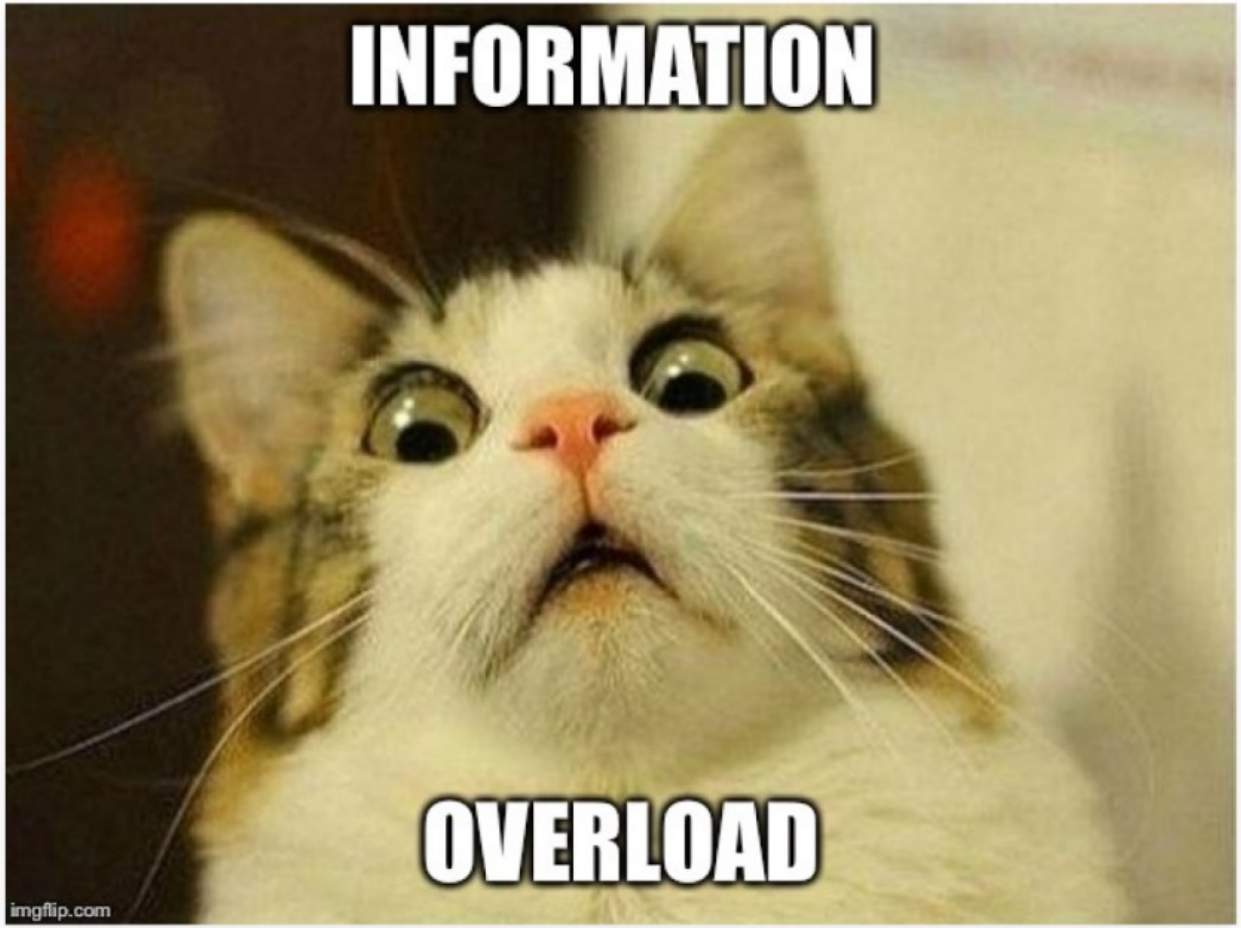
Dinko Korunić
HAProxy Technologies

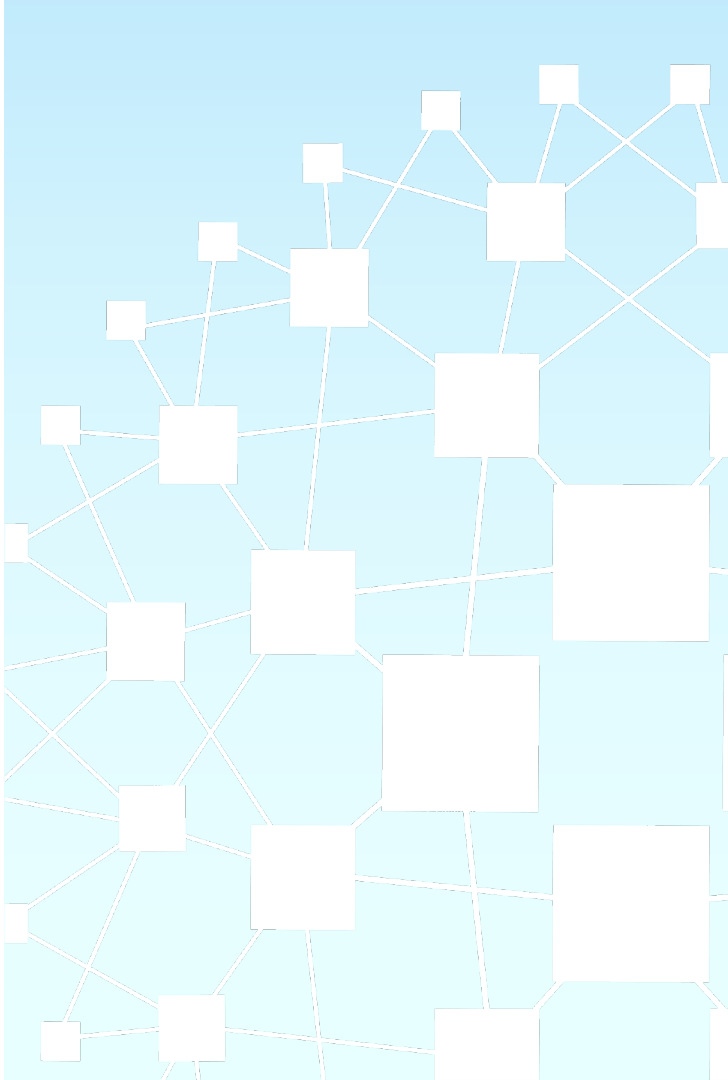
Caveats using eBPF in Linux

(“Safe, fast, and… occasionally terrifying.”)


