Exploring eBPF and XDP

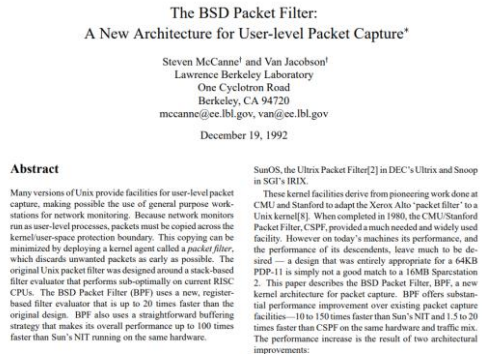
Empowering Linux Programmability and Beyond

BO CUI

Agenda

- From cBPF to eBPF
- eBPF Overview (verifier/JIT/map/helper)
- BPF Tools (BCC/bpftrace)
- Libbpf and BPF CO-RE
- NGINX BPF use case
- XDP/AF_XDP

BPF History



The BSD Packet Filter:
A New Architecture for User-level Packet Capture*

Steven McCanne[1] and Van Jacobson[1]
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, van@ee.lbl.gov

December 19, 1992

Abstract

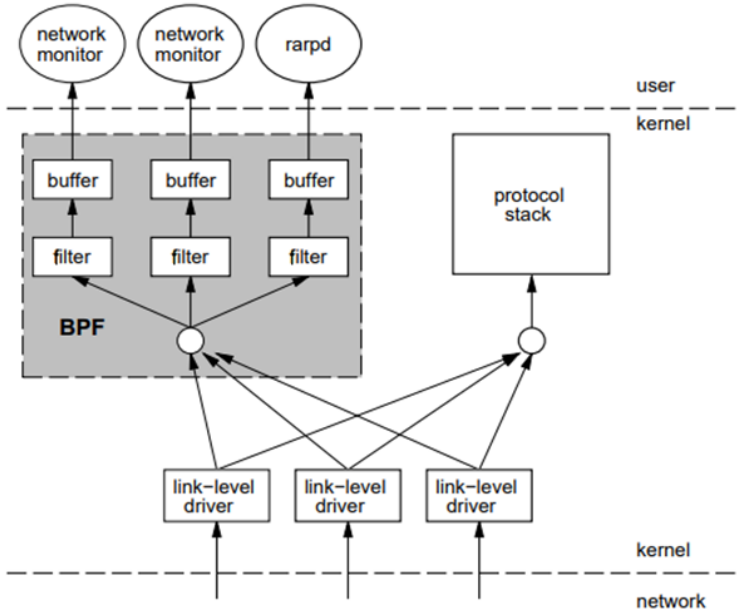
Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

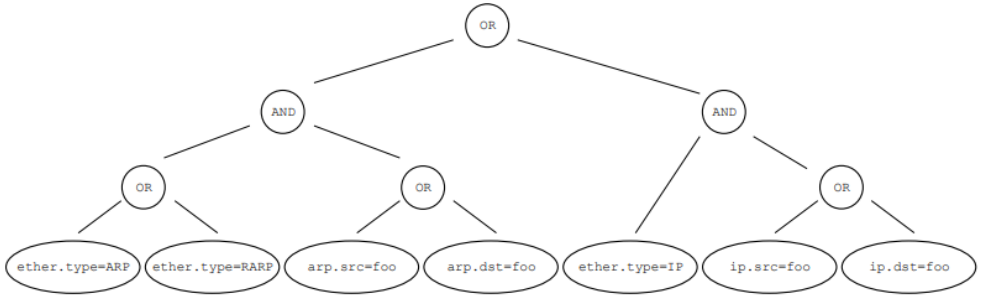
SunOS, the Ultrix Packet Filter[2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and the performance of its descendents, leave much to be desired — a design that was entirely appropriate for a 64KB PDP-11 is simply not a good match to a 16MB Sparcstation 2. This paper describes the BSD Packet Filter, BPF, a new kernel architecture for packet capture. BPF offers substantial performance improvement over existing packet capture facilities—10 to 150 times faster than Sun's NIT and 1.5 to 20 times faster than CSPF on the same hardware and traffic mix. The performance increase is the result of two architectural improvements:

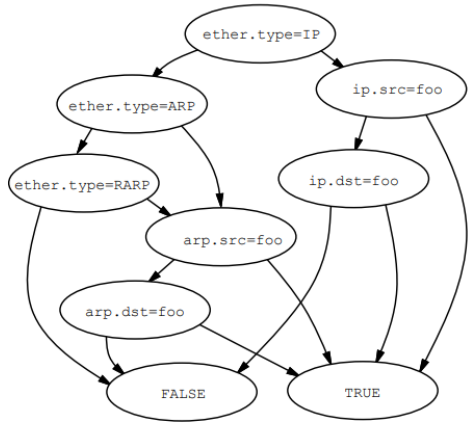
For high performance user space network monitors, a kernel agent (packet filter) is designed to discard unwanted packets as early as possible for minimal memory copy cost.

BPF introduced a register-based virtual machine to describe filtering actions





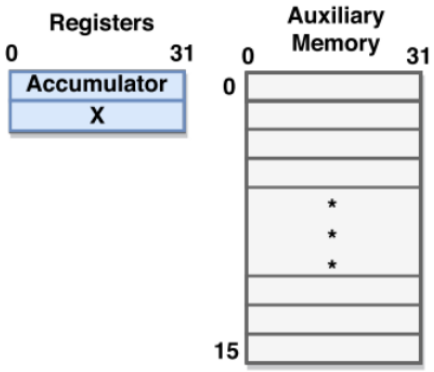
Expression Tree Model: seven comparison predicates and six boolean operations are required to traverse the entire tree



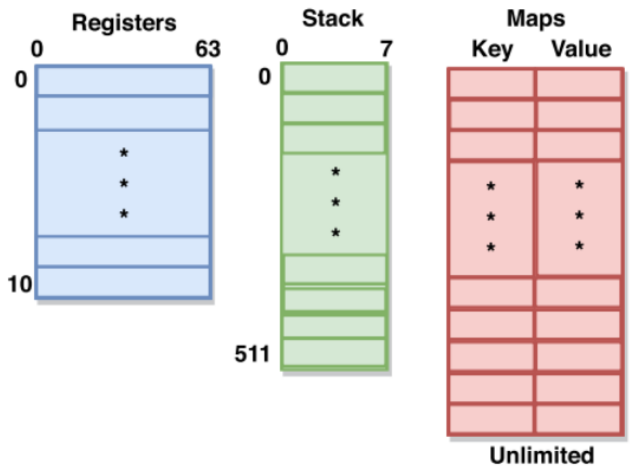
Config Flow Graph: up to five comparison operations, and the average number of comparisons is three.

