KubeCon | CloudNativeCon

Europe 2021

Virtual

Forward Together »

KubeCon | CloudNativeCon

Europe 2021

Virtual

eBPF on the Rise

Getting Started

Quentin Monnet

Software Engineer at Isovalent

5th May 2021

A Technology to Watch… but Why?

CNCF
@CloudNativeFdn

#CNCF TOC chair @lizrice is sharing the 5 technologies to watch in 2021 according to the TOC:
1. Chaos engineering
2. @kubernetesio for the edge
3. Service mesh
4. Web assembly and eBPF
5. Developer + operator experience

7:04 PM · Nov 20, 2020

212 89 Share this Tweet

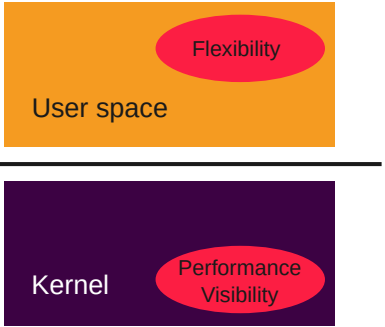
CNCF: Cloud Native Computing Foundation | TOC: Technical Oversight Committee

In-Kernel, Safe and Flexible Programs

Linux kernel / user space “paradox”

- **Kernel**: System awareness, but lacks flexibility
- **User space**: Programmable, but no direct access to kernel structures, resources
- Kernel modules: Difficult, unsafe, not stable

Kernel components are well-bounded frameworks



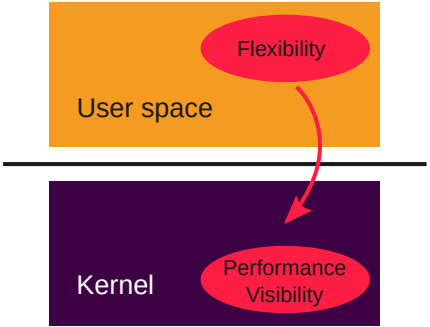
In-Kernel, Safe and Flexible Programs

Linux kernel / user space “paradox”

- **Kernel**: System awareness, but lacks flexibility
- **User space**: Programmable, but no direct access to kernel structures, resources
- Kernel modules: Difficult, unsafe, not stable

Kernel components are well-bounded frameworks
Get out of the box:

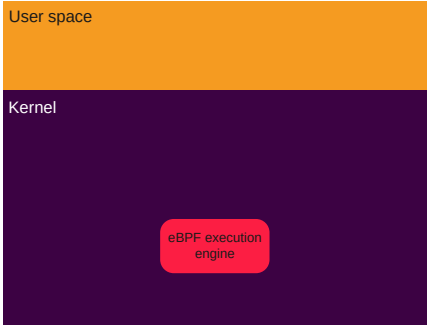
- **Can we have programmability in the kernel?**
- **How can this benefit to cloud-native environments?**



Meet eBPF

extended Berkeley Packet Filter

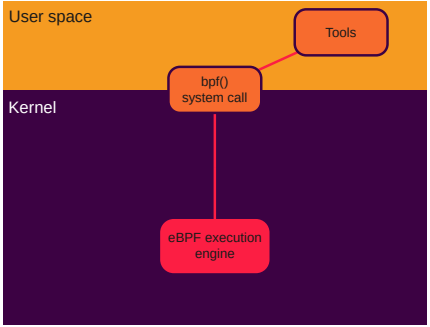
eBPF is a **general purpose execution engine**



extended Berkeley Packet Filter

eBPF is a **general purpose execution engine**

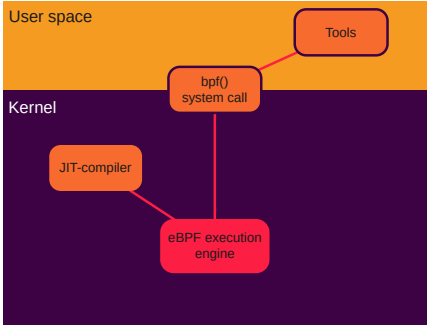
- **Kernel space**
  - Rework from cBPF
    (1990s: tcpdump, seccomp)
  - Bytecode injected with **`bpf()` syscall**
  - Program attached to **kernel hook**, runs on events