INFORMATION
OVERLOAD
imgflip.com

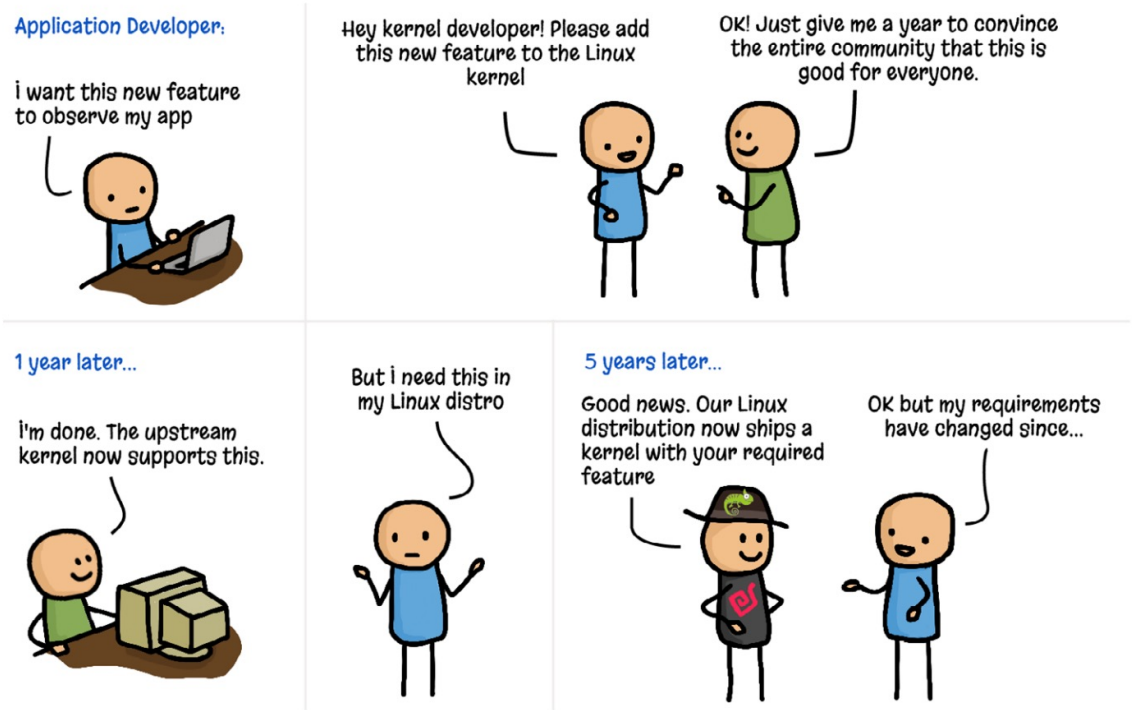
Agenda

- Short recap ⚡
- Pros and cons, complexities and limitations ⚖️
- Verifier limitations 🛡️
- Case study: dkorunic/pktstat-bpf 🔍
- EOF 🏁