BPF Virtual Machine

```
# tcpdump -d host 127.0.0.1 and port 80
(000) ldh [12]
(001) jeq #0x800 jt 2 jf 18
(002) ld [26]
(003) jeq #0x7f000001 jt 6 jf 4
(004) ld [30]
(005) jeq #0x7f000001 jt 6 jf 18
(006) ldb [23]
(007) jeq #0x84 jt 10 jf 8
(008) jeq #0x6 jt 10 jf 9
(009) jeq #0x11 jt 10 jf 18
(010) ldh [20]
(011) jset #0x1fff jt 18 jf 12
(012) ldxb 4*([14]&0xf)
(013) ldh [x + 14]
(014) jeq #0x50 jt 17 jf 15
(015) ldh [x + 16]
(016) jeq #0x50 jt 17 jf 18
(017) ret #262144
(018) ret #0
```



Classic BPF machine



Extended BPF machine

	Classic BPF	Extended BPF
Word size	32b	64b
Registers	2	10+1
Storage	16 slots	512B stack + maps
Events	Packets	Many event sources

eBPF to Mainstream

CBPF-> eBPF conversion (ISA/interpreter/verifier/JIT, Mar. 2014)

eBPF backend merged into upstream LLVM3.7 (Aug. 2015)

Linux 4.1 BPF Kprobe support added (Api. 2015)

BCC project created shortly after LLVM lands (May. 2015)

Network TC BPF cls_bpf enabled (Mar. 2015)

eXpress DataPath merged in Linux 4.8 (Jul. 2016)

Linux 4.9 Release (Dec' 2016)

Linux 5.4 LTS with fully BPF enabled kernel (Nov, 2019)

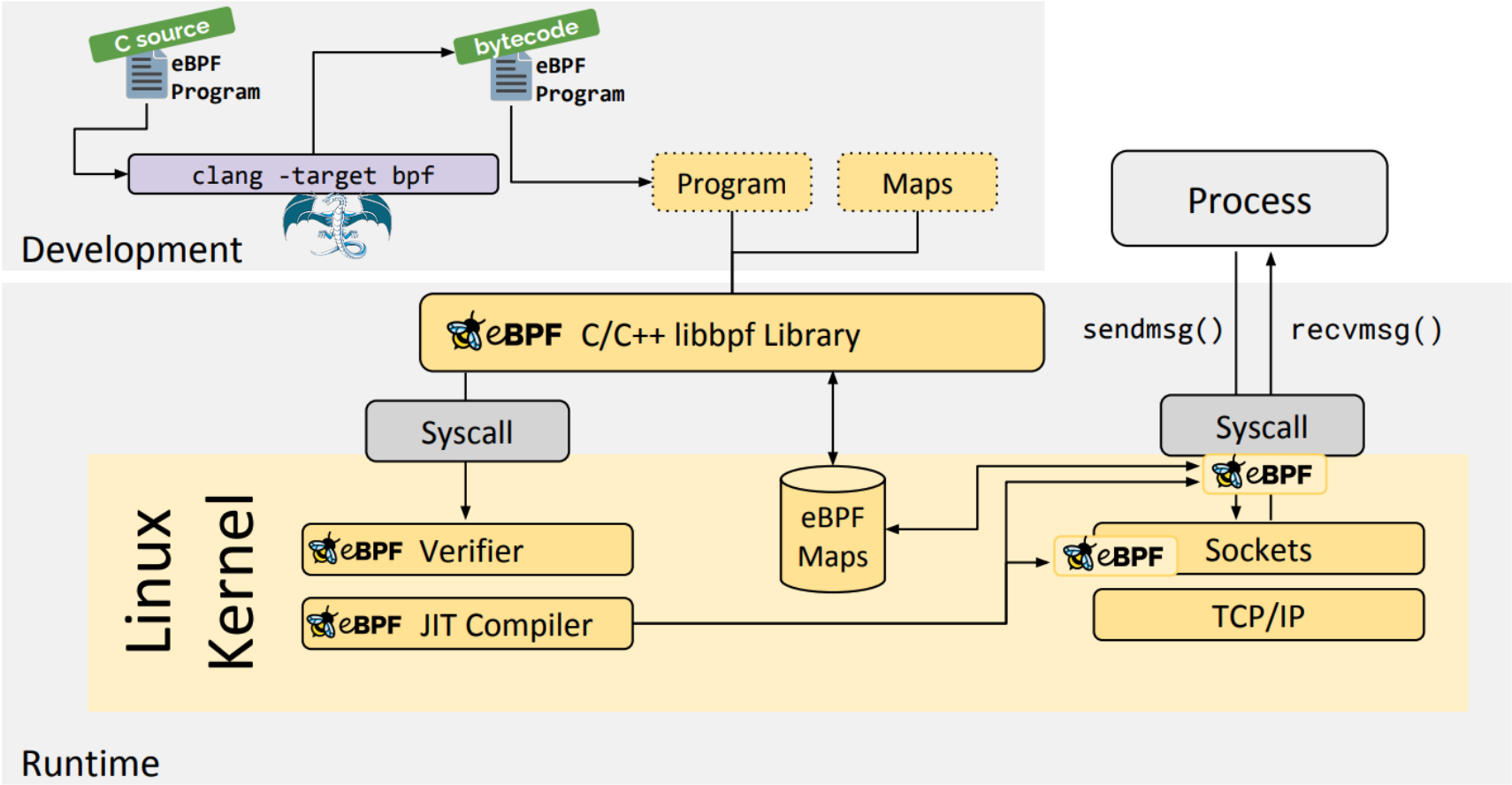
	Execution model	User defined	Compil-ation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers

"BPF: A New Type of Software", Brendan Gregg

BPF Overview

BCC: Bpf tools for for kernel tracing

Bpftrace: High-level tracing language
Bpftool: kernel utility for inspecting of BPF programs and maps
libbpf + CO-RE: compile once run everywhere



Verifier & JIT

To ensure the integrity and security of the operating system, the kernel uses a verifier that performs static program analysis of eBPF instructions being loaded into the system. (/bpf/verifier.c)

- Program size (1m)

