Presentation

Dinko Korunić

EBPF: FEATURES, CAPABILITIES AND IMPLEMENTATION

Agenda

1. Intro and overview
2. Coding and deployment
3. Network hooks and performance
4. Security concerns

ONE DOES NOT SIMPLY

STOP WRITING SLIDES

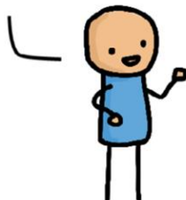
imgflip.com

Intro and overview

Application Developer:

i want this new feature to observe my app

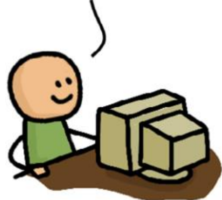
Hey kernel developer! Please add this new feature to the Linux kernel



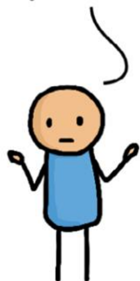
OK! Just give me a year to convince the entire community that this is good for everyone.

1 year later...

i'm done. The upstream kernel now supports this.



But i need this in my Linux distro



5 years later...

Good news. Our Linux distribution now ships a kernel with your required feature

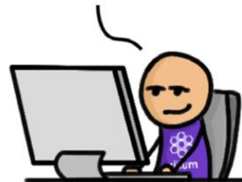
OK but my requirements have changed since...

Application Developer:

i want this new feature to observe my app

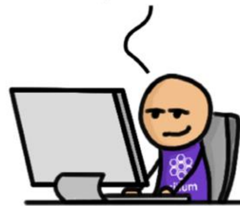
eBPF Developer:

OK! The kernel can't do this so let me quickly solve this with eBPF.



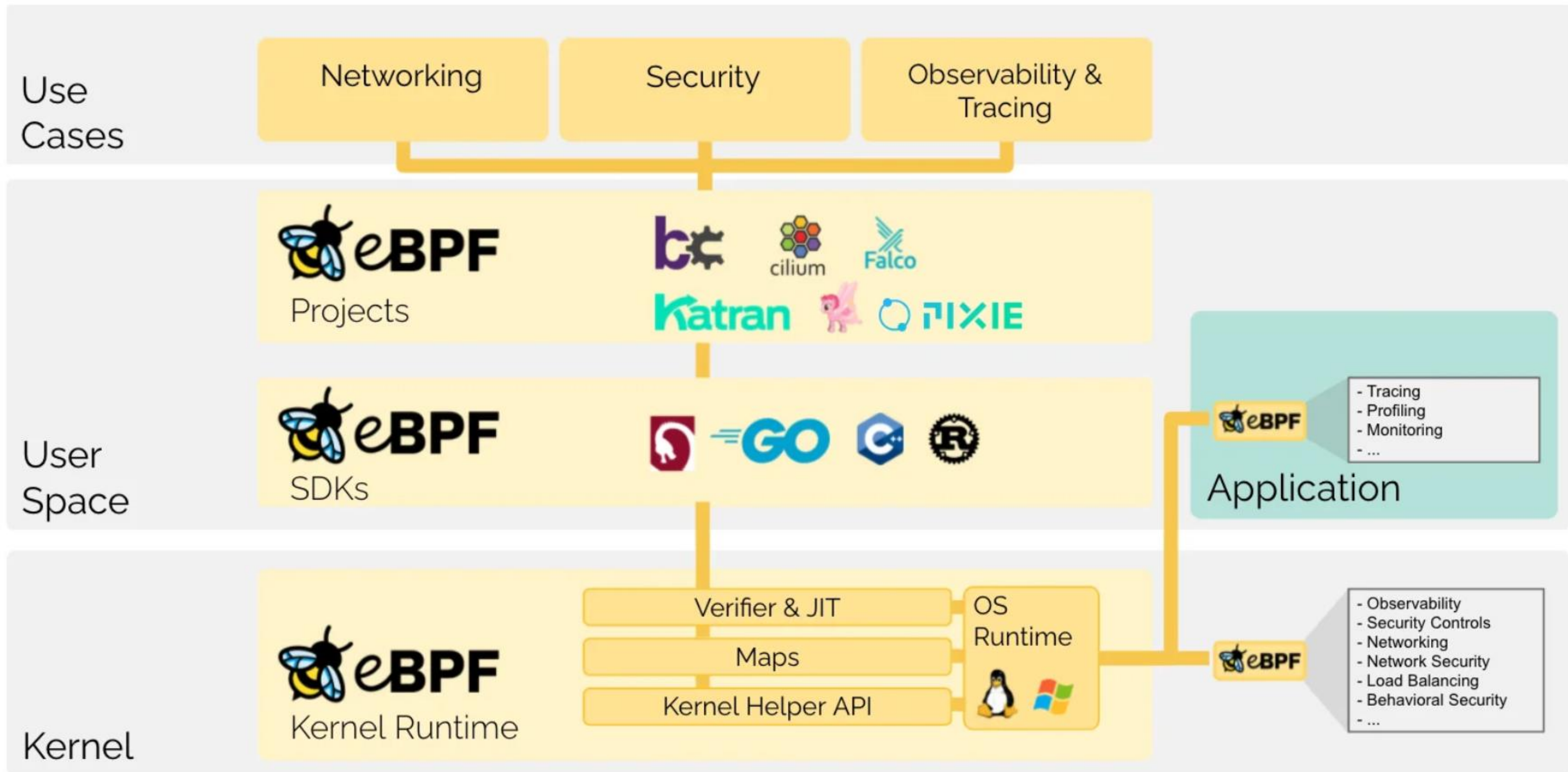
A couple of days later...

Here is a release of our eBPF project that has this feature now. BTW, you don't have to reboot your machine.