Short recap

Linux kernel development flow



eBPF development flow

Source: x.com/breakawaybilly

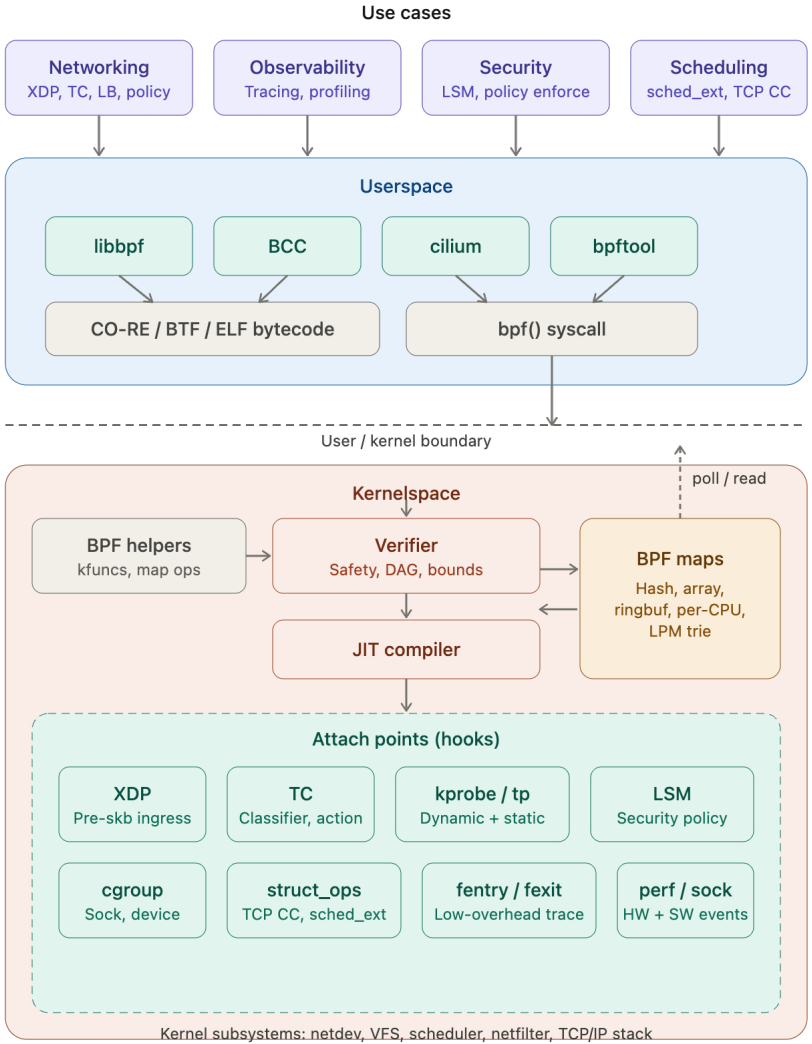
Overview

- eBPF evolved from packet filtering to a **general purpose** “computing machine”
- provides:
  - high-performance **networking**, **observability**, continuous **profiling**, **monitoring**, **security** tools
- allows **custom code** to run in **Linux kernel** in a **safe** manner
  - it takes **years** to get new functionality added (**kernel patch** accepted) and for it to reach **production** (Linux distributions) environments
  - no need to **patch** kernel and/or maintain kernel patches
  - LKMs are **risky**: security, performance, stability, compatibility…
  - custom code can be **loaded** and **unloaded** on demand without any negative impact
- macro perspective:
  - **passive** - event driven (does not poll)
  - runs in a kernel “**virtual machine**”
  - with various **attachment points** (hooks) – different program types

Overview (cont)

- micro perspective:
  - verified **JIT-compiled** code runs in a **kernel sandbox** as **native instructions**
  - interacts with **userland** software, kernel and kernel modules (ie. NIC drivers)
  - even able to run in fully **offloaded** mode (SmartNICs)
  - can change kernel data/logic and communicate with userspace application through **maps**
  - **high performance** – no translation kernelspace-userspace
- **attachments** - eBPF hooks for custom programming w/ different eBPF programs:
  - **kprobes**: dynamic kernel probes for tracing kernel functions
  - **uprobes**: userspace probles for tracing userspace application functions
  - **tracepoints**: static probes for tracing specific kernel events
  - **networking**:
    - **XDP**: for high-performance packet processing directly w/ NIC drivers and HW (Express Data Path)
    - **TC**: traffic shaping and packet processing (Traffic Control)
    - **sockmap/sockops**: socket splicing, processing, policies, etc.
  - **cgroup-bpf**: applying eBPF to cgroups
  - **LSM**: Linux Security Module API for security enforcing