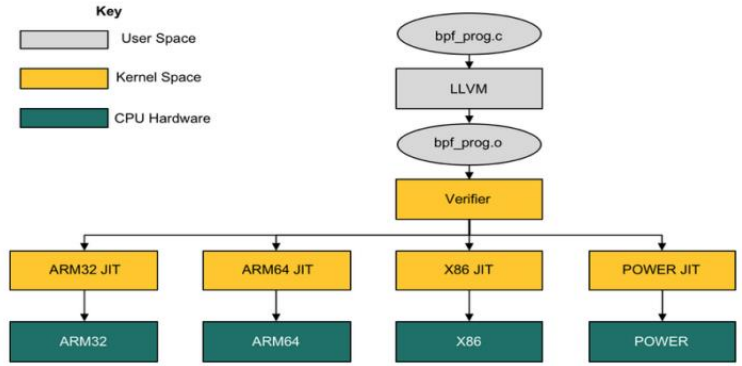
- Control flow DAG build and analysis (ensure no loop/unsupported/unreachable instructions, safe memory access, help argument etc)

- Analyzes every instruction, maintaining per-register state, and making sure no invalid operations are performed.

- Rewrite parts of program while still semantically equivalent
  - External access: Map->actual map pointer/BTF fd + id -> kernel symbol address
  - Safety: Division-by-zero/zero-extension pseudo-instructions/
  - Performance optimization: inline BPF helper/indirect -> direct call

- Evolving with new features (e.g. bounded loop)

Just-in-time compiler translates BPF code directly into the host system's assembly code



Program/Attach Type

```
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,
    BPF_PROG_TYPE_SOCKET_FILTER,
    BPF_PROG_TYPE_KPROBE,
    BPF_PROG_TYPE_SCHED_CLS,
    BPF_PROG_TYPE_SCHED_ACT,
    BPF_PROG_TYPE_TRACEPOINT,
    BPF_PROG_TYPE_XDP,
    BPF_PROG_TYPE_PERF_EVENT,
    BPF_PROG_TYPE_CGROUP_SKB,
    BPF_PROG_TYPE_CGROUP_SOCK,
    BPF_PROG_TYPE_LWT_IN,
    BPF_PROG_TYPE_LWT_OUT,
    BPF_PROG_TYPE_LWT_XMIT,
    BPF_PROG_TYPE_SOCK_OPS,
    … … …
```

eBPF program type determines the kernel hook, input context, and available helper functions.

bpf_prog_load(filename, BPF_PROG_TYPE_SOCKET_FILTER, &obj, &prog_fd) / bpf(BPF_PROG_LOAD, &attr, sizeof(attr))

setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd) / bpf(BPF_PROG_ATTACH, &attr, sizeof(attr))

net/core/sock.c

```
int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
{
        int err;

        err = sk_filter(sk, skb);
        if (err)
                return err;

        return __sock_queue_rcv_skb(sk, skb);
}
EXPORT_SYMBOL(sock_queue_rcv_skb);
```

Kernel hook

sample/bpf/sockex1_kern.c

```
SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
        int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
        long *value;

        if (skb->pkt_type != PACKET_OUTGOING)
                return 0;

        value = bpf_map_lookup_elem(&my_map, &index);
        if (value)
                __sync_fetch_and_add(value, skb->len);

        return 0;
}
```

context

helper

Include/uapi/linux.bpf.h

```
/* user accessible mirror of in-kernel sk_buff.
 * new fields can only be added to the end of this structure
 */
struct __sk_buff {
        __u32 len;
        __u32 pkt_type;
        __u32 mark;
        __u32 queue_mapping;
        __u32 protocol;
        __u32 vlan_present;
        __u32 vlan_tci;
        __u32 vlan_proto;
        __u32 priority;
        __u32 ingress_ifindex;
        __u32 ifindex;
        __u32 tc_index;
        __u32 cb[5];
        __u32 hash;
        __u32 tc_classid;
        __u32 data;
        __u32 data_end;
        __u32 napi_id;
```

Map

A generic key-value stores available to eBPF programs. Keys and values are treated as binary blobs, allowing the storage of user-defined data structures and types.

Created by user-space process and can be shared by processes and ebpf programs

The map types include generic hash maps, arrays and radix trees, as well as specialised types containing pointers to eBPF programs (used for tail calls), or redirect targets, or even pointers to other maps.

Refcnt maintained for eBPF object (programs, maps, and debug info) life cycle

pin a map (or any other eBPF object) to /sys/fs/bpf/ to keep a shared map alive

bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr)).

```
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, u32);
        __type(value, long);
        __uint(max_entries, 256);
} my_map SEC(".maps");
```

Helper / Kfuncs

limited callable helper functions within one program type; Offered by the kernel infrastructure to enable interaction with the context and other kernel facilities/structures (maps/routing table/tunnel mechanism etc)

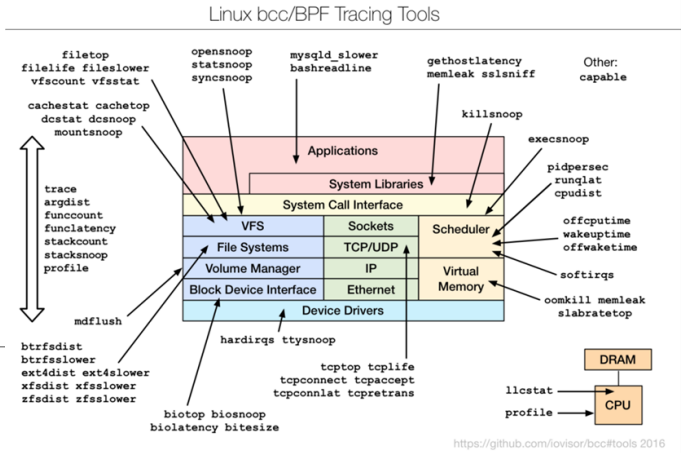
Iterating together with new kernels to provide new kernel handlers, which are stable interface /UAPI.

bpf_get_prandom_u32: returns a 32-bit pseudo-random value
bpf_ktime_get_ns: returns time since system boot, in nanoseconds;
bpf_skb_vlan_pop, bpf_skb_vlan_push: remove/add, respectively, VLAN tags from a packet;
bpf_tail_call: jump into another BPF program retrieved from a BPF_MAP_TYPE_PROG_ARRAY

Kfuncs is the option to access internal kernel functions registered with BPF subsystem. (aggregated into one or more BTF kfunc-sets , register_btf_kfunc_id_set with specified eBPF program type), no compatibility guarantees between kernels.