Source: x.com/breakawaybilly

What is eBPF



source: eBPF documentation

Overview

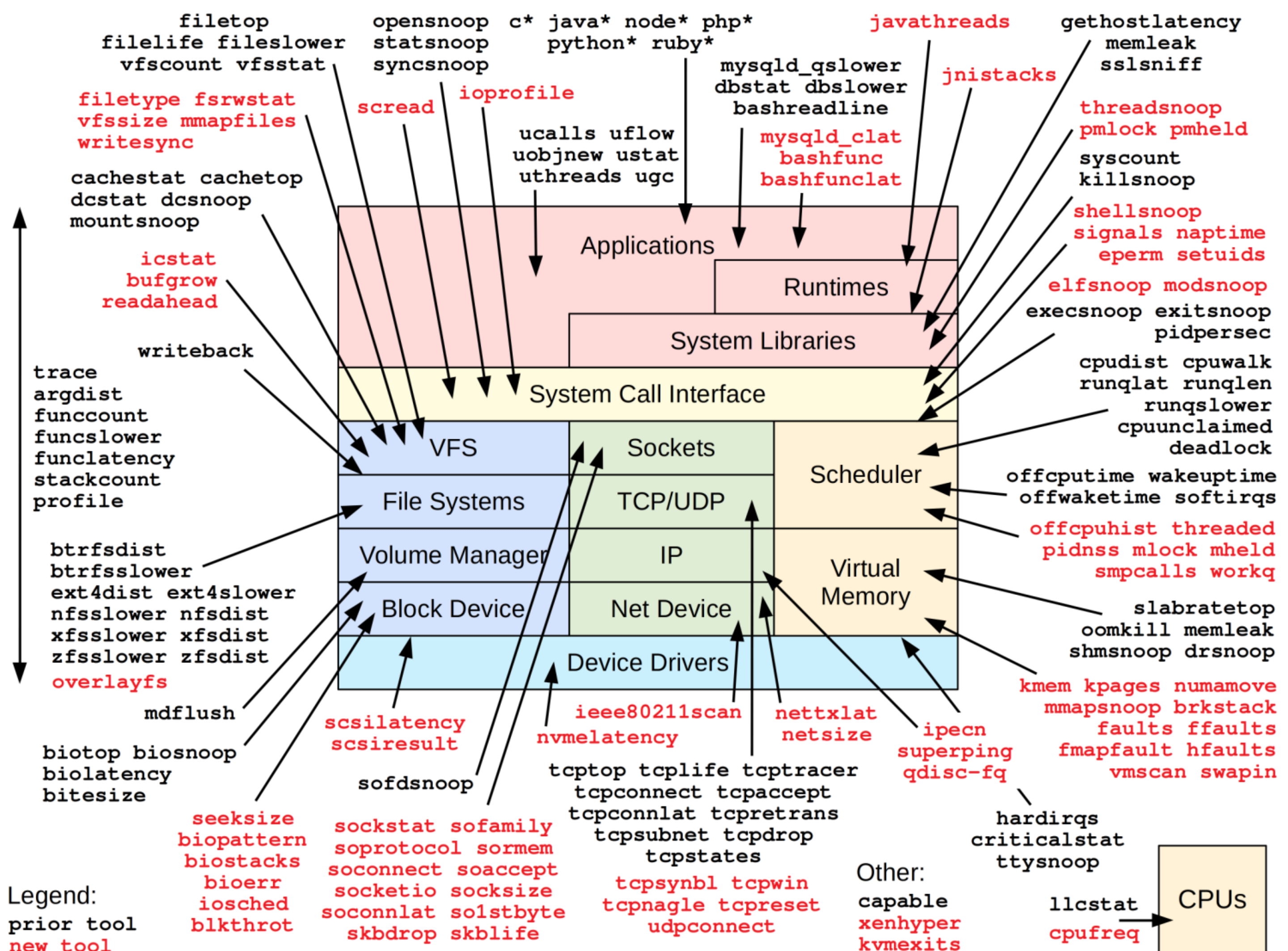
- eBPF evolved from packet filtering to a **general purpose** "computing machine"
- provides:
  - high-performance **networking**, **observability**, continuous **profiling**, **monitoring**, **security** tools
- allows **custom code** to run in **Linux kernel** in a **safe** manner
  - it takes **years** 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
    - **TC**: traffic shaping and packet processing
    - **sockmap/sockops**: socket splicing, processing, policies, etc.
  - **cgroup-bpf**: applying eBPF to cgroups
  - **LSM**: Linux Security Module API for security enforcing

Program types

- networking:
  - BPF_PROG_TYPE_SOCKET_FILTER: program to perform socket filtering
  - BPF_PROG_TYPE_SCHED_CLS: program to perform traffic classification at the TC layer
  - BPF_PROG_TYPE_SCHED_ACT: program to add actions to the TC layer
  - BPF_PROG_TYPE_XDP: program to be attached to the eXpress Data Path hook
  - BPF_PROG_TYPE_SOCKET_OPS: program to catch and set socket operations such as retransmission timeouts, passive/active connection establishment etc.
  - BPF_PROG_TYPE_SK_SKB: program to access socket buffers and socket parameters (IP addresses, ports, etc) and to perform packet redirection between sockets
  - BPF_PROG_TYPE_FLOW_DISSECTOR: program to do flow dissection (to find important data in network packet headers)
- kernel tracing, monitoring:
  - BPF_PROG_TYPE_PERF_EVENT: program to attach to HW and SW perf events
  - BPF_PROG_TYPE_KPROBE: program to attach to kprobes (kernel routines)
  - BPF_PROG_TYPE_TRACEPOINT: program to attach to predefined trace points
- many more...