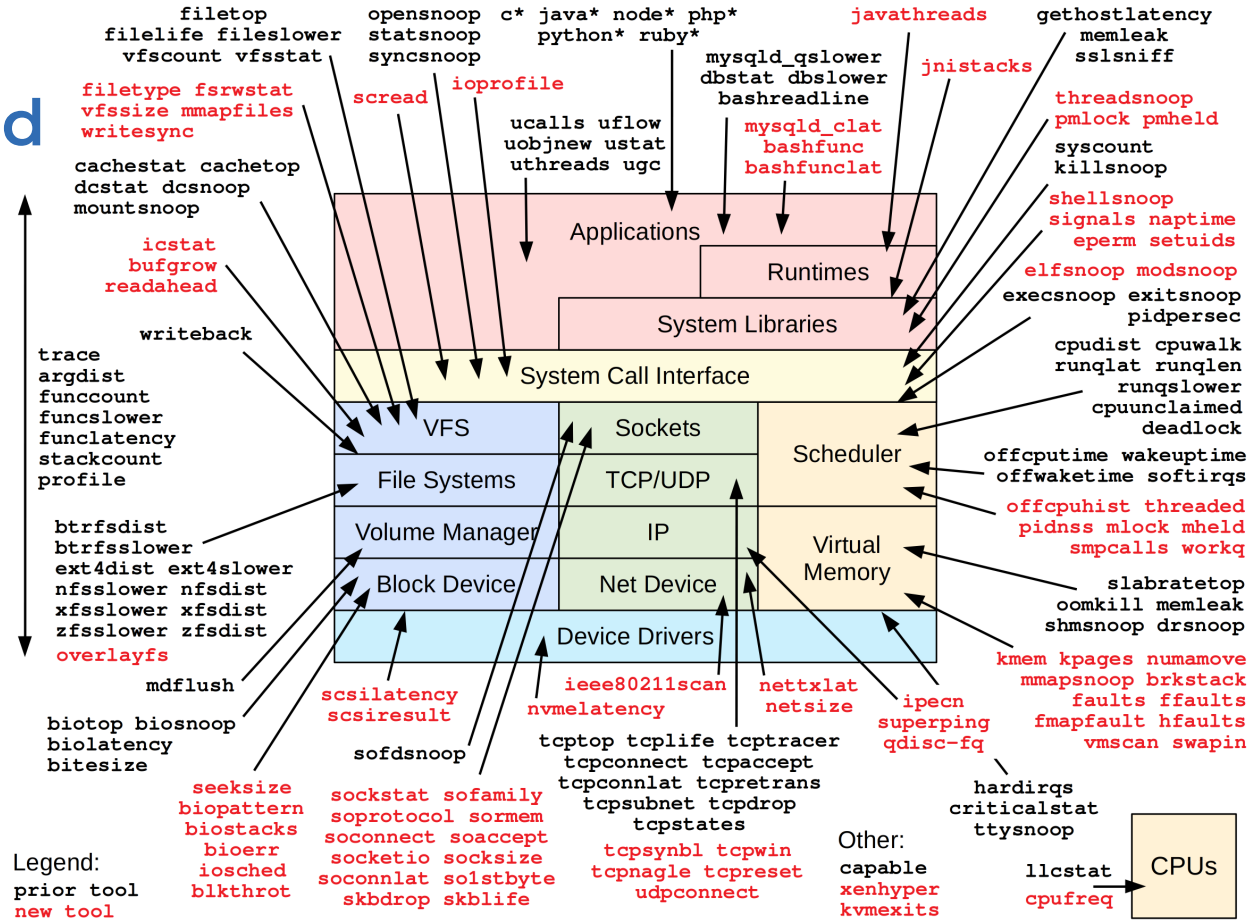
eBPF architecture



Workflow

1. **source code** (only C or Rust) is compiled into **bytecode**
2. bytecode is loaded from userspace to **kernelspace** and **statically verified** by eBPF **verifier**: on more recent kernel versions w/ **BTF**, **CO:RE** enables program to run “everywhere”
3. **only verified** code can run, so it is not possible to:
   - tamper with kernel memory
   - exhaust resources via unbounded loops
   - leak kernelspace memory to userspace
4. upon approval kernel performs **JIT** (dynamic translation) and **hardened**
   - kernel memory holding program is made **r/o** to prevent manipulation
   - constants are **blinded** - memory addresses for constants are **randomized**
5. program can be either **offloaded** to HW or **executed** by CPU
6. **communication** with userspace programs through **maps**

Sysops redefined



source: BPF Performance Tools: Linux System and Application Observability

Cloudops redefined

- containers, K8s, Lambda, Fargate, etc.
- **shared kernel** on host/hypervisor – runs eBPF code and has **full visibility**
  - containers
  - networking (internal and external communication, open sockets)
  - files, processes, etc.
- no need to **change applications** or configuration
- eBPF programs run completely **transparent**
  - **per-container security** and **network policies**, but processed in-kernel
- **sidecars** in K8s – **many cons**:
  - instrumentation runs ad separate containers
  - application pod needs restart to add/remove sidecar
  - configuration (YAML) changes
  - pod start time slowed down w/ sidecars, different readiness
  - additional networking latencies w/ networking sidecars

Pros and cons, complexities and limitations

Best features

- **observability** w/o instrumentation overhead
  - kprobes/uprobes attach to kernel/userspace functions w/o recompilation or restart
  - full strack tracing (kernel -> libc -> app) with reduced perf buffer overhead compared
  - high perf BPF ring buffer

- **networking** – datapath programmability
  - eXpress Data Path: XDP – at driver level (wirespeed!) before sk_buff alloc
  - Traffic Control: TC – programmable classification/action at ingress/egress, more flexible than iptables/nftables and much faster
  - socket-level – per socket policy, transparent proxying (Cilium), socket-level LB

- **security** enforcement
  - Seccomp BPF filters w/ programmable logic
  - MAC policies as BPF programs and stacks w/ existing LSMs (AppArmor, SELinux)

Best features (++)

- performance **profiling**
  - kernel & user stack capture w/ minimal overhead
  - continous production profiling (Parca, Pyroscope etc.) w/o sampling distortion
  - fentry/fexit trampolines patch call sites directly and avoid breakpoint trap overhead - great for instrumenting hot paths

- **verifier** – conservative, rejects programs that it cannot prove safe
  - DAG walk – bounded loops are allowed, every instruction must be reachable
  - pointer types are not interchangeable, arithmetic type-constrained
  - packet access safe – you have to prove w/ conditional checks offset + size <= pkt_end
  - guaranteed termination, memory safety, bounted execution
  - stack r/o, reading uninitialized stack – reject
  - map access – prove w/ bounds on register offset + size <= map->value_size
  - deeply nested loops & large programs hit 1M verified instructions limit – logic has to be split across tail calls
  - inserted speculation barriers to mitigate Spectre v1 gadgets, pointer values never readable as scalars

Best features (+++)

- **portability**
  - BTF + CO-RE solved kernel-header dependency
  - programs relocate field access at load time for the running kernel

- **scheduling** and resource control
  - prototyping custom schedulers (NUMA, latency-aware, workload-specific) w/o reboots

- **shared state** model
  - hashes, maps, arrays, LRU, LPM trie, bloom filters, per-CPU variants
  - per-CPU maps avoid atomic contention but consume NR_CPU x value_size
  - shared maps need atomic ops or spinlocks

Not so great...

- write-heavy state is awkward – **maps** are not databases
- **complex logic** better done in userspace w/ BPF event export
- **debugging** is a pain:
  - bpf_printk is your printf and it as crude as it gets
  - verified log is essential for debugging rejections – but level 2 on large program can OOM log buffer
- **verifier** is also a pain: known verifier weakness and historical bypasses
  - ALU bounds mistrack – off by one in 32bit vs 64bit sign extension
  - speculative paths can access memory verifier proved unreachable
  - overly aggressive state pruning – discards constraints that matter on a different path
  - acquired references must be released on every path
- kernel source – is pretty much the only **documentation**
  - in packet processing, a lot of times you need to double check source and destination ports, etc.; also most of the time you will be double-checking RFCs...

Portability issues

- kernel struct **layout instability**
  - fields get added, reordered or even conditionally compiled
- CO-RE and **BTF dependancy**
  - BTF landed in 5.2, become common in distros with 5.4-5.8 kernel
  - RHEL 7/8, older Debians and embedded builds mostly lack BTF
  - BTFHUB Archive: https://github.com/aquasecurity/btfhub-archive/
- **helper** and **kfunc availability**
  - verifier rejects unknown helpers – needed compile-time or load-time feature detection
  - not really graceful: typically libbpf_probe_helper() or features.HaveProgramHelper() that construct minimal eBPF program and try to load it
  - kfuncs even worse, unstable API and no backward compatibility
  - hard failure if missing...
- **map type** and **feature availability**
  - newer map types (bloom filter, user ringbuf, arena) don't exist on older kernels
  - hard failure if missing...