Program Type	BPF_PROG_TYPE_SOCKET_FILTER	BPF_PROG_TYPE_SOCK_OPS	BPF_PROG_TYPE_KPROBE
Helper Functions	BPF_FUNC_skb_load_bytes() BPF_FUNC_skb_load_bytes_relative() BPF_FUNC_get_socket_cookie() BPF_FUNC_get_socket_uid() BPF_FUNC_perf_event_output() Base functions	BPF_FUNC_setsockopt() BPF_FUNC_getsockopt() BPF_FUNC_sock_ops_cb_flags_set() BPF_FUNC_sock_map_update() BPF_FUNC_sock_hash_update() BPF_FUNC_get_socket_cookie() Base functions	BPF_FUNC_perf_event_output() BPF_FUNC_get_stackid() BPF_FUNC_get_stack() BPF_FUNC_perf_event_read_value() BPF_FUNC_override_return() Tracing functions

BCC - BPF Compiler Collection



Bpf program

hash map

/sys/kernel/debug/tracing/trace_pipe <--

kprobe

```
from __future__ import print_function
from bcc import BPF
from bcc.utils import printb

REQ_WRITE = 1           # from include/linux/blk_types.h

# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blk-mq.h>

BPF_HASH(start, struct request *);

void trace_start(struct pt_regs *ctx, struct request *req) {
        // stash start timestamp by request ptr
        u64 ts = bpf_ktime_get_ns();

        start.update(&req, &ts);
}

void trace_completion(struct pt_regs *ctx, struct request *req) {
        u64 *tsp, delta;

        tsp = start.lookup(&req);
        if (tsp != 0) {
                delta = bpf_ktime_get_ns() - *tsp;
                bpf_trace_printk("%d %x %d\\n", req->__data_len,
                    req->cmd_flags, delta / 1000);
                start.delete(&req);
        }
}
""")

if BPF.get_kprobe_functions(b'blk_start_request'):
        b.attach_kprobe(event="blk_start_request", fn_name="trace_start")
b.attach_kprobe(event="blk_mq_start_request", fn_name="trace_start")
if BPF.get_kprobe_functions(b'__blk_account_io_done'):
    b.attach_kprobe(event="__blk_account_io_done", fn_name="trace_completion")
else:
    b.attach_kprobe(event="blk_account_io_done", fn_name="trace_completion")

# header
print("%-18s %-2s %-7s %8s" % ("TIME(s)", "T", "BYTES", "LAT(ms)"))
```

A set of tools and libraries built on top of BPF that simplifies the process of writing, loading, and using BPF programs.

High-level languages (Python, Lua, and others) and APIs that abstract away some of the complexities of working directly with BPF. Easy to use

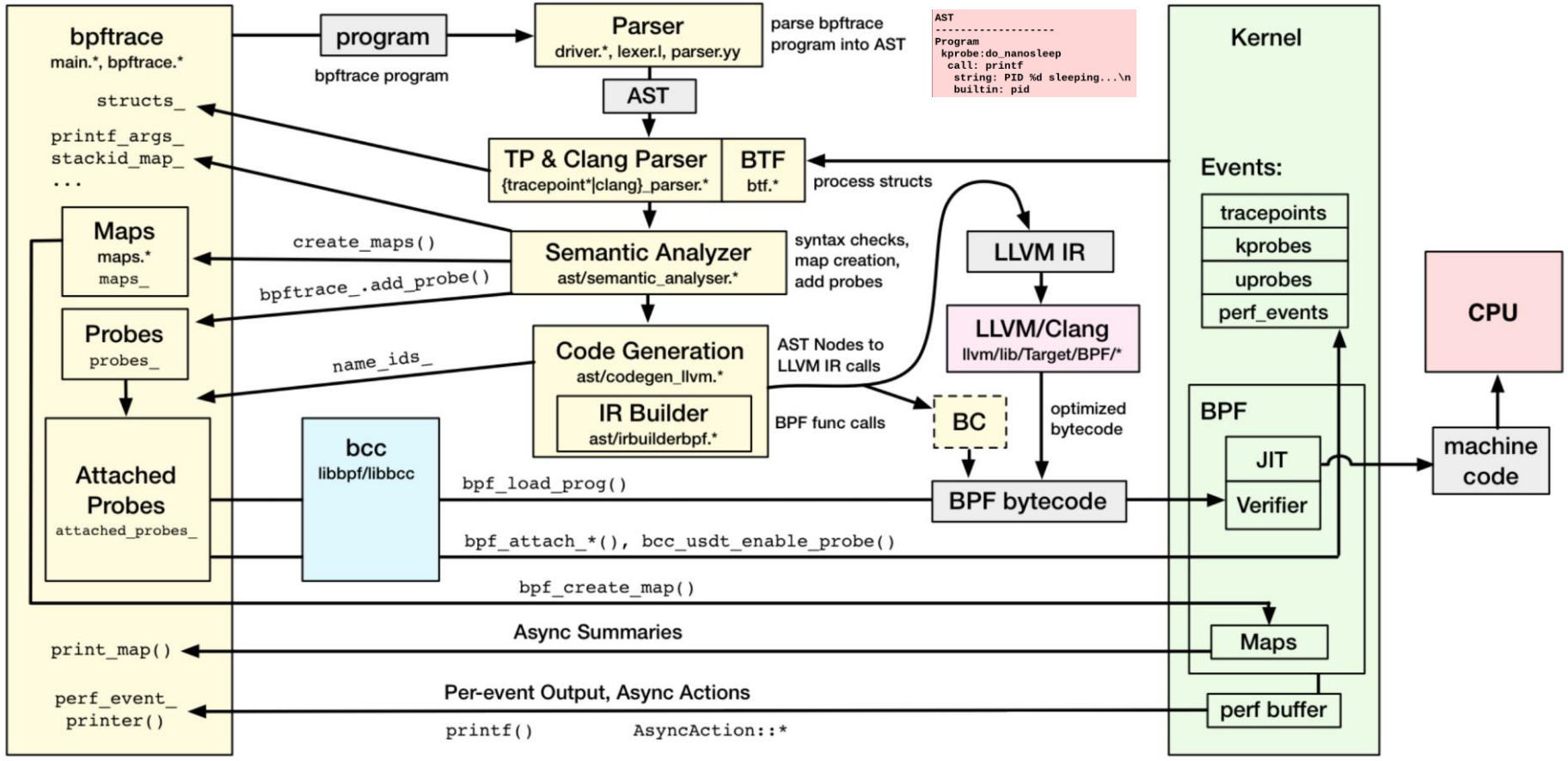
End-to-end BPF workflow in a shared library