Source: BPF Performance Tools: Linux System and Application Observability

Popular examples

- **Cloudflare XDP** DDoS mitigation
- Facebook/Meta **Katran** L4 load balancer
- **Netflix Flow Exporter** for TCP flows (IP change events and flow log data) in realtime for system monitoring, profiling, network segmentation, forecasting, etc.
- **Vmware Carbon Black EDR** Linux sensors
- **Datadog NPM**: network performance monitoring
- **Tetragon**: security, observability, runtime enforcement
- **Falco**: cloud-native threat-detection for containers and K8s
- **Inspektor-Gadget**: system insights for K8s/containers, observability etc.
- **Pixie**: debugging for applications on K8s, protocol tracing, connection tracing, etc.
- **Cilium**: networking (segmentation), observability, security
- **Calico**: cloud-native container networking and security (eBPF dataplane)

Cloud-native environments

- 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

Fast evolving

Feature	Kernel required	Description	Scenario
TC	4.1	Network traffic classification and control	Networking
XDP	4.8	Network dataplane programming technology (for L2/L3 services)	Networking
Cgroup socket	4.10	Network filtering and accounting attached to Cgroups	Container
AF_XDP	4.18	Network packets are directly sent to the user mode (similar to DPDK)	Networking
Sockmap	4.20	Short circuit processing	Container

source: OpenEuler - eBPF Introduction

Fast evolving (cont)

Feature	Kernel required	Description	Scenario
Cgroup sysctl	5.2	Monitor control and access to sysctl usage in Cgroups	Container
BPF trampoline	5.5	Replace kprobe/kretprobe for better performance (zero overhead), troubleshooting and debugging eBPF programs	Performance tracing
KRSI (LSM + eBPF)	5.7	Kernel Runtime Security Instrumentation – attaching to various LSM hooks and control (allow/deny); custom MAC policies with arbitrary code	Security

source: OpenEuler - eBPF Introduction

Coding and deployment

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**

BPF PROGRAM
LIGHTYEAR
WORKS EEEEVERYWHERE!
imgflip.com

Portability - CO:RE

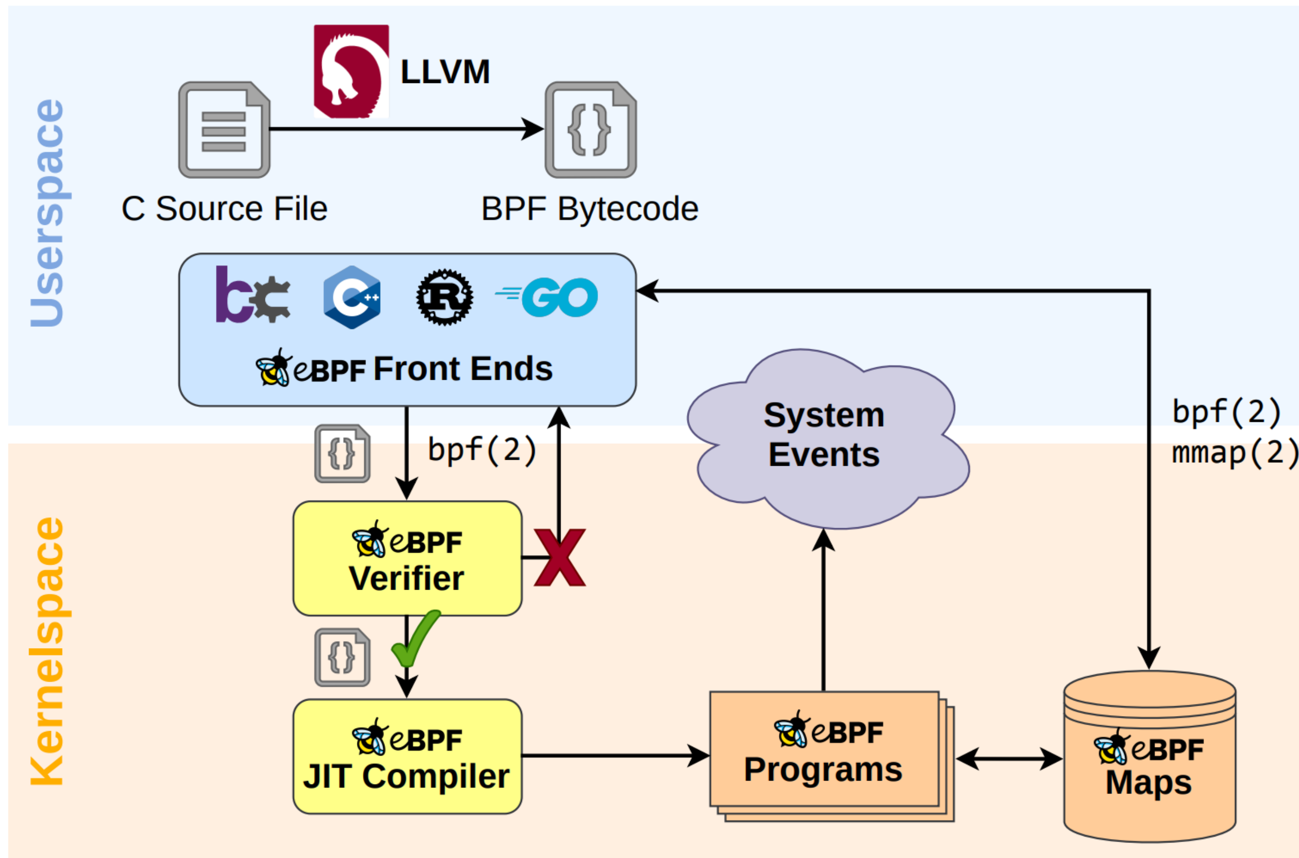
- **BCC** (BPF Compiler Collection) framework
  - requires **large** compile **toolchain** (Clang/LLVM)
  - used to **compile** BPF on target host during **runtime**
  - requires local kernel headers which have to match with running kernel and compiles on the fly – memory layout of the kernel is exactly what BPF program expects
- **CO:RE** – Compile Once Run Everywhere
  - **portability**: compile, pass verification and work correctly across different kernel versions
  - kernel types and data structures are **frequently changed** (fields renamed, shuffled, moved into new inner struct, removed, types changed, etc.)
  - BPF interfaces are **stable**
- how does it work:
  - **BTF** (BPF Type Format) information exposed by kernel in `/sys/kernel/btf/vmlinux`
  - in-kernel BTF from Linux kernel 4.18, for older kernels there is btfhub repository
  - Clang emits BTF **relocations**: descriptions of what BPF program intended to access

Portability - CO:RE (cont)