extended Berkeley Packet Filter

eBPF is a **general purpose execution engine**

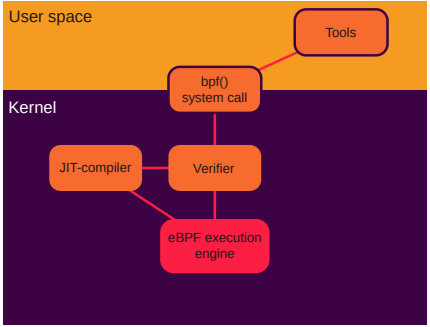
- **Kernel space**
  - Rework from cBPF
    (1990s: tcpdump, seccomp)
  - Bytecode injected with **`bpf()` syscall**
  - Program attached to **kernel hook**, runs on events
- **Performance**
  - In-kernel **JIT-compiler** (*Just In Time*)
  - Maps well to native code



extended Berkeley Packet Filter

eBPF is a **general purpose execution engine**

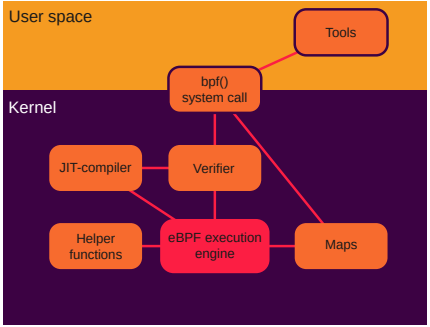
- **Kernel space**
  - Rework from cBPF (1990s: tcpdump, seccomp)
  - Bytecode injected with **`bpf()` syscall**
  - Program attached to **kernel hook**, runs on events
- **Performance**
  - In-kernel **JIT-compiler** (*Just In Time*)
  - Maps well to native code
- **Safety**
  - In-kernel **verifier** ensures **termination** and **safety**



extended Berkeley Packet Filter

eBPF is a **general purpose execution engine**

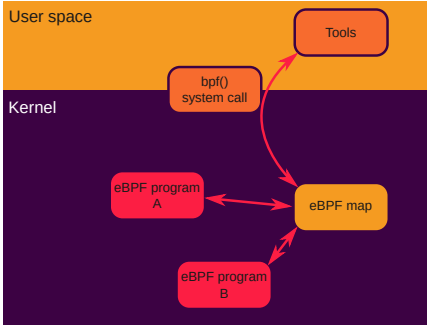
- **Kernel space**
  - Rework from cBPF (1990s: tcpdump, seccomp)
  - Bytecode injected with **`bpf()` syscall**
  - Program attached to **kernel hook**, runs on events
- **Performance**
  - In-kernel **JIT-compiler** (*Just In Time*)
  - Maps well to native code
- **Safety**
  - In-kernel **verifier** ensures **termination** and **safety**
- **Versatility**
  - Programs: 31 types, some with multiple hooks
  - Helper functions: 165
  - Maps: 30 types



Communicate With Maps

eBPF “Maps” are special kernel memory areas accessible to a program

- Typically, “key/value” storage: hash map, array
- Shared between
  - Several eBPF program runs
  - Several eBPF programs
  - eBPF and user space



A Powerful Execution Engine

eBPF programs support many features:

- Up to one million instructions
- Tail calls
- eBPF-to-eBPF function calls
- Calls to specific kernel functions
- Bounded loops
- BTF (*BPF Type Format*)
- Sleepable programs
- Spinlocks
- ...

Ever closer to “classic” code

Countless Use Cases

- **Networking**
  - Hooks: TC (*Traffic Control*), XDP (*eXpress Data Path*), sockets
  - Anti-DDoS
  - Load-Balancing
  - Routing, overlay, NAT
  - TCP control
- **Tracing & Monitoring**
  - Hooks: kprobes, uprobes, tracepoints, perf events
  - Inspect, trace, profile kernel or user space functions
  - Aggregate and correlate metrics in the kernel, return meaningful data
- Others
  - Security (LSM)
  - Infrared protocols
  - File systems, storage, ...

eBPF Tooling

LLVM Backend to the Rescue

The eBPF bytecode is usually generated with the **clang/LLVM backend**

Compile from C, store eBPF bytecode into an ELF object file:

```
$ clang -O2 -g -emit-llvm -c prog.c -o - | \
        llc -march=bpf -mcpu=v2 -filetype=obj -o prog.o
```

