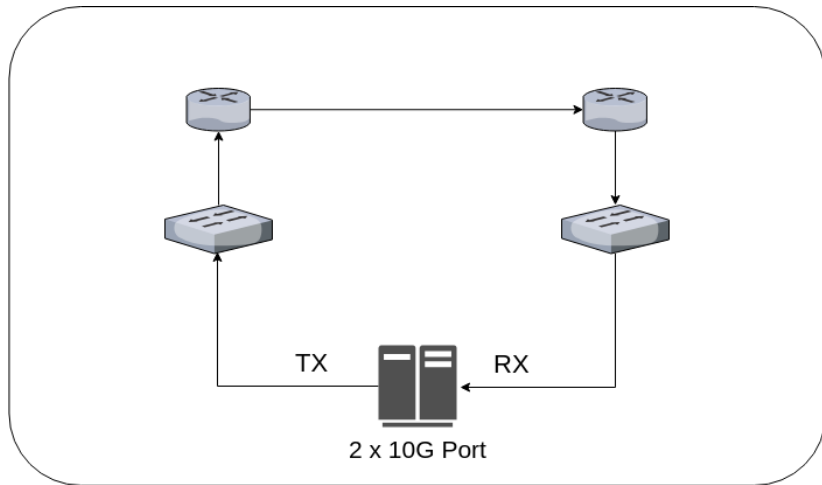
Packet dropping performance
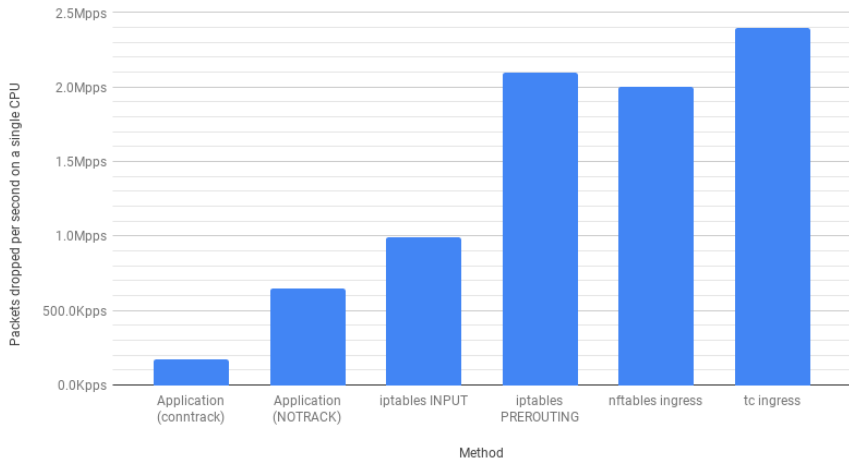
Packet dropping performance

Packet dropping performance (XDP included)
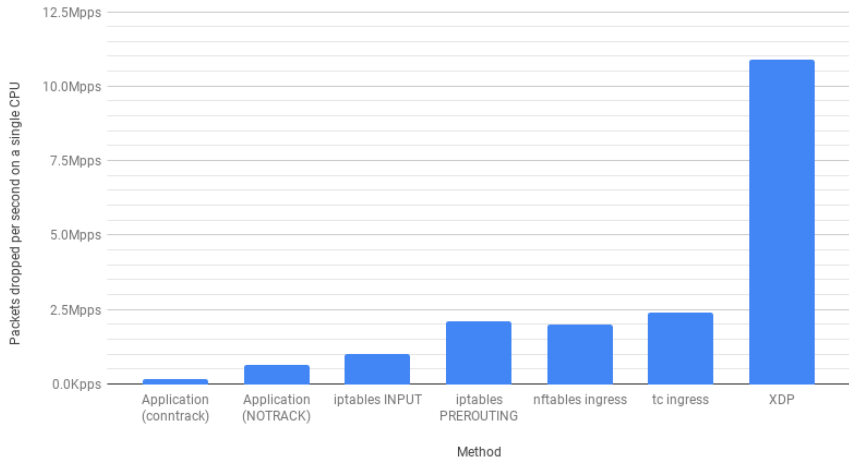
DDoS Pipeline

- Traffic sampling
- Traffic aggregation and analysis
- Attack reaction

- DDoS Attacks that our uplink can handle
- If that's not the case:
    - Anycast with different PoPs
    - Flowspec
- Flowspec has its limitations
- Software and not hardware limitations

In contrast with an IDS, sampling instead of working on the whole traffic is crucial here.

- Use *NFLOG (+Statistic)*
- Userspace daemon to process packet

In contrast with an IDS, sampling instead of working on the whole traffic is crucial here.

- Use *NFLOG (+Statistic)*
- Userspace daemon to process packet

Issue: no visibility in the dropped traffic

Sample before drop rules - in XDP

- Copy sampled packet to perf event buffer (xdp_event_output)
- Userspace daemon to obtain the packet

How do we categorize traffic?

How do we categorize traffic?