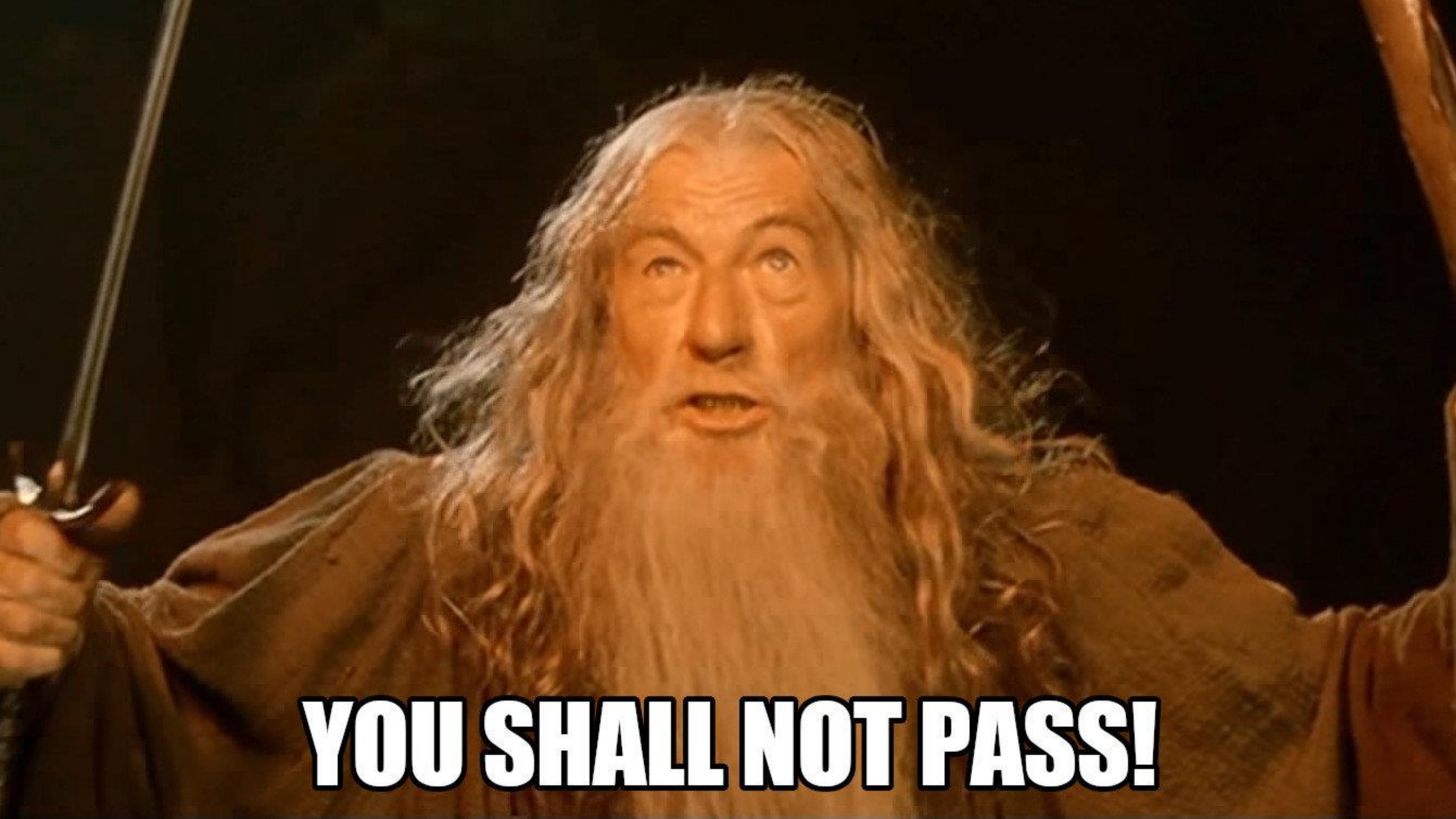
- BPF program loader (ex. libbpf or cilium-ebpf) on start:
  - processes BPF ELF bytecode
  - reads **BPF program BTF** information (types, relocation information etc.)
  - reads **running kernel BTF** information
  - matches both for all types and fields and **updates** all **offsets** and **relocatable data** to make sure BPF program logic is matching running kernel
  - **custom tailored** BPF program is passed on to eBPF verifier
  - possible to use extern Kconfig variables and struct flavors for **incompatible changes**

```
extern u32 LINUX_KERNEL_VERSION __kconfig;
…
if (LINUX_KERNEL_VERSION >= KERNEL_VERSION(4, 11, 0))
    utime_ns = BPF_CORE_READ(task, utime);
else
    utime_ns = BPF_CORE_READ(task, utime) * (1000000000UL / CONFIG_HZ);
```

Architecture



source: BPFCONTAIN: Fixing the Soft Underbelly of Container Security


YOU SHALL NOT PASS!

Verifier

- **type checking** of operations
- **stack limit** of 512b
- limit **1M instructions** – but it is possible to call other eBPF programs
- no signed division
- build state machine, check for correct behaviours
- guarantee of termination:
  - DFS search if program can be parsed to **directed acyclic graph** (DAG)
  - only if there are **no backward jumps**
  - **loops** have to be **predefined size** (can have loop unroll)
  - check for **unreachable instructions**
- compute **worst-case execution**

Verifier (cont)

- some **helpers** can be called only if the **license** is compatible (**GPL**)
- **guaranteed safety** – invalid memory access must never happen
- **disallowed memory access** beyond local variables and packet boundaries
  - to access any byte in the packet, it is required to perform a **border check**
  - **following pointers** – only through bpf_probe_read()
- **floating point arithmetic** – not permitted

```
static inline int process_ip4(struct iphdr *ip4, void *data_end, statkey *key) {
  // validate IPv4 size
  if ((void *)ip4 + sizeof(*ip4) > data_end) {
  return NOK;
}
```