- A modified C language for BPF backends
- Integration with llvm-bpf backend for JIT
- Dynamic (un)loading of JITed programs
- Support for BPF kernel hooks: socket filters, tc classifiers, tc actions and kprobes

bpftrace

```
# bpftrace -e 'kretprobe:vfs_read { @bytes = lhist(retval, 0, 1000, 200); }'
Attaching 1 probe...
^C

@bytes:
(..., 0)               7 |                                                    |
[0, 200)            1389 |@@@@@@@@@@@@@@@@@@@@                                |
[200, 400)           246 |@@@                                                 |
[400, 600)            42 |                                                    |
[600, 800)            43 |                                                    |
[800, 1000)           43 |                                                    |
[1000, ...)         3508 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

Another front end to craft bpf tools, high-level tracing language inspired by awk and C, Dtrace/SystemTap.
Low effort for new tool hacking (One-Liners)



Libbpf + CO-RE (Compile Once – Run Everywhere)

BPF portability for widely deployed production: verify and work correctly across different kernel versions!

- kernel types and data structures are in constant flux. (filed shuffle/rename/remove); kernel version/configuration vary
- BCC relies on runtime compilation and brings the entire huge LLVM/Clang library in (fat binary distribution/heavy building overhead/kernel header dependency/compilation err till runtime)

BPF CO-RE support of pre-compiled BPF program depends on:

- Kernel BTF (BPF Type Format): provide all data type information (CONFIG_DEBUG_INFO_BTF=y)

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

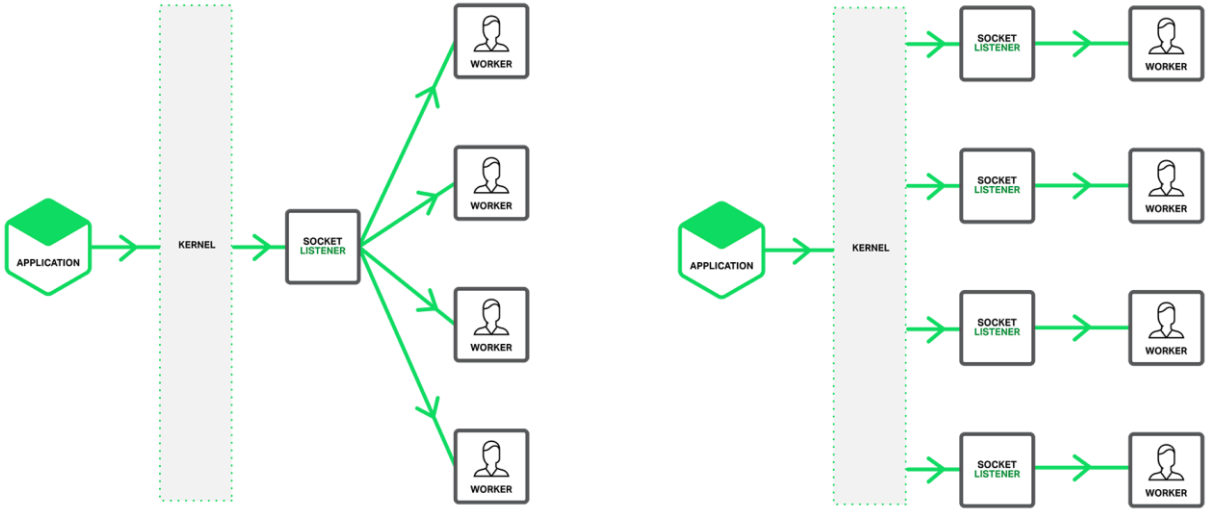
-Compiler (Clang): captures a high-level description of BPF program code read intention(BTF relocations of field offset/existence/size)

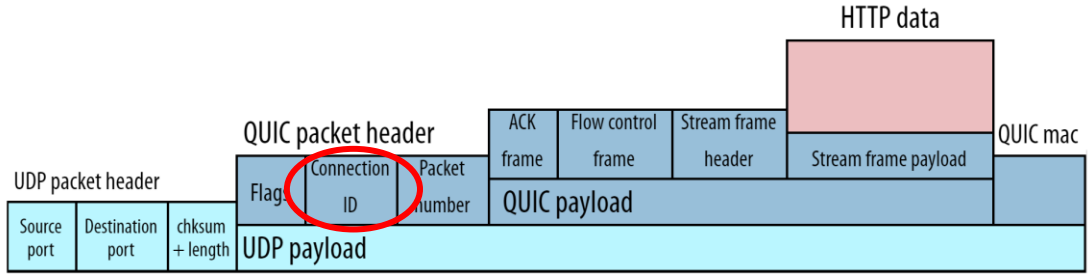
- BPF loader (libbpf): ties BTFs from kernel and BPF program together to adjust compiled BPF code to specific kernel on target hosts; tailor BPF program code to a particular running kernel, resolves and matches all the types and fields, update offset/relocatable data
- Kernel: staying completely BPF CO-RE-agnostic

Many Linux distributions enable kernel BTF by default, 25+ BCC tools got converted to libbpf + BPF CO-RE

libbpf-bootstrap project created to simplify starting BPF development with libbpf and BPF CO-RE

Example - NGINX Socket Shading





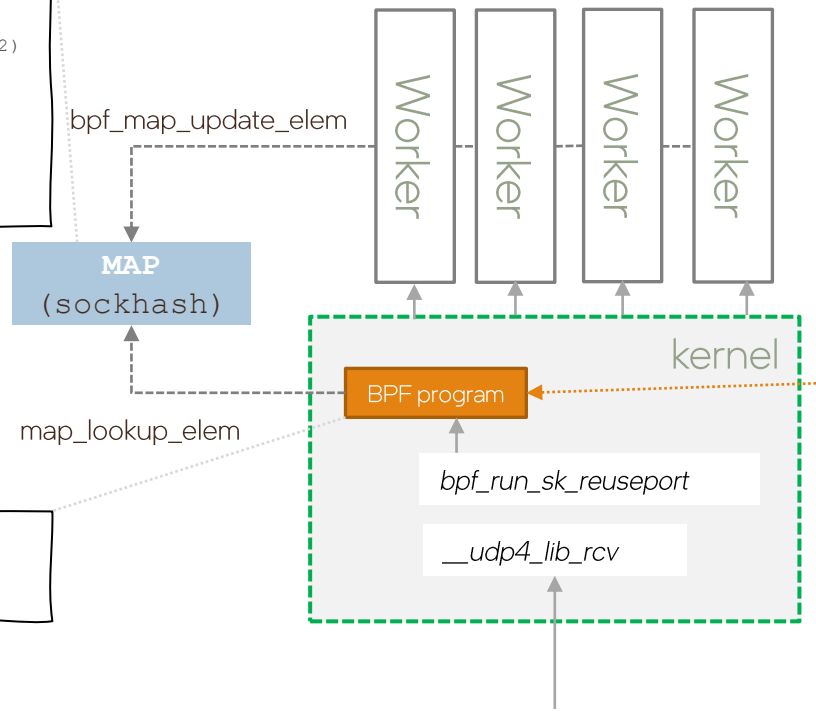
SO_REUSEPORT allows multiple sockets to listen on the same IP address and port combination so kernel can load balances incoming connections across the sockets.

For QUIC packet identified by Connection ID(as part of UDP payload), quic kernel bpf helper inspects payload for DCID then routes the packet into socket owned by specified worker.

- ngx_quic_create_server_id (dcid = rand | so_cookie)
- ngx_quic_bpf_group_add_socket (map[cookie] = socket)

Example - NGINX QUIC select_socket

