From tracing to kernel programming
Agenda

01 Verifiable instruction set
02 tracing
03 networking
04 security
05 next
Tracing roots of BPF
tetris implemented as bpftrace script

https://github.com/mmisono/bpftrace-tetris
BPF is a sequence of commands that can be understood
BPF is an universal assembly language

- strictly typed assembly language
- safe for kernel and for HW
- stable instruction set
- extensions are backwards compatible
BPF use cases

- user space tells kernel what to do
- HW tells kernel what to do
- one computer tells another computer what to do

... the response could be: "yeah, I can do this" or "No, not right now".

... and since the intent is understood it's execution can be in a different form.
  (CPU executes one instruction, BPF executes whole program)
BPF vs Sandboxing (wasm, ...)

- BPF program is understood before execution
- Sandbox restricts execution environment. It doesn't understand what's running in the sandbox.
BPF design

- verifiable ISA
- write programs in C and compile into BPF ISA with GCC/LLVM
- Just-In-Time convert to modern 64-bit CPU
- minimal performance overhead:
  - bpf vs native (C -> BPF ISA -> native ISA vs C -> native ISA)
  - transition from native to bpf (native code -> BPF code -> native code)
- BPF calling convention compatible with modern 64-bit ISAs
- extend BPF (eBPF) ISA proposed in 2013
  - first appeared in the kernel as internal BPF (iBPF)

- Quiz:
  - What's faster the kernel in C or compiled through BPF ?
extensions of extended BPF (2014 till now)

- ISA was extended 5 times
  - <, <= instructions
  - 32-bit compare
  - atomics
- LLVM support -mcpu=v1, v2, v3
- -mcpu=v4 landed July 2023.
  - sign extending loads
  - bswap
  - long jmp
  - sdiv/smod
- GCC and LLVM support -mcpu=v4
BPF enables innovation

• BPF satisfies my own thirst for innovation

• BPF enables others to innovate
  – within BPF infra
  – in other kernel subsystems

• That's why I still work on BPF!
Innovation in the kernel BPF subsystem (Sep 2023)

Number of BPF developers per month (green - Meta BPF team, blue - the rest of BPF community)
BPF hooks hierarchy
BPF tracing - BPF for kernel observability

- Tracing mechanisms:
  - [ku]probe + bpf
  - tracepoint + bpf
  - fentry + bpf

- Capabilities
  - read all kernel data

- Restrictions
  - cannot modify kernel state
  - cannot crash or warn
BPF for kernel and user observability

- Tools
  - bcc
  - bpftrace
  - retsnoop
  - pyperf

- Use cases implemented with "BPF tracing"
  - Explain why kernel returns -EINVAL
  - Measure the latency of this syscall
  - How much time GCC spends processing #include vs compiling the rest
  - Tell me where my python program spends time
  - How many Gbytes my android phone used on facebook and youtube
BPF networking - BPF in firewalls, routers

- Network stack:
  - XDP + bpf
  - TC + bpf
  - cgroup + bpf
  - netfilter + bpf
  - TCP + bpf

- Capabilities
  - read packet data
  - modify and drop packets
  - modify TCP state

- Restrictions
  - cannot read arbitrary kernel data
  - cannot modify kernel state
BPF networking

- Tools
  - cilium
  - katran

- Use cases
  - firewall
  - K8s network connectivity
  - L4 load balancer
  - L7 socket load balancing
  - live task upgrade without connectivity loss
  - TCP congestion control
BPF security

- Hooks:
  - LSM + bpf
  - syscall + bpf

- Capabilities and restrictions
  - can read arbitrary kernel data
  - can deny operations
  - can sleep
BPF security

- Tools
  - systemd
  - the rest is non public :(  

- Use cases
  - Alternative to selinux and apparmor
  - Disallow use of file system X
BPF next

- Hooks:
  - scheduler + bpf
  - hid + bpf
  - oom + bpf
  - fuse + bpf

- Capabilities and restrictions
  - TBD
  - subsystem defines what is necessary
sched-ext

- New sched_class, at a lower priority than CFS
- Enables scheduling policies to be written in BPF programs
- No ABI stability restrictions – purely a kernel <-> kernel interface
- Run-time safety checks to make sure tasks are not starved
Implementing scheduling policy