Credit: *https://www.slideshare.net/InfoQ/xdp-in-practice-ddos-mitigation-cloudflare*

Traffic aggregation and analysis
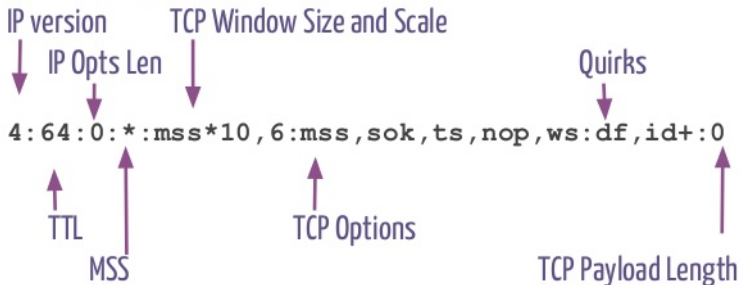
Traffic aggregated into groups

- TCP SYNs, TCP ACKs, UDP/DNS
- Destination IP/Port
- Known attack vectors and other heuristics

```
$ ./p0f2ebpf.py --ip 1.2.3.4 --port 1234
    '4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0'

static inline int match_p0f(void *data, void *data_end) {
    struct ethhdr *eth_hdr;
    struct iphdr *ip_hdr;
    struct tcphdr *tcp_hdr;
    u8 *tcp_opts;

    eth_hdr = (struct ethhdr *)data;
    if (eth_hdr + 1 > (struct ethhdr *)data_end)
        return XDP_ABORTED;
    if_not (eth_hdr->h_proto == htons(ETH_P_IP))
        return XDP_PASS;
```

```
ip_hdr = (struct iphdr *)(eth_hdr + 1);
if (ip_hdr + 1 > (struct iphdr *)data_end)
    return XDP_ABORTED;
if_not (ip_hdr->daddr == htonl(0x1020304))
    return XDP_PASS;
if_not (ip_hdr->version == 4)
    return XDP_PASS;
if_not (ip_hdr->ttl <= 64)
    return XDP_PASS;
if_not (ip_hdr->ttl > 29)
    return XDP_PASS;
if_not (ip_hdr->ihl == 5)
    return XDP_PASS;
if_not ((ip_hdr->frag_off & IP_DF) != 0)
    return XDP_PASS;
if_not ((ip_hdr->frag_off & IP_MBZ) == 0)
    return XDP_PASS;
```

```
tcp_hdr = (struct tcphdr*)((u8 *)ip_hdr +
ip_hdr->ihl * 4);
if (tcp_hdr + 1 > (struct tcphdr *)data_end)
    return XDP_ABORTED;
if_not (tcp_hdr->dest == htons(1234))
    return XDP_PASS;
if_not (tcp_hdr->doff == 10)
    return XDP_PASS;
if_not ((htons(ip_hdr->tot_len) - (ip_hdr->ihl * 4) - (tcp_hdr->dof
    return XDP_PASS;

...

return XDP_DROP;
}
```
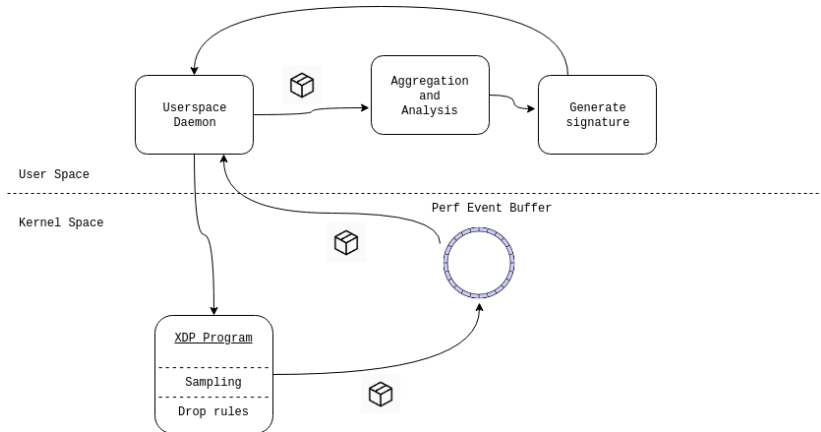
- Generate 8.7Mpps[1] TCP SYN packets (trafgen)
- Randomized source IP/Port
- Traffic is steered towards a single CPU
- Measure number of packets handled by the kernel on that CPU

---
[1] We can do better!

Methodology:

- Sample traffic
- Feed samples to p0f
- Grab signature
- Generate XDP program to drop the malicious traffic
- Attach it to XDP hook

```
Every 2.0s: sudo ethtool -S enp4s0f0|grep tx_packets          2019-03-27 12:26:06.834351

    tx_packets:        8.6m/s
```

```
Every 2.0s: sudo ethtool -S enp4s0f1|grep rx_packets          2019-03-27 12:26:05.811454

    rx_packets:        8.6m/s
```

```
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
Enabled kernel qdisc bypass
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
TX,V2: 0.50 MiB, 256 Frames, each 2048 Byte allocated
```

```
Every 2.0s: sudo ethtool -S enp4s0f1|grep no_dma          2019-03-27 12:26:06.303452

    rx_no_dma_resources:        71.7k/s
```

Questions?