```
# bpftool map list
95: sockhash  flags 0x0
        key 8B  value 8B  max_entries 16  memlock 4096B
        pids nginx(6248), nginx(6249), nginx(6250), nginx(6251), nginx(6252)

# bpftool map dump id 95
key: 4a 20 00 00 00 00 00 00  value: 4a 20 00 00 00 00 00 00
key: 49 20 00 00 00 00 00 00  value: 49 20 00 00 00 00 00 00
key: 48 20 00 00 00 00 00 00  value: 48 20 00 00 00 00 00 00
key: 47 20 00 00 00 00 00 00  value: 47 20 00 00 00 00 00 00
Found 4 elements
```




```
# bpftool prog list
169: sk_reuseport  tag d12d7ab636954a1d
        loaded_at 2023-07-26T06:32:54-0400  uid 0
        xlated 496B  jited 304B  memlock 4096B  map_ids 95
```

```c
SEC(PROGNAME)
int ngx_quic_select_socket_by_dcid(struct sk_reuseport_md *ctx)
{
    int             rc;
    __u64           key;
    size_t          len, offset;
    unsigned char  *start, *end, *data, *dcid;

    start = ctx->data;
    end = (unsigned char *) ctx->data_end;
    offset = 0;

    advance_data(sizeof(struct udphdr)); /* data at UDP header */
    advance_data(1); /* data at QUIC flags */

    if (data[0] & NGX_QUIC_PKT_LONG) {

        advance_data(4); /* data at QUIC version */
        advance_data(1); /* data at DCID len */

        len = data[0];   /* read DCID length */

        if (len < 8) {
            /* it's useless to search for key in such short DCID */
            return SK_PASS;
        }

    } else {
        len = NGX_QUIC_SERVER_CID_LEN;
    }

    dcid = &data[1];
    advance_data(len); /* we expect the packet to have full DCID */

    /* make verifier happy */
    if (dcid + sizeof(__u64) > end) {
        goto failed;
    }

    key = ngx_quic_parse_uint64(dcid);

    rc = bpf_sk_select_reuseport(ctx, &ngx_quic_sockmap, &key, 0);

    switch (rc) {
    case 0:
        debugmsg("nginx quic socket selected by key 0x%llx", key);
        return SK_PASS;

    /* kernel returns positive error numbers, errno.h defines positive */
    case -ENOENT:
        debugmsg("nginx quic default route for key 0x%llx", key);
        /* let the default reuseport logic decide which socket to choose */
        return SK_PASS;

    default:
        debugmsg("nginx quic bpf_sk_select_reuseport err: %d key 0x%llx",
                 rc, key);
        goto failed;
    }