- BPF program must implement a set of callbacks
  - Task wakeup (similar to select_task_rq())
  - Task enqueue/dequeue
  - Task state change (runnable, running, stopping, quiescent)
  - CPU needs task(s) (balance)
  - Cgroup integration
HID-BPF: changing how the device looks and talks

```c
SEC("fmod_ret/hid_bpf_rdesc_fixup")
int BPF_PROG(rdesc_fixup, struct hid_bpf_ctx *hid_ctx)
{
    _u8 *data = hid_bpf_get_data(hid_ctx, 0, 4096 /* size */);
    /* invert X and Y definitions in the event stream interpretation */
    data[39] = 0x31;
    data[41] = 0x30;
    return 0;
}
```

`data` now contains the report descriptor of the device.

(Un)attaching this program triggers a disconnect/reconnect of the device.

Only 1 program of this type per HID device.
```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>

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

    // Check packet length
    if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) > data_end) {
        return XDP_DROP;
    }

    // Parse Ethernet header
    struct ethhdr *eth = data;
    if (eth->h_proto != htons(ETH_P_IP)) {
        return XDP_PASS;
    }

    // Parse IPv4 header
    struct iphdr *ip = data + sizeof(struct ethhdr);
    if (ip->version != 4) {
        return XDP_PASS;
    }

    // Extract source and destination IP addresses
    __u32 src_ip = ip->saddr;
    __u32 dst_ip = ip->daddr;

    // Do something with the IP addresses
    // ...

    return XDP_PASS;
}
```

Everything on this slide is AI generated
https://en.wikipedia.org/wiki/Gartner_hype_cycle

- Peak of Inflated Expectations
- Trough of Disillusionment
- Slope of Enlightenment
- Plateau of Productivity

- Innovation Trigger
- burned by the verifier
- new BPF users
Major shift in BPF architecture happened in 2022
BPF and the kernel

- all BPF programs before 2022

stable hook:
- kernel prepares data it wants BPF program to see
- kernel interprets return code
BPF in the kernel

- hid-bpf, sched-ext, netfilter, struct-ops
- "new tracing"

- Native calls: kernel $\leftrightarrow$ BPF $\leftrightarrow$ kernel
- BPF can refcnt ++, -- and stash kernel objects
- explicit bpf_rcu_read_lock/unlock

- NO stable API
Katran - production BPF prog written in "Restricted C"

SEC("xdp")

int balancer_ingress(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct eth_hdr *eth = data;
    __u32 eth_proto;
    __u32 nh_off;

    nh_off = sizeof(struct eth_hdr);
    if (data + nh_off > data_end)
        return XDP_DROP;
    eth_proto = eth->eth_proto;
    if (eth_proto == bpf_htons(ETH_P_IP))
        return process_packet(data, nh_off, data_end, false, ctx);
    else if (eth_proto == bpf_htons(ETH_P_IPV6))
        return process_packet(data, nh_off, data_end, true, ctx);
    else
        return XDP_PASS;
}
Early days of BPF aka "Restricted C"

• All functions are __always_inline
• Single input argument
  – a pointer to context that is program type dependent. Ex: struct __sk_buff.
• No loops
• No memory allocation
• No type information
__attribute__((always_inline)) is no longer necessary.
BPF supports global and static functions.

```c
static __always_inline __u32 get_packet_hash(struct packet_description *pckt,
                    bool ipv6)
{
  // ..
}

static __always_inline bool get_packet_dst(struct real_definition **real,
                    struct packet_description *pckt,
                    struct vip_meta *vip_info,
                    bool is_ipv6)
{
  __u32 hash = get_packet_hash(pckt, is_ipv6) % RING_SIZE;
  // ...
}

static __always_inline int parse_icmpv6(void *data, void *data_end, __u64 off,
                    struct packet_description *pckt)
{

#pragma unroll is no longer necessary.

BPF supports iterators.