Maps

- generic **key-value stores**, user-defined structures and types with fixed sizes
- accessible from both **userspace** and **kernelspace** – means of information exchange
- 24 different map types
  - some are **per CPU** (performance reasons, agreggation happens in userspace application), most are global (**spinlock** or **atomic operations** required)
- use **locked memory** – sometimes limits (RLIMIT_MEMLOCK) are too low
- **ref-counted** and can be pinned to filesystem at `/sys/fs/bpf`

```
struct {
  __uint(type, BPF_MAP_TYPE_LRU_HASH); // LRU hash requires 4.10 kernel
  __uint(max_entries, MAX_ENTRIES);
  __type(key, statkey);
  __type(value, statvalue);
} pkt_count SEC(".maps");
```

Map types

- BPF_MAP_TYPE_ARRAY: a map where entries are indexed by a number
- BPF_MAP_TYPE_PROG_ARRAY: a map that stores references to eBPF programs
- BPF_MAP_TYPE_HASH: stores entries using a hash function
- BPF_MAP_TYPE_PERCPU_HASH: a map/hash table for each processor
- BPF_MAP_TYPE_LRU_HASH: a map that stores entries using hash function with LRU removal
- BPF_MAP_TYPE_LRU_PERCPU_HASH: allows the creation of a hash table for each processor core with LRU remove policy
- BPF_MAP_TYPE_PERCPU_ARRAY: an array for each processor core
- BPF_MAP_TYPE_LPM_TRIE: longest-prefix match (LPM) trie
- BPF_MAP_TYPE_ARRAY_OF_MAPS: an array to store references to eBPF maps
- BPF_MAP_TYPE_HASH_OF_MAPS: a hash table to store references to eBPF maps
- many more..

Helper functions

- special **functions** offered by **kernel infrastructure**
- interacting with maps, modifying packets, printing messages to kernel trace
- each program type has different helper functions
- 100s of helpers:
  - bpf_map_delete_elem, bpf_map_update_elem, bpf_map_lookup_elem: used to remove, install or update, and search elements from maps
  - bpf_get_prandom_u32: returns a 32-bit pseudo-random value
  - bpf_l4_csum_replace, bpf_l3_csum_replace: used to recalculate L4 and L3 checksums
  - bpf_ktime_get_ns: returns time since system boot, in nanoseconds
  - bpf_redirect, bpf_redirect_map: functions to redirect packets to other network devices. The second allows specifying the device dynamically through a special redirection map
  - bpf_skb_vlan_pop, bpf_skb_vlan_push: remove/add VLAN tags from a packet
  - bpf_getsockopt, bpf_setsockopt: similar to user-space calls to getsockopt() and setsockopt() to get/set socket options
  - bpf_get_local_storage: returns a pointer to a local storage area, can be shared
  - …

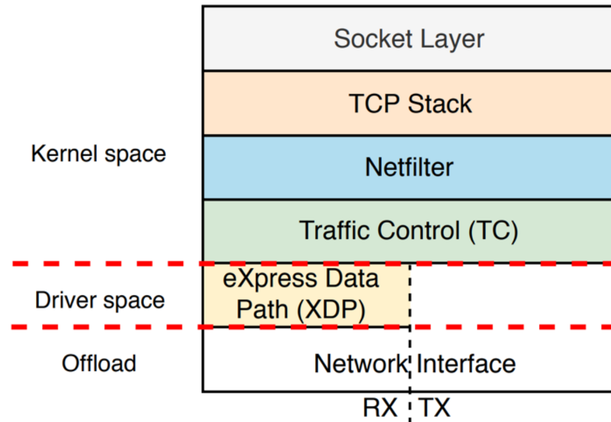
Compiler toolchain and frontends

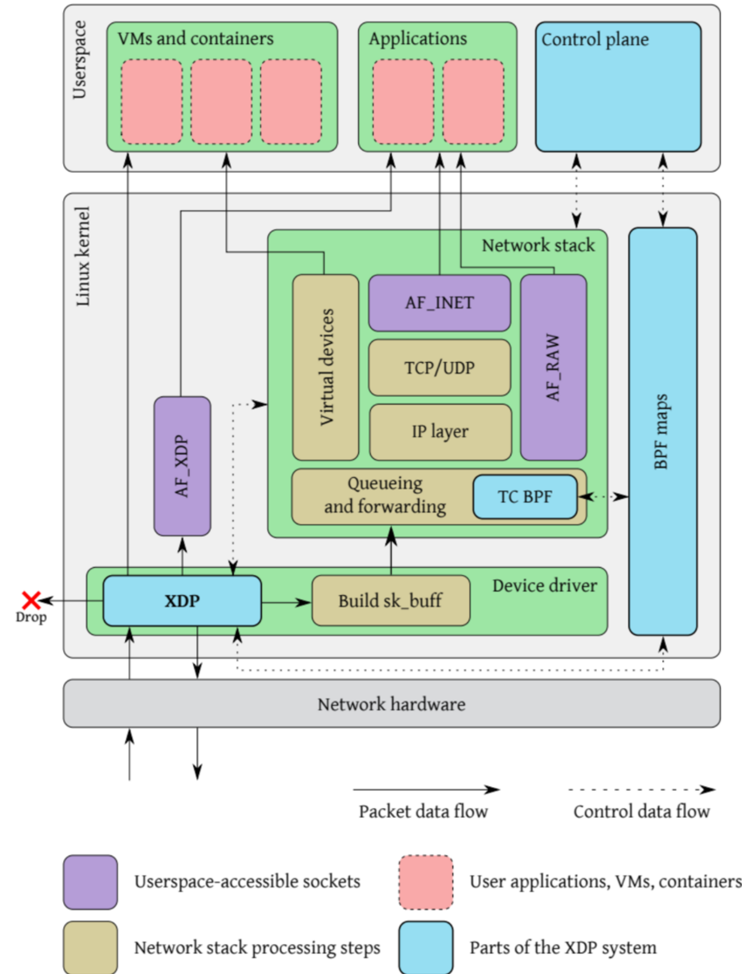
- **C** (BPF Compiler Collection):
  - libbpf
- **Rust**:
  - Libbpf-rs: Rust wrapper around libbpf
  - Aya: purely in Rust, syscalls through libc crate, can be built w/ musl
- **Golang**:
  - Libbpfgo: Go wrapper around libbpf C code, sadly uses CGo
  - ebpf-go (Cilium): bpf2go compiles C to eBPF bytecode, generates Go file containing eBPF and Go types for map keys and values
- Second-tier support:
  - Python:
    - libbpf Python bindings, PyEBPF, pybpf, bpfmaps
  - Ruby, Lua...