failed:
    /*
     * SK_DROP will generate ICMP, but we may want to process "invalid" packet
     * in userspace quic to investigate further and finally react properly
     * (maybe ignore, maybe send something in response or close connection)
     */
    return SK_PASS;
}
```

eXpress Data Path (XDP)

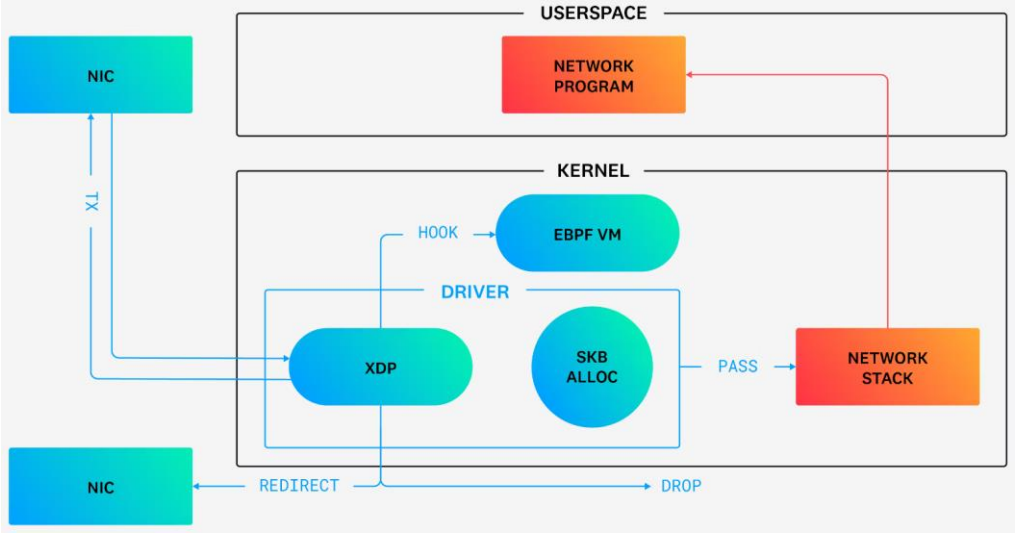
```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
struct xdp_md {
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        /* Below access go through struct xdp_rxq_info */
        __u32 ingress_ifindex; /* rxq->dev->ifindex */
        __u32 rx_queue_index;  /* rxq->queue_index  */
};
```

XDP program operates on the rx_ring buffer before the packet gets copied to an sk_buff(native mode):

- Attached to a specific NIC in a machine and can process incoming packets from that NIC only
- Visible only to incoming packets
- The incoming packet info are in structure xdp_md for packet process and manipulation in XDP program

3 XDP execution mode:

- Native mode: attached as a hook point in the NIC driver
- Offloaded: offload from the CPU to the NIC
- Generic mode: handled by kernel network stack netif_receive_skb(), slower performance.



by Daniel Lavie

Driver XDP Rx-processing(NAPI) loop

Code needed in driver for supporting XDP is straightforward

Impacting driver memory model to be compatible with XDP:

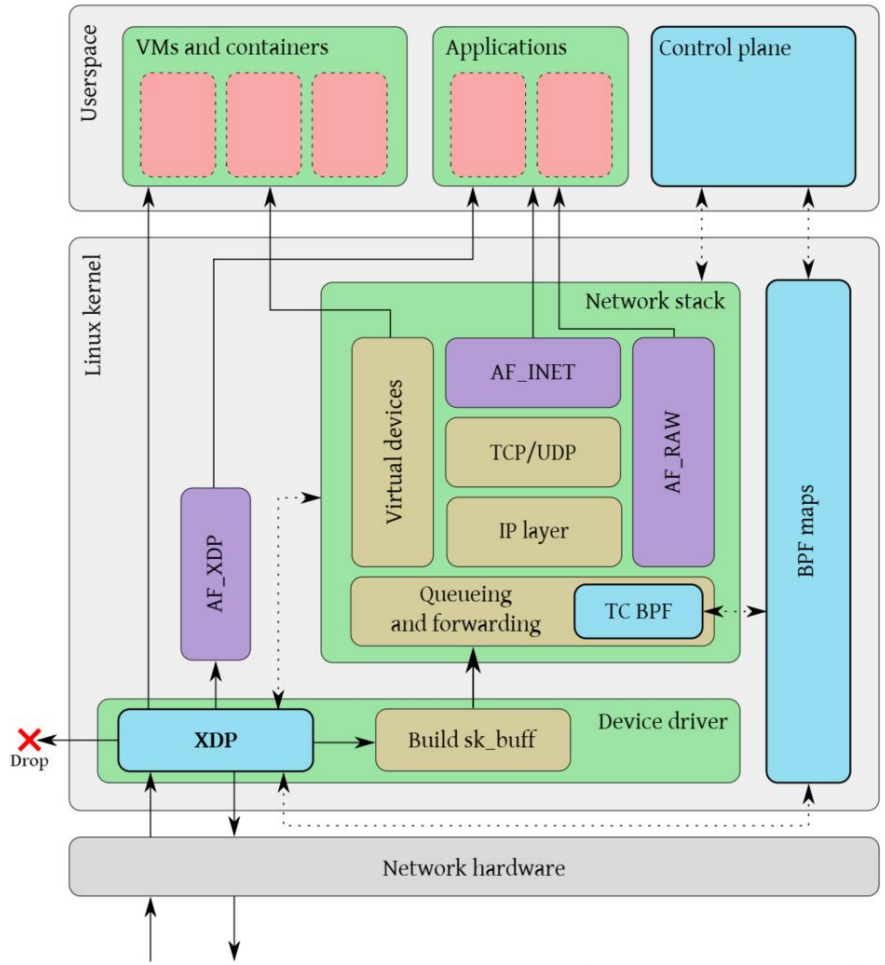
- XDP frame in physical contiguous memory
- headroom for xdp_frame, tailroom for skb_shared_info
- Register memory model within drivers
  xdp_rxq_info_reg_mem_model()

```
while (desc_in_rx_ring && budget_left--) {
  action = bpf_prog_run_xdp(xdp_prog, xdp_buff);
  /* helper bpf_redirect_map have set map (and index) via this_cpu_ptr */
  switch (action) {
    case XDP_PASS:     break;
    case XDP_TX:       res = driver_local_xmit_xdp_ring(adapter, xdp_buff); break;
    case XDP_REDIRECT: res = xdp_do_redirect(netdev, xdp_buff, xdp_prog); break;
    default:           bpf_warn_invalid_xdp_action(action); /* fall-through */
    case XDP_ABORTED:  trace_xdp_exception(netdev, xdp_prog, action); /* fall-through */
    case XDP_DROP:     page_pool_recycle_direct(pp, page); res = DRV_XDP_CONSUMED; break;
  } /* left out acting on res */
} /* End of NAPI-poll */
xdp_do_flush_map(); /* Bulk chosen by map, can store xdp_frame's for flushing */
driver_local_XDP_TX_flush();
```

XDP Kernel program

5 return codes/actions (pass/drop/tx/redirect/aborted)
Redirect target set through a helper function before exit:
- raw packet out on a different interface (devmap)
- a different CPU for further processing (cpumap)
- a special userspace socket address family (AF_XDP, xskmap); a Lock-free channel directly from driver RX-queue into an (AF_XDP) socket.

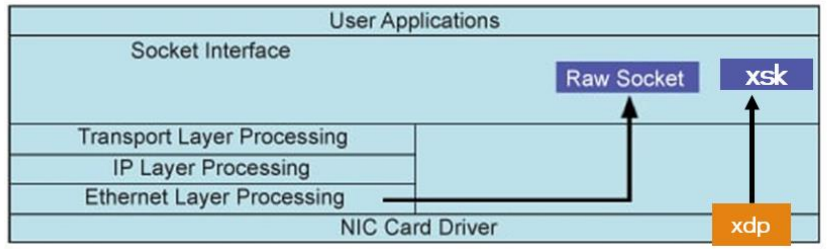
```
SEC("xdp")
int xdp_sock_prog(struct xdp_md *ctx)
{
    int index = ctx->rx_queue_index;
    __u32 *pkt_count;

    pkt_count = bpf_map_lookup_elem(&xdp_stats_map, &index);
    if (pkt_count) {

        /* We pass every other packet */
        if ((*pkt_count)++ & 1)
            return XDP_PASS;
    }