Example: Networking


```c
#include <arpa/inet.h>
#include <linux/bpf.h>
#include <linux/if_ether.h>

int block_non_ipv4(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data;

    /* Check packet length before dereferencing "eth" pointer */
    if (data + sizeof(*eth) > data_end)
        return XDP_DROP;

    /* Allow IPv4 packets, drop everything else */
    if (eth->h_proto == htons(ETH_P_IP))
        return XDP_PASS;

    return XDP_DROP;
}
```

- Compile with `clang`
- Load from the object file with `ip link set xdp`

Example: Tracing With BCC


```
from bcc import BPF

b = BPF(text="""
#include <uapi/linux/ptrace.h>

int trace_open(struct pt_regs *ctx, int dfd,
               const char __user *filename, int flags)
{
        u64 id = bpf_get_current_pid_tgid();
        u32 pid = id >> 32;

        bpf_trace_printk("%d: open(%s, %x)\\n", pid, filename, flags);

        return 0;
}
""")
b.attach_kprobe(event="do_sys_open", fn_name="trace_open").trace_print()
```

BCC

- Framework for eBPF tools
- Handles compilation (libllvm), provides Python wrappers
- Contains many examples

BCC Tools: Opensnoop

Trace usage of `open()` system call

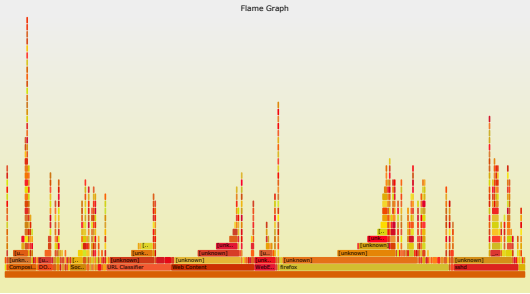
- Attach a kprobe and a kretprobe to `sys_do_open()`
- Kprobe stores command name, filename, fd in a map
- Kretprobe retrieves info from map and prints it, with return value

```
# ./opensnoop.py
PID    COMM       FD ERR PATH
1576   snmpd      11   0 /proc/sys/net/ipv6/neigh/lo/retrans_time_ms
1576   snmpd      11   0 /proc/sys/net/ipv6/conf/lo/forwarding
1576   snmpd      11   0 /proc/sys/net/ipv6/neigh/lo/base_reachable_time_ms
1576   snmpd       9   0 /proc/diskstats
1576   snmpd       9   0 /proc/stat
1576   snmpd       9   0 /proc/vmstat
1956   supervise   9   0 supervise/status.new
1956   supervise   9   0 supervise/status.new
17358  run         3   0 /etc/ld.so.cache
[...]
```

BCC Tools: CPU Profiling, Flame Graphs

Profile CPU usage: “flame graph” indicating how much time functions run

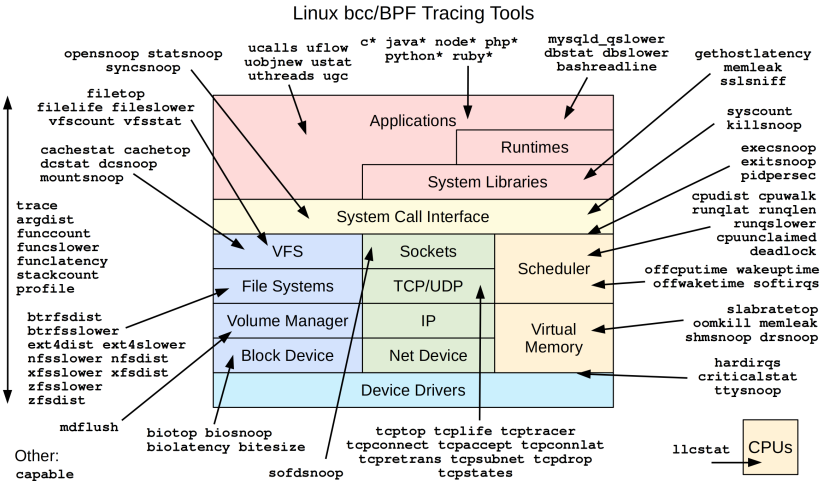
- Poll software perf event `CPU_CLOCK`, collect stack data
- Info and `flamegraph.pl` script at https://github.com/brendangregg/FlameGraph



```
# ./profile.py -f 10 > data.out
$ ./flamegraph.pl data.out > graph.svg
```

Also usable for Python stack, Ruby, PHP, C*, Java, Node.js, …

BCC Tools (2019)



https://github.com/iovisor/bcc#tools 2019

Credits: Brendan Gregg

Bpftrace for Powerful One-Liners

bpftrace, built on top of BCC

- Awk-inspired syntax, one-liners or short scripts
- “Linux equivalent to DTrace”

Usage

- `probe_type:probe_target /filter/ { command block }`
- Built-in variables and functions (handle maps, draw histograms, ...)