Network hooks and performance

Network hooks

- most important hooks:
  - eXpress Data Path **XDP**
    - **only for RX**
    - **high performance**,
    - can be **offloaded** to NIC
  - Traffic Control **TC**
    - both RX and TX
    - mid performance





source: Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications

eXpress Data Path – XDP

- programmable **packet processing** technology
- widely adopted: Cilium, Meta Katran, various anti-DDoS tools, Calico
- issues: works only on **ingress/RX path**, fairly **basic context** (`struct xdp_md`)
- alternatives (DPDK, Netmap) bypass kernel and have better performance (poll mode), but completely take-over NIC:
  - XDP has lower CPU usage, dynamic attach/detach without service interruptions
  - XDP does not break existing networking stack, security, routing, etc.
- on packet arrival before processing data, eBPF program is called to execute actions:
  - XDP_ABORTED: error, drop packet w/ exception
  - XDP_DROP: drop packet silently
  - XDP_PASS: forwards packet to regular stack (TC eBPF program further in the chain)
  - XDP_TX: forward packet back (can be modified) on the same interface
  - XDP_REDIRECT: redirect to different interface, CPU (further processing) or userspace AF_XDP sockets (userspace processing)
- packet data can be **read/written** and even **resized** (with checksum recalculation)

eXpress Data Path – XDP (cont)

- models:
    - **generic** XDP – ordinary network path, kernel emulates native execution, doesn’t have full performance due to extra socket buffer allocation
    - **native** XDP – loaded by NIC driver, works in the initial receive path, needs driver (i40e, nfp, mlx*, ixgbe) support, drivers are regularly getting XDP support
    - **offloaded** XDP – runs directly on NIC, executes w/o CPU, needs driver and HW support (SmartNICs: nVidia/Mellanox BlueField and ConnectX, Netronome Agilio NFP), runs at wirespeed, great for low-latency high-speed workloads
- DDoS mitigation
    - XDP_DROP happens at **early stage**, **efficient filtering** w/ very **low cost** per packet
    - **scrubbing** and forwarding legitimate traffic using XDP_TX
- forwarding, load-balancing
    - use XDP_TX and XDP_REDIRECT
- monitoring, flow sampling
    - possible complex packet analysis, adding custom metadata etc.

XDP C example

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
__uint(type, BPF_MAP_TYPE_ARRAY);
__type(key, __u32);
__type(value, __u64);
__uint(max_entries, 1);
} pkt_count SEC(".maps");

char __license[] SEC("license") = "Dual
    MIT/GPL";
```

```c
SEC("xdp")
int count_packets() {
  __u32 key = 0;
  __u64 *count =
    bpf_map_lookup_elem(&pkt_count, &key);
  if (count) {
   __sync_fetch_and_add(count, 1);
  }

  return XDP_PASS;
}
```

XDP Rust/Aya example

```
#![no_std]
#![no_main]

use aya_ebpf::{bindings::xdp_action, macros::xdp,
    programs::XdpContext};
use aya_log_ebpf::info;

#[xdp]
pub fn xdp_hello(ctx: XdpContext) -> u32 {
match unsafe { try_xdp_hello(ctx) } {
  Ok(ret) => ret,
  Err(_) => xdp_action::XDP_ABORTED,
  }
}
```

```
unsafe fn try_xdp_hello(ctx: XdpContext) -> Result<u32,
    u32> {
info!(&ctx, "received a packet");
Ok(xdp_action::XDP_PASS)
}