    /* A set entry here means that the correspnding queue_id
     * has an active AF_XDP socket bound to it. */
    if (bpf_map_lookup_elem(&xsks_map, &index))
        return bpf_redirect_map(&xsks_map, index, 0);

    return XDP_PASS;
}
```



socket(AF_INET, SOCK_STREAM, 0);
socket(AF_INET, SOCK_DGRAM, 0)

socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

socket(PF_XDP, SOCK_RAW, 0)



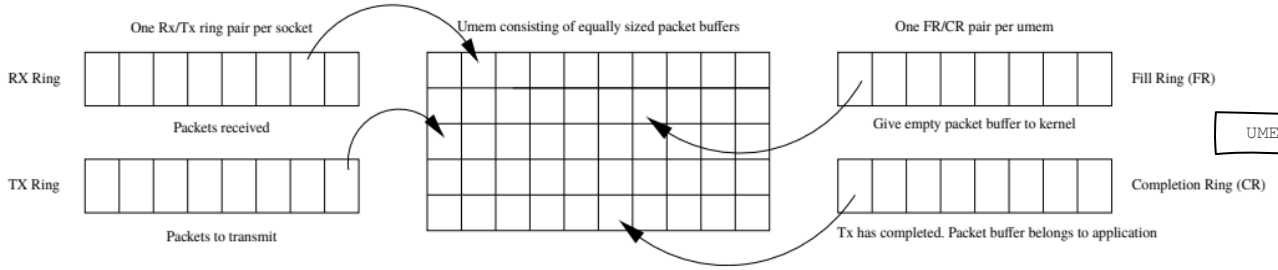
AF_XDP User Space

Descriptor interaction over 4 Single-Producer/Single-Consumer (SPSC) desc queues
zero-copy movement of packet data between user space and the kernel

```
sfd = socket(PF_XDP, SOCK_RAW, 0);
buffs = calloc(num_buffs, FRAME_SIZE);
setsockopt(sfd, SOL_XDP, XDP_MEM_REG, buffs);
setsockopt(sfd, SOL_XDP, XDP_{RX|TX|FILL|COMPLETION}_RING, ring_size);
mmap(..., sfd, ......); /* map kernel rings */
bind(sfd, "/dev/eth0", queue_id,....);
for (;;) {
        read_process_send_messages(sfd);
};
```

<xdp/xsk.h>
xsk_umem__create
xsk_socket__create
xsk_umem__delete
xsk_socket__delete
xsk_umem__fd
xsk_socket__fd
xsk_ring_prod__fill_addr
xsk_ring_cons__comp_addr
xsk_ring_prod__tx_desc
xsk_ring_cons__rx_desc
xsk_prod_nb_free
xsk_cons_nb_avail
xsk_ring_prod__reserve
xsk_ring_prod__submit
xsk_ring_cons__peek
xsk_ring_cons__release
xsk_umem__get_data

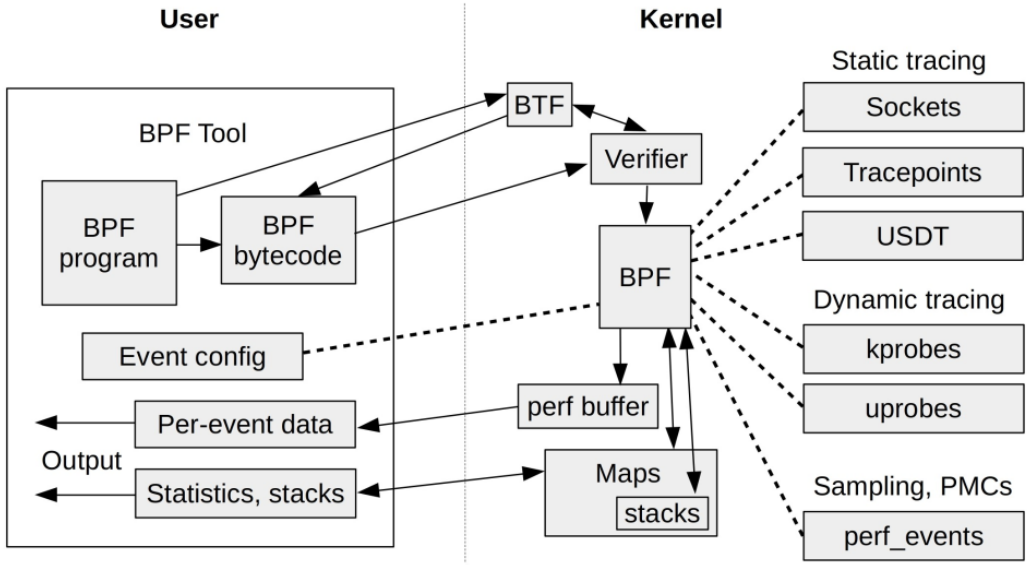
```
/* Rx/Tx descriptor */
struct xdp_desc {
        __u64 addr;
        __u32 len;
        __u32
options;
};
```



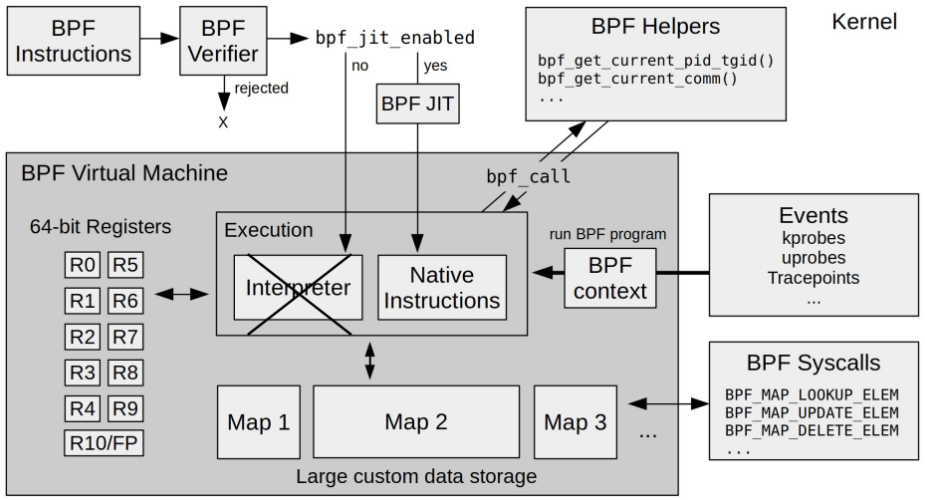

```
UMEM descriptor is __u64
```

net2018_talks/lpc18_paper_af_xdp_perf-v2

Summary



External



Internal

A game-changer in the Linux ecosystem, providing unprecedented programmability and flexibility within kernel in a safe manner.