Bpftrace for Powerful One-Liners

bpftrace, built on top of BCC

- Awk-inspired syntax, one-liners or short scripts
- “Linux equivalent to DTrace”

Usage

- `probe_type:probe_target /filter/ { command block }`
- Built-in variables and functions (handle maps, draw histograms, ...)

Tracing `open()`

```
# bpftrace -e 'kprobe:do_sys_open { printf(”%d-%s: %s\n”, pid, comm, str(arg1)) }'
```

More examples

```
# Read size distribution by process, present results as an histogram
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Count LLC cache misses by process name and PID (uses PMCs)
bpftrace -e 'hardware:cache-misses:1000000 { @[comm, pid] = count(); }'
```

Build Your Own: Libraries

- C/C++: **Libbpf**, reference library
- Go: several libraries
  - **ebpf** from Cloudflare and Cilium: Pure Go library
  - libbpfgo: Wraps around libbpf
  - gobpf: Wraps around bcc
- Rust:
  - libbpf-rs: Wraps around libbpf
  - RedBPF: Wraps around bcc

Bpftool: Manage eBPF Objects

Bpftool to manage and inspect eBPF objects

Load a program

```
# bpftool prog load <program> <pinned_path>
```

List all BPF programs loaded on the system

```
# bpftool prog show
38: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2021-04-01T12:28:28+0100  uid 0
        xlated 64B  jited 61B  memlock 4096B
39: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2021-04-01T12:28:28+0100  uid 0
        xlated 64B  jited 61B  memlock 4096B
58: xdp  name process_packet  tag 6deef7357e7b4530  offloaded_to nfp_p1
        loaded_at 2021-04-01T21:37:12+0100  uid 0
        xlated 5848B  jited 14072B  memlock 8192B  map_ids 29,30
```

Bpftool: Inspect Programs

Dump eBPF bytecode

```
# bpftool prog dump xlated id 4
    0: (b7) r0 = 0
    1: (95) exit
```

Dump JIT-ed instructions

```
# bpftool prog dump jited id 4
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   sub    $0x28,%rsp
   b:   sub    $0x28,%rbp
   f:   mov    %rbx,0x0(%rbp)
  13:   mov    %r13,0x8(%rbp)
  [...]
  33:   mov    0x18(%rbp),%r15
  37:   add    $0x28,%rbp
  3b:   leaveq
  3c:   retq
```

Bpftool: Manage Maps

List all maps loaded on the system

```
# bpftool map show
64: array  name iterator.rodata  flags 0x480
        key 4B  value 98B  max_entries 1  memlock 4096B
        btf_id 235  frozen
2731: prog_array  name test_map  flags 0x0
        key 4B  value 4B  max_entries 4  memlock 4096B
        owner_prog_type tracing  owner jited
2768: array  name rules  flags 0x0
        key 4B  value 32B  max_entries 3  memlock 4096B
```

Lookup a map entry (full map dump also available)

```
# bpftool map lookup id 2768 key 0x01 0x00 0x00 0x00
key:
01 00 00 00
value:
11 02 00 40 8c a4 6f aa  8c 10 00 00 00 54 b7 f9
cc 71 d4 b1 89 b1 a7 9c  00 2a 5f 3d d6 85 45 f0
```

Update a map entry

```
# bpftool map update id 182 key 3 0 0 0 value 1 1 168 192
```

More Bpftool!

Test-run programs with user-defined input data and context

```
# bpftool prog run pinned /sys/fs/bpf/sample_ret0 data_in input data_out - repeat 10
0000000 0000 0000 0000 0000 0000 0000 0000 0000 | ........ ........
0000010 2e3e 2e48 656c 6c6f 4b75 6265 436f 6e21 | .>.Hello KubeCon!
Return value: 0, duration (average): 58ns
```

More features

- Attach programs