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
unsafe { core::hint::unreachable_unchecked() }
}
```

XDP performance

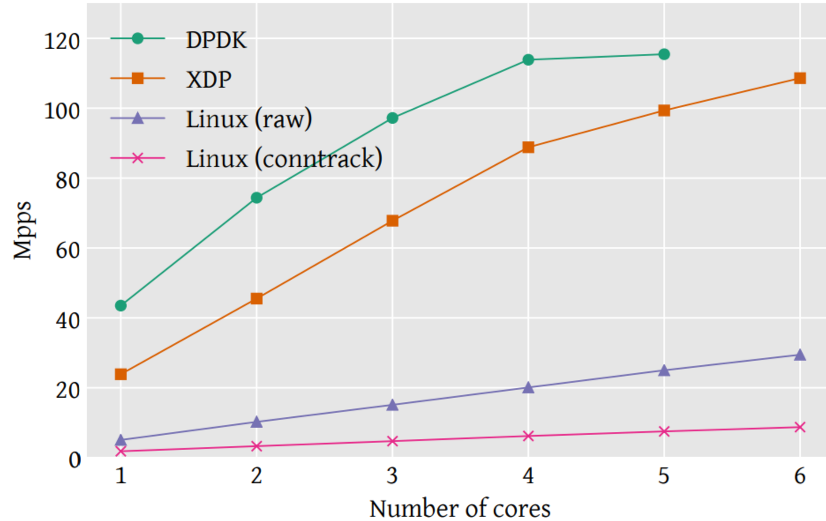


Figure 3: Packet drop performance. DPDK uses one core for control tasks, so only 5 are available for packet processing.

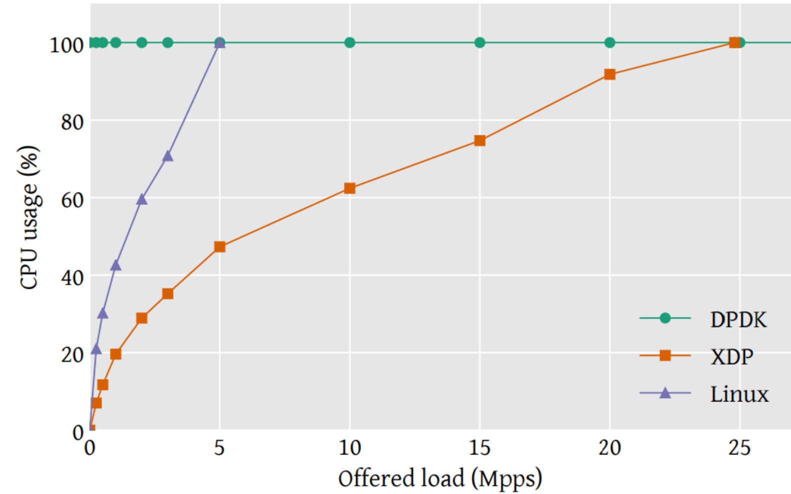


Figure 4: CPU usage in the drop scenario. Each line stops at the method's maximum processing capacity. The DPDK line continues at 100% up to the maximum performance shown in Figure 3.

source: The eXpress data path: fast programmable packet processing in the operating system kernel

XDP performance (cont)

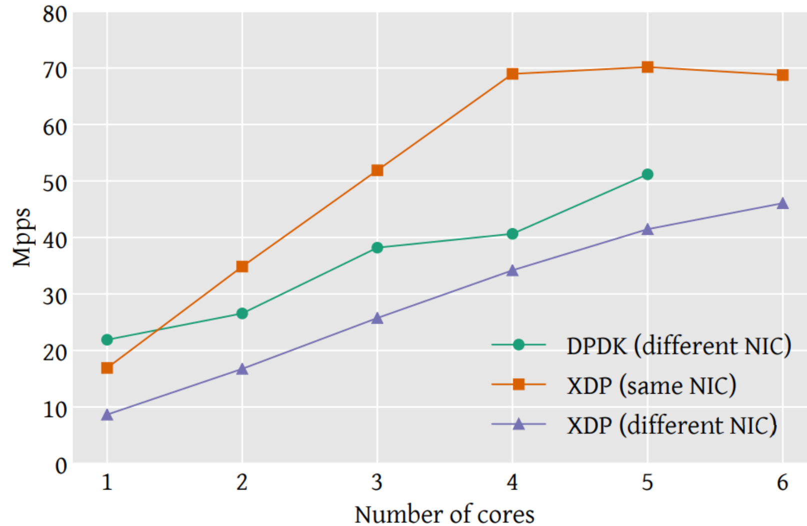


Figure 5: Packet forwarding throughput. Sending and receiving on the same interface takes up more bandwidth on the same PCI port, which means we hit the PCI bus limit at 70 Mpps.

source: The eXpress data path: fast programmable packet processing in the operating system kernel

- DPDK – best performance in drop packet scenario
- XDP forwarding w/ different NIC requires packet buffers allocation by device driver
- XDP does not match DPDK, but mostly due to lack of driver micro optimisations

Traffic Control - TC

- for both **egress** and **ingress** path
- **lower throughput** than XDP as it happens **later** in the stack
- has access to **entire Ethernet frame**, works at TC layer – queueing disciplines for packet queues, filters to allow/deny/modify packets
- **more** packet **information** compared to XDP (`struct __sk_buff`) as packet has been already parsed
- actions available:
  - TC_ACT_OK: deliver the packet in TC queue
  - TC_ACT_SHOT: drop packet
  - TC_ACT_UNSPEC: use standard TC action
  - TC_ACT_PIPE: perform next action (if it exists)
  - TC_ACT_RECLASSIFY: restart classification
- possible to use **both TC** and **XDP** at the same time: TC for TX and XDP for RX traffic

TC – XDP coop

```c
static inline void xdp_process_packet(struct xdp_md *xdp) {
void *data = (void *)(long)xdp->data;
void *data_end = (void *)(long)xdp->data_end;
    process_eth(data, data_end, data_end - data);
}

SEC("xdp")
int xdp_count_packets(struct xdp_md *xdp) {
xdp_process_packet(xdp);
    return XDP_PASS;
}
```

```c
static inline void tc_process_packet(struct
    __sk_buff *skb) {
void *data = (void *)(long)skb->data;
void *data_end = (void *)(long)skb->data_end;
    process_eth(data, data_end, skb->len);
}