Portability issues (++)

- **verifier behavior** drift
  - complexity limit, loop handling, allowed patterns evolve
  - bounded loops in 5.3, etc.
- **architecture-specific** concerns
  - JIT backends have varying level of completeness on less-common architectures
  - possible problems: atomics, 32-bit subregister handling, bpf_tail_call chain depth
- **privilege** and **security model** changes
  - kernel.unprivileged_bpf_disabled=2 in 5.16+, CAP_BPF in 5.8+, BPF_TOKEN in 6.9+

- practical approach w/ workarounds:
  - use CO-RE, conditional compilation with bpf_core_field_exists(), bpf_core_enum_value_exists(), extern kconfig variables, probe features at load time, avoid kfuncs, maintain minimum kernel version floor
  - splitting eBPF code and loading on demand, defensively coding, etc.

Security - BPF_TOKEN

- until Linux 6.9+ CAP_BPF is **non-namespaceable** because some BPF helpers can read **arbitrary kernel memory** -> unprivileged containers cannot use BPF
- **token**
  - kernel object (bpffs fd), carries scoped set of **permissions** (bitmask mount options)
  - **privileged process** (container runtime, systemd) **delegates operations** to **unprivileged process**
  - operations permitted: delegate_cmds, delegate_maps, delegate_progs, delegate_attachs
  - bpf() syscall from userspace contains token_fd
- kernel side:
  - checks token bitmasks
  - uses **ns_capable()** namespace local checking – checks CAP_BPF, CAP_NET_AMIN, CAP_PERFMON etc. against **user namespace** that contains token!
- caveats: token is non-revocable, namespace binding is the (only) containment mechanism, no tracing delegation in practice (tracing allows arbitrary kernel reads)

Verifier limitations

Verifier – NULL deref

- NULL deref – missing map lookup check

```
SEC("kprobe/sys_read")
int bad_null_check(struct pt_regs *ctx) {
    __u32 key = 0;
    __u64 *value = bpf_map_lookup_elem(&my_map, &key);
    return *value;  // direct dereference without NULL check
}
```

- ; return *value;
  R0 invalid mem access 'map_value_or_null'

- w/o conditional branch that tests R0 against zero, deref is rejected
- fix is trivial:
  `if (!value) return 0;`

Verifier – out-of-bounds access

- out-of-bounds packet access – missing bounds check

```
SEC("xdp")
int bad_packet_access(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = data;
    // Missing: if ((void *)eth + sizeof(*eth) > data_end) return XDP_DROP;
    __u16 proto = eth->h_proto;  // read at data+12..13
    return XDP_PASS;
}
```

- ; __u16 proto = eth->h_proto;
  invalid access to packet, off=12 size=2, R1(id=0,off=0,r=0)
  R1 offset is outside of the packet
- r=0 means proven range is 0 bytes, no evidence that single byte is accessible; w/ bounds check r=14 (Ethernet header size) and access at offset 12 with size 2 is within bounds

Verifier – unbounded/bounded loops

- unbounded loops rejected before 5.3; bounded loops allowed in 5.3+ but needs proven termination

```c
SEC("kprobe/sys_read")
int bad_loop(struct pt_regs *ctx) {
    int i = 0;
    volatile int sum = 0;
    while (i < some_map_value) {  // bound depends on runtime map value
        sum += i;
        i++;
    }
    return sum;
}
```

- pre 5.3: back-edge from insn 8 to 4
- 5.3+: BPF program is too large. Processed 1000001 insns
- fix: needs constant upper bounds or map values/function arguments with a mask

Verifier – stack out-of-bounds

- BPF stack – 512 bytes
- negative offsets from R10 (frame pointer) are OK, positive offsets are NOK
- reading uninitialized stack memory is rejected (you can't read before writing)

```
SEC("kprobe/sys_read")
int bad_stack(struct pt_regs *ctx) {
    char buf[600];  // 600 > 512 byte stack limit
    bpf_probe_read_kernel(buf, sizeof(buf), (void *)0);
    return 0;
}
```

- invalid write to stack R10 off=-600 size=600

```
SEC("kprobe/sys_read")
int bad_stack_read(struct pt_regs *ctx) {
    int x;
    return x;  // read from stack without prior write
}
```

- invalid read from stack R10 off=-4 size=4

Verifier – unreachable instructions

- first pass is DAG analysis – dead code rejected unconditionally
- Clang usually optimises dead code away

```
SEC("kprobe/sys_read")
int bad_unreachable(struct pt_regs *ctx) {
    return 0;
    int x = 42;  // dead code after unconditional return
    return x;
}
```

- unreachable insn 3

Verifier – invalid helper for program type

- each program type has a list of permitted helpers

```c
SEC("xdp")
int bad_helper(struct xdp_md *ctx) {
    __u64 pid = bpf_get_current_pid_tgid();  // tracing helper, not XDP
    return XDP_PASS;
}
```

- unknown func bpf_get_current_pid_tgid#14
- bpf_get_current_pid_tgid() is available in kprobe/tracepoint but not in XDP or TC
- and that's why process tracking is not available in XDP/TC mode in the later example…

Verifier – pointer arithmetic safety

- unprivileged_bpf_disabled=1 on many distros by default (requires root or CAP_SYS_ADMIN)
- in unprivileged mode pointer-pointer arithmetic and pointer leaking to scalars are rejected to prevent kernel address disclosure

```
SEC("socket_filter")
int bad_ptr_leak(struct __sk_buff *skb) {
    __u64 addr = (__u64)(void *)skb;  // cast context pointer to scalar
    return 0;
}
```