```c
int i;

#if NEW_KERNEL
bpf_for(i, 0, STACK_MAX_LEN) {
#else
#pragma clang loop unroll(full)
for (i = 0; i < STACK_MAX_LEN; i++) {
#endif
   // ...
}
```
int err_cast(struct task_struct *tsk)
{
    return((struct sk_buff *)tsk)->len;
}

int err_release_twice(struct __sk_buff *skb)
{
    struct bpf_sock_tuple tuple = {};

    struct bpf_sock *sk = bpf_sk_lookup_tcp(skb, &tuple, sizeof(tuple), 0, 0);
    bpf_sk_release(sk);
    bpf_sk_release(sk); // NOT OK in BPF C
    return 0;
}
Extended C with Symbolic Access

struct __sk_buff {
  __u32 len;
  __u32 pkt_type;
  __u32 mark;
  __u32 queue_mapping;
  ...
};

struct sk_buff {
  /* field names and sizes should match to those in the kernel */
  unsigned int len, data_len;
  __u16 mac_len, hdr_len, queue_mapping;
  struct net_device *dev;
  /* order of the fields doesn't matter */
  refcount_t users;
} __attribute__((preserve_access_index));

Instructs compiler to generate symbolic field access instead of constant integer offsets.
Dynamic structure layout.
BPF program adjusts itself depending on the target kernel.
Extended C with Type Information

SEC("tp_btf/netif_receive_skb")
int BPF_PROG(trace_netif_receive_skb, struct sk_buff *skb)
{
    p.type_id = bpf_core_type_id_kernel(struct sk_buff);
    p.ptr = skb;
    /* pretty print an skb */
    bpf_snprintf_btf(str, STRSIZE, &p, sizeof(p), 0);

    int .. = bpf_core_type_size(struct task_struct);

    bool .. = bpf_core_type_exists(struct io_uring);
}

BTF type id is determined at load time
Extended C with Kconfig

```
extern unsigned long CONFIG_HZ __kconfig;
extern int LINUX_KERNEL_VERSION __kconfig;

SEC("tc")
int nf_skb_ct_test(struct __sk_buff *ctx)
{
    struct nf_conn *ctlk;

    test_delta_timeout = ctk->timeout - bpf_jiffies64();
    test_delta_timeout /= CONFIG_HZ;
}
```

Unlike kernel modules the kconfig values are not known at compile time. They become known at load time.

The verifier can optimize the code with dead-code-elimination.
Extended C with Exception Tables

```c
#pragma clang attribute push (__attribute__((preserve_access_index)), apply_to = record)

struct net_device {
    int ifindex;
};

struct sk_buff {
    struct net_device *dev;
};

SEC("tp_btf/kfree_skb")
int BPF_PROG(trace_kfree_skb, struct sk_buff *skb, void *location)
{
    return skb->dev->ifindex;
}
```

Load instructions are replaced with inline version of `copy_from_kernel_nofault()` and exception tables generated.
Extended C with Type Tags

// include/linux/compiler_types.h

#if defined(CONFIG_DEBUG_INFO_BTF) && defined(CONFIG_PAHOLE_HAS_BTF_TAG) && __has_attribute(btf_type_tag)
#define BTF_TYPE_TAG(value) __attribute__((btf_type_tag(#value)))
#endif

#define __user BTF_TYPE_TAG(user)
#define __rcu BTF_TYPE_TAG(rcu)

normal C: for debug kernel and for sparse tool to warn
extended C: access is enforced by the verifier.
Cannot do RCU dereference outside of RCU critical section.
rcu_read_unlock() invalidates the pointers.
Use-After-Free is prevented.
# Extended C with Operator new

```c
#define __kptr __attribute__((btf_type_tag("kptr")))
struct foo {
    int var;
};
struct map_value {
    // __kptr tag makes C pointer behave like std::unique_ptr<struct foo>
    struct foo __kptr *ptr;
};
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __type(value, struct map_value);
} lru_map SEC(".maps");

SEC("fentry/do_nanosleep")
int nanosleep(void *ctx)
{
    // equivalent to C++ operator new that returns std::unique_ptr<struct foo>
    // std::make_unique<struct foo>();
    struct foo *p = bpf_obj_new(typeof(*p));