SEC("tc")
int tc_count_packets(struct __sk_buff *skb) {
tc_process_packet(skb);
    return TC_ACT_UNSPEC;
}
```

Security concerns

eBPF security

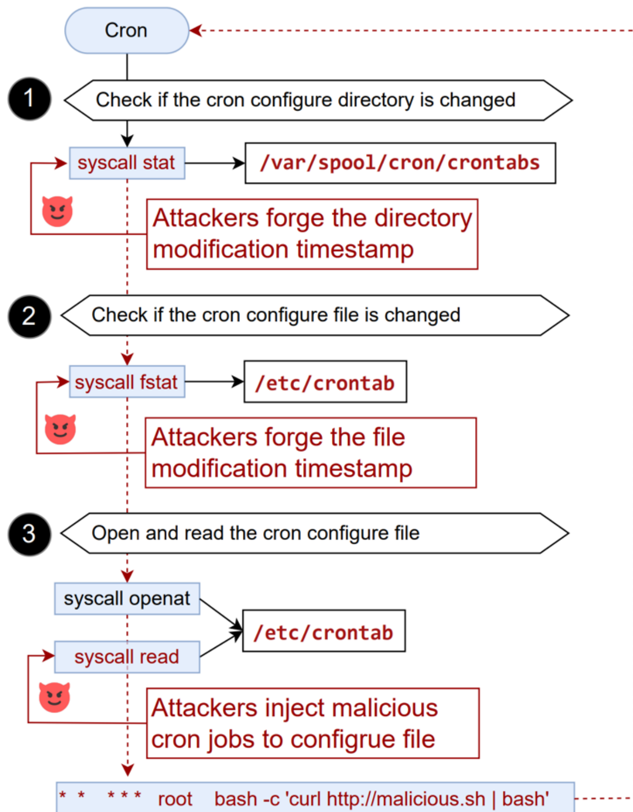
- programs can **monitor** (and **alter**) **processes** on the **whole** system, from host VM to other containers
- potential for **malware** and **rootkits**:
  - **manipulating** network **packets**
  - **hijacking processes** (execute malicious commands) and manipulating memory of processes - **bpf_probe_write_user** helper permits writing to memory of other processes
  - **DoS** against processes (terminating processes)
  - **stealing** sensitive **data** (reading memory and opened files)
  - modifying **syscalls' arguments** or **return code**
  - **container escape** – through hijacking privileged processes
  - **K8s exploitation** through abusing insecure Pods or Operator Service Accounts
  - attacks **stealthy** and **difficult** to **detect** (tracing/kprobes can defeat detector tools, possible to implement **C&C channels**)
- eBPF programs require CAP_SYS_ADMIN
  - a lot of **insecure containers** w/ such permissions (~2.5% of all Docker Hub containers)
  - frequently enabled in containers due to `mount` or some other dependancies

eBPF security (cont)

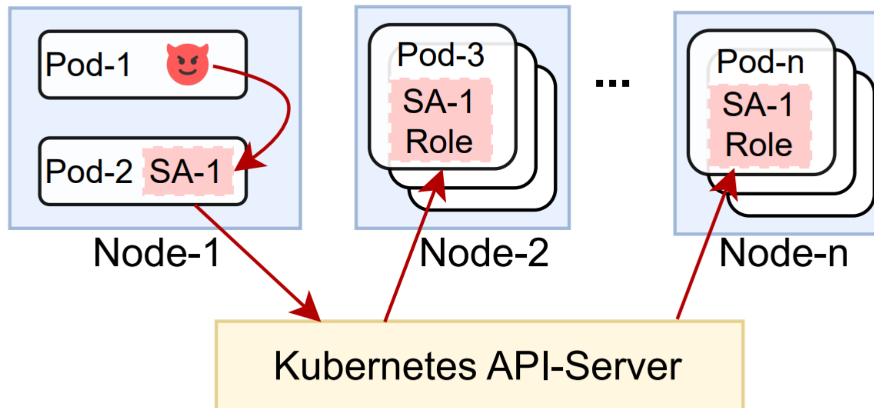
- lack of fine-grained ACLs for eBPF (only toggle on/off)
  - **CAP_BPF** is not a solution either (requires **CAP_PERFMON** and **CAP_NET_ADMIN** too)
- restricting eBPF w/ RBAC to trusted programs
  - **supply chain** attacks still possible due to large usage of eBPF in common tracing tools (Datadog, Falco, Tetragon, Inspektor, Pixie)

Attack Vector	ID	Description and Impact	Required eBPF Feature	Offensive Helpers					Victim Process
				H1	H2	H3	H4	H5	
Process/System DoS	D1	Killing processes by sending signal	eBPF Trace				✓		Any Process
	D2	Abusing LSM rules to crash processes	eBPF LSM						Any Process
	D3	Altering processes' syscall arguments or return code	eBPF Trace	✓		✓			Any Process
Information Theft	T1	Stealing processes' opened files	eBPF Trace		✓				Any Process
	T2	Stealing kernel data addresses to bypass KASLR	eBPF Trace		✓				-
Container Escape by Hijacking Processes	E1	Code reuse attacks (ROP) to hijack processes	eBPF Trace	✓	✓	✓			Any Process
	E2	Manipulating container's routine tasks	eBPF Trace	✓	✓	✓			Cron, Kubelet
	E3	Shellcode injection during mprotect syscall	eBPF Trace	✓	✓	✓			UPX/JIT
	E4	Forging credentials to login as root via SSH	eBPF Trace	✓	✓	✓			SSH
eBPF Map Tamper	M1	Altering other eBPF programs' maps to manipulate them	Any					✓	eBPF Program

eBPF security (cont)



ID	Helper Name	Functionality
H1	bpf_probe_write_user	Write any process's user space memory
H2	bpf_probe_read_user	Read any process's user space memory
H3	bpf_override_return	Alter return code of a kernel function
H4	bpf_send_signal	Send signal to kill any process
H5	bpf_map_get_fd_by_id	Obtain eBPF programs' eBPF maps fd



source: Cross Container Attacks: The Bewildered eBPF on Clouds

eBPF security (cont)

- Docker insecure setups:
  - `--privileged` flag
  - `--cap-add SYS_ADMIN`
  - exposing `docker.sock` to the container
- K8s insecure setups:
  - Pod Service Account files readable in `/var/run/secrets/kubernetes.io/serviceaccount/`
  - Operator Pods deployed together with Pods that host public services (steal SA, deploy malicious Pods)
  - possible to hijack host processes (cron, shell scripts, etc.)
- **virtualized containers** – help enforce security boundaries and reduce attack surface
- no solution for a fine-grained permission model (per program, etc.)
- **bpfman**
  - software stack eBPF management, monitoring and access control

EOF

Recommended literature

- books:
  - Liz Rice: Learning eBPF (O'Reilly)
  - ebpf-go documentation
  - The Aya Book
- articles:
  - Vishal Patil: Oxidize eBPF: eBPF programming with Rust
  - Marcos A. M. Viera et al.: "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications"
  - Yi He and Roland Guo et al.: "Cross Container Attacks: The Bewildered eBPF on Clouds"
  - Niclas Hedam: "eBPF - From a Programmer's Perspective"
  - Toke Høiland-Jørgensen, et al.: "The eXpress data path: fast programmable packet processing in the operating system kernel"

- https://github.com/zoidyzoidzoid/awesome-ebpf
- https://github.com/gojue/ebpf-slide

Have more questions?

dkorunic@haproxy.com

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

TECH

EBPF: FEATURES, CAPABILITIES AND IMPLEMENTATION