- R1 pointer arithmetic on pkt_or_null prohibited
- even in privileged mode adding two pointers is rejected

```
void *a = data;
void *b = data_end;
void *c = a + (__u64)b;  // pointer + pointer
```

- R2=SCALAR_VALUE ... math between pkt pointer and register with unbounded min value is not allowed

Verifier – complexity limit exceeded

- all reachable states explored - deeply nested conditionals cause exponential state explosion
- each independent conditional doubles the state space: 30 conditionals = 2^30 paths
- real-world programs hit this with deeply nested protocol parsers or large switch statements
- fix: refactor into tail calls or use bpf_loop() from 5.17+

```
SEC("kprobe/sys_read")
int bad_complexity(struct pt_regs *ctx) {
    __u64 x = bpf_ktime_get_ns();
    if (x & 1) x += 1;
    if (x & 2) x += 2;
    if (x & 4) x += 4;
    // ... repeat 30+ times
    if (x & (1<<30)) x += (1<<30);
    return x;
}
```

- BPF program is too large. Processed 1000001 insns

Verifier – register type mismatch on helper args

- helpers have typed argument signatures – passing wrong pointer type is caught at verification

```c
SEC("xdp")
int bad_helper_arg(struct xdp_md *ctx) {
    __u32 key = 0;
    // Passing packet pointer where map_key pointer is expected
    void *data = (void *)(long)ctx->data;
    bpf_map_lookup_elem(&my_map, data);  // data is PTR_TO_PACKET
    return XDP_PASS;
}
```

- R2 type=pkt expected=fp
- fix: bpf_map_lookup_elem arg2 expects stack pointer (fp-based) or similar safe memory, a packet pointer does not qualify

Verifier – invalid map fd / BTF mismatch

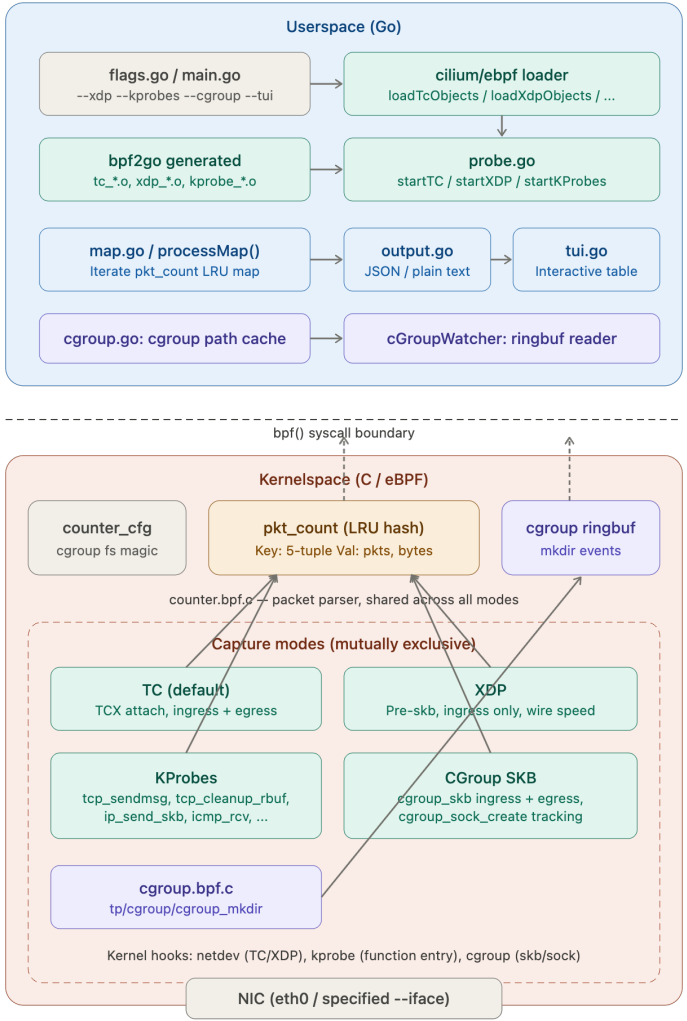
- ELF references a nonexistant map or BTF key/value types don’t match
- invalid map_ptr to access map

- … or at load time even before verifier
- map 'nonexistent_map': not found in kernel

- typical when running CO-RE program on kernel where BTF struct layouts have changed enough for the reallocation to fail

Case study: dkorunic/pktstat-bpf

pktstat-bpf diagram



Performance or compatibility?

- network traffic capture:
  - **performance** (XDP), both **ingress** and **egress** (TC), **rich data** (kprobes), **sockets** (skb)
  - pkt_count LRU hash map + atomics
  - max **compatibility**: split XDP, TC, kprobes, sockets into separate eBPF programs and load on demand separately
- kprobes:
  - TCP send/receive (tcp_sendmsg, tcp_cleanup_rbuf), IP send (ip_send_skb, ip6_send_skb), UDP consume (skb_consume_udp) and ICMP send/receive paths
  - **process-level visibility** (PID, comm, cgroup path) but higher overheads
- map traffic to cgroups:
  - only for cgroup and kprobe modes
  - kernel only has **cgroup IDs**: attach separate eBPF program and attach to cgroup/cgroup_mkdir
  - emit events to **perf ringbuf** that Go side consumes and maintains in-memory cgroup ID – **path cache**
- a lot, a lot of **manual packet processing**!