    struct map_value *v = bpf_map_lookup_elem(&lru_map, ...);
    // equivalent to C++ std::swap(v->ptr, p)
    old = bpf_kptr_xchg(&v->ptr, p);
}
```
struct foo {
    struct bpf_list_node node;
    int data;
};
struct bar {
    struct bpf_rb_node node;
    int var;
};
private(A) struct bpf_spin_lock lock;
private(A) struct bpf_list_head head __contains(foo, node);
private(A) struct bpf_rb_root root __contains(bar, node);

static bool cmp_less(struct bar *a, struct bar *b)
{
    return a->var < b->var;
}
void bpf_prog(struct foo *f, struct bar *b)
{
    bpf_spin_lock(&lock);
    f->data = 42;
    b->var = 0xB9F;
    bpf_list_push_front(&head, &f->node);
    bpf_rbtree_add(&root, &b->node, cmp_less);
    bpf_spin_unlock(&lock);
}
struct val_t {
    long b, c, d;
};

struct elem {
    long sum;
    struct val_t __percpu_kptr *pc;
};

struct {
    __uint(type, BPF_MAP_TYPE_CGRP_STORAGE);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, int);
    __type(value, struct elem);
} cgrp SEC(".maps");

const volatile int nr_cpus;

int BPF_PROG(test_cgrp_local_storage) {
    struct task_struct *task;
    struct val_t __percpu_kptr *p;
    struct val_t *v;
    struct elem *e;
    int i;

    task = bpf_get_current_task_btf();
    e = bpf_cgrp_storage_get(&cgrp,
                              task->cgroups->dfl_cgrp, 0, 0);

    p = e->pc;

    bpf_for(i, 0, nr_cpus) {
        v = bpf_per_cpu_ptr(p, i);
        if (v)
            sum_field_c += v->c;
    }

    return 0;
}
Extended C with Assertions

```c
u8 cpu_to_dom_id(u32 cpu) {
    u8 dom_id;
    bpf_assert(cpu < MAX_CPUS);
    dom_id = cpu_dom_id_map[cpu];
    bpf_assert(dom_id < MAX_DOMS);
    return dom_id;
}

void dom_add_cpu(u32 cpu, u8 dom_id) {
    u64 *word = &dom_cpu[dom_id][cpu / 64];
    bpf_assert_within(word, dom_cpu, sizeof(dom_cpu));
    *word |= 1LLU << (cpu % 64);
}
```

assert() is a verifier aid. The verifier doesn't have to compute and enforce the bounds. The BPF program will automatically abort. The stack will be unwound, destructors called and program detached.
## Early days BPF vs modern BPF

<table>
<thead>
<tr>
<th></th>
<th>Early BPF</th>
<th>Modern BPF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution context</strong></td>
<td>rcu_read_lock + preempt_disable</td>
<td>rcu_read_lock_trace</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>stable uapi/bpf.h</td>
<td>Unstable and kernel dependent</td>
</tr>
<tr>
<td></td>
<td>fixed input context (single argument)</td>
<td>many arguments (type match)</td>
</tr>
<tr>
<td></td>
<td>fixed set of helpers</td>
<td>whitelisted set of kernel functions</td>
</tr>
<tr>
<td></td>
<td>fixed output codes</td>
<td>scalars and pointer return values (type match)</td>
</tr>
<tr>
<td></td>
<td>fixed set of hooks</td>
<td>whitelisted empty functions</td>
</tr>
<tr>
<td><strong>New features appear as</strong></td>
<td>new prog types</td>
<td>one prog type</td>
</tr>
<tr>
<td></td>
<td>new map types</td>
<td>kernel exposes new 'struct bpf_' types and kfuncs</td>
</tr>
<tr>
<td></td>
<td>new hooks</td>
<td></td>
</tr>
<tr>
<td><strong>Backward compatibility</strong></td>
<td>guaranteed</td>
<td>relies on CO-RE. May fail to load depending on kconfig, version</td>
</tr>
</tbody>
</table>

Like user space  Like kernel modules
Pros and Cons of kernel modules vs BPF

• Pro
  — Arbitrary C code
  — Access to all EXPORT_SYMBOL

• Con
  — One wrong step and panic
  — Have to be compiled with the kernel sources
    — Dynamic Kernel Modules Support require compiler on the host
  — Once compiled becomes a binary blob with no visibility
BPF programs are safe and portable kernel modules

• Safety is builtin
• Portability is achieved with CO-RE, kconfigs, type info
  – It's not guaranteed. BPF program may need to be adjusted to remain portable.
• Debuggable
  – All types are embedded in BPF prog and maps
  – Source code is embedded in binary
  – The verifier understands the purpose. No way to hide what bpf prog is doing.
• EXPORT_SYMBOL_GPL == BPF kfuncs
  – there is no EXPORT_SYMBOL equivalent. All modern BPF progs are GPL.
BPF flavor of the C language is a better choice for kernel programming.

Any kernel subsystem may choose to extend itself with BPF programs without touching BPF core and sending emails to bpf@vger.