Circumventing limitations

- **separate** eBPF **programs**:
  - increase **kernel compatibility** but more complex development and attaching
  - **different contexts** and **helpers**:
    - XDP uses xdp_md raw packet pointers
    - TC uses __sk_buff with L2/L3 metadata
    - kprobe uses pt_regs with access to pid/tgid, comm and cgroup_id
    - cgroup SKB use __sk_buff but w/ different attach points
  - shared parsing logic
  - **cannot tail-call** across different program types

Mode	Min upstream	Min RHEL/CentOS	BTF required	Interface-bound
KProbes	5.2 (4.10 w/ backport)	8 (4.18)	Yes	No — all interfaces
CGroup SKB	5.2 (4.10 w/ backport)	8 (4.18)	Yes	No — per cgroup
XDP	5.9	9 (5.14)	No	Yes — single NIC
TC (TCX)	6.6	9 (5.14 maybe)	No	Yes — single NIC

Use the source Luke

- kprobes vs TC/XDP:
  - TC (runs in qdisc path) and XDP (runs before skb alloc) see **raw frames** on specific NIC
  - kprobes operate at **kernel function** level – events happen regardless of interface (NAT, veth, forwarding – does not matter) and work from 4.18+ kernel
  - at high pps more **expensive**: kprobe inserts breakpoint (int3 on x64) at function prologue, triggering a trap into debug handler, then jump to BPF and return
- it is not easy to pick correct kprobe hooks – **kernel is the documentation**
  - TCP
    - tcp_sendmsg – egress, when process calls send() or write() on a TQCP socket
    - tcp_cleanup_rbuf – ingress, when receive buffer is cleaned up after data delivery to userspace
  - UDP
    - ip_send_skb, ip6_send_skb – egress, fires after UDP layer has built skb and is about to hand to IP for routing
    - skb_consume_udp – ingress, fires when UDP datagram is consumed from the receive queue
  - ICMP
    - __icmp_send, icmp6_send – egress, fires when kernel generates ICMP/ICMPv6 message
    - icmp_recv, icmpv6_recv – ingress, fires on ICMP receive path
    - raw_sendmsg, rawv6_sendmsg – raw socket ICMP (sadly requires more modern kernel!)

Kprobe networking caveats

- many catch22s and caveats w/ contexts, structs, packet lens, options, swaps, etc.

- TCP send w/ **tcp_sendmsg()** – skc_num/inet_sport is src port, skc_dport is dest port
- TCP receive w/ **tcp_cleanup_rbuf()** – swaps src/dst, so skc_rc_saddr is local and skc_daddr is remote!
- UDP send w/ **ip6_send_skb()** – we read from sk->sk_protocol instead of ip6hdr->nexthdr because of IPv6 extensions (IPv6 extension headers would make nexthdr something other than UDP…)
- UDP receive w/ **skb_consume_udp()** – again swapped src/dst
- ICMP send w/ **__icmp_send()** – they fire with triggering packet's skb and not ICMP reply, so src/dst Ips come from original packet (src sender, dst local), need to swap to reflect actual ICMP error direction…

Storage – LRU_HASH definition

- bounded memory w/o userspace cleanup:
  - **auto-eviction** least recently user entries when map reaches capacity
  - old idle connections lose their counters, but **active flows survive**
  - LRU eviction – implicit GC driven by **access recency**

```
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, MAX_ENTRIES);  // likely 65536 or similar
    __type(key, struct stat_key);
    __type(value, struct stat_value);
} pkt_count SEC(".maps");
```

Storage – writes and atomics

- kernel update pattern:
  - **atomics** (__sync_fetch_and_add) required – we use **unified view** across CPUs
  - on SMP same **flow** can be processed **concurrently** on multiple CPUs: same for TC/XDP (multiple RX queues hitting same BPF) and kprobes (multiple tasks sending simultaneously)
  - if another **CPU raced** and **inserted key** between failed lookup and update call, we rather do **–EEXIST** than overwrite other CPUs counters; we are fine with losing a single packet count

```
struct stat_value *val = bpf_map_lookup_elem(&pkt_count, &key);
if (val) {
    __sync_fetch_and_add(&val->packets, 1);
    __sync_fetch_and_add(&val->bytes, pkt_len);
} else {
    struct stat_value newval = { .packets = 1, .bytes = pkt_len };
    bpf_map_update_elem(&pkt_count, &key, &newval, BPF_NOEXIST);
}
```

Storage – writes and atomics (++)

- race-free 3 step pattern: lookup -> insert-if-absent -> retry-lookup

```
statvalue *val = bpf_map_lookup_elem(&pkt_count, key);
if (val) {
    __sync_fetch_and_add(&val->packets, 1);
    __sync_fetch_and_add(&val->bytes, size);
} else {
    statvalue initval = {.packets = 1, .bytes = size};
    if (bpf_map_update_elem(&pkt_count, key, &initval, BPF_NOEXIST) != 0) {
        val = bpf_map_lookup_elem(&pkt_count, key);
        if (val) {
            __sync_fetch_and_add(&val->packets, 1);
            __sync_fetch_and_add(&val->bytes, size);
        }
    }
}
```

Storage – userspace reads

- userspace (Go) reading options:
  - **userspace** is just a **passive observer**, never writes map (simpler concurrency model)
  - **iterate** or **batch** (requires kernel 5.6+) reading
  - BatchLookup reads entire map in chunks of 4096 with cursor
- why batch reading matters for LRU?
  - get_next_key interation on LRU has a fundamental issue
  - LRU can **evict entry** between to calls causing **iterator** to **lose position** and kernel returns ENOENT – **aborting interation**!
  - **batch lookup is robust** – copies a **snapshot** of entries atomically (per batch) reducing window of LRU eviction between batches
- very important – **buffer pooling**
  - **minimize latencies** during map reading
  - avoid repeated 4096-element array allocations, use pre-allocated slices, avoid GC pressure in hot read loop

Cgroup nightmare

- bpf_get_current_cgroup_id() – cgroup ID u64 from kernel
  - corresponds to **inode number** of **cgroup** directory
  - **no eBPF helper** to get **cgroup path**, kernel does not expose kernfs_path() or cgroup_path()
- how to detect cgroup v1 vs v2 – **statfs(2)** the **/sys/fs/cgroup** mount point:
  - **0x63677270** – pure cgroup v2, bpf_get_current_cgroup_id() returns v2 inode ✅
  - **0x27e0eb** – pure cgroup v1, bpf_get_current_cgroup_id() might return 0 or root cgroup ID 😔
  - **0x1021994** – systemd hybrid case, /sys/fs/cgroup is tmpfs with v1 controller mounts and unified v2 hierarchy is mounted at /sys/fs/cgroup/unified
- eBPF code – needs fs magic hint from userspace:
  - userspace detects fs magic -> hands through map to eBPF
  - cgroup v2: code can **walk task->cgroups->dfl_cgrp** directly ✅
  - cgroup v1: code needs to tranverse **css_set** to find right controller 😔

Cgroup nightmare (++)

- we still just have cgroup ID
- need to handle this in userspace – snapshot initial state:
  - **walkdir** /sys/fs/cgroup on start
  - **stat()** each entry to get inode numbers
  - **store inode** -> path mapping in **cache hashmap**
- live updates:
  - userspace **misses** cgroups created **after startup**
  - **not possible** to get **inotify/fanotify** events from kernel on virtual fs
  - **separate cgroup** eBPF **module** – attached to raw_tracepoint **cgroup_mkdir**
  - each time kernel creates new cgroup directory – **perf event** is sent to userspace
  - userspace gets events, inserts into **cache map**
  - perf buffer overflows – trigger full **fs re-walk**
  - **percpu_array** used with **single entry** as **scratch buffer** to build event struct – no stack allocations (BPF stack is 512-byte stack)

Cgroup nightmare (+++)

- **userspace** cache **lookups** – when reading map entry with cgroup ID:
  - **fast path** – r_lock, check cache, r_unlock; O(1) hash lookups, no contention
  - **slow path** – mu lock (no thundering herd), rebuild to fresh map, copy map, mu unlock
  - **negative caching** – ID not found, negative cache it to avoid repeated walks

- a lot of failures and edge cases:
  - **cgroup v1 ID unreliability**: on pure v1 systems bpf_get_current_cgroup_id() may return root cgroup ID (1)...
  - we **keep** cached **stale entries** and avoid cache eviction due to short-lived containers issue
  - in **hybrid mode** v2 and v1 hierarchies are in **same tree**, inode numbers from v1 controller might not match bpf_get_current_cgroup_id() returned values (for v2 hierarchy), inherent limitation of hybrid mode

Packet processing

- L2 Ethernet parsing – only IP and IPv6, everything else ignored

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

- L3 IPv4 parsing – also handles IPv4 options (IHL field: Record Route, Timestamp, Loose/Strict Source Routing etc.)

```
__u8 ihl = ip4->ihl;
if (ihl < 5) return NOK;
__u32 ip4_hdr_len = (__u32)ihl * 4;
if ((void *)ip4 + ip4_hdr_len > data_end) return NOK;
void *transport = (void *)ip4 + ip4_hdr_len;
```

  - IPv4 addresses need to be converted to IPv4-mapped IPv6 format for uniform storage

Packet processing (++)

- L3 IPv6 parsing – also walks extensions headers (Hop-By-Hop, Routing, Fragment, Destination Options); loop limit is 6 due to verifier bound loop constraint. Authentication Header and Encapsulating Security Payload are not walked.

```
for (int i = 0; i < 6; i++) {
    if (nexthdr != IPPROTO_HOPOPTS && nexthdr != IPPROTO_ROUTING &&
        nexthdr != IPPROTO_FRAGMENT && nexthdr != IPPROTO_DSTOPTS)
        break;
    if (nexthdr == IPPROTO_FRAGMENT) {
        transport += 8;  // fragment header is fixed 8 bytes
    } else {
        transport += ((__u32)(hdrlen + 1) * 8);  // variable-length
    }
}
```

- L4 TCP

```
struct tcphdr *tcp = transport;
if ((void *)tcp + sizeof(*tcp) > data_end) return NOK;
key->src_port = bpf_ntohs(tcp->source);
key->dst_port = bpf_ntohs(tcp->dest);
```

Packet processing (+++)

- L4 TCP – TCP options are not inspected (MSS, Window Scale, SACK, timestamps). TCP flags either, we just do traffic counting

```
struct tcphdr *tcp = transport;
if ((void *)tcp + sizeof(*tcp) > data_end) return NOK;
key->src_port = bpf_ntohs(tcp->source);
key->dst_port = bpf_ntohs(tcp->dest);
```

- L4 UDP – UDP options (RFC 9308) are not considered. UDP-Lite (proto 136) not handled.

```
struct udphdr *udp = transport;
if ((void *)udp + sizeof(*udp) > data_end) return NOK;
key->src_port = bpf_ntohs(udp->source);
key->dst_port = bpf_ntohs(udp->dest);
```

Packet processing (++++)

- L4 ICMP/ICMPv6 – repurposing same storage/struct to store type and code (Destination Unreachable, Time Exceeded, Redirect etc.)

```
// ICMPv4
key->src_port = icmp->type;
key->dst_port = icmp->code;

// ICMPv6
key->src_port = icmp->icmp6_type;
key->dst_port = icmp->icmp6_code;
```

- byte count calculation for ICMP v4

```
size_t msglen = sizeof(struct icmphdr) + ihl_bytes + (payload > 8 ? 8 : payload);
```

EOF: Congratulations on reaching this! ☺

Have more questions?

dkorunic@haproxy.com

https://www.haproxy.com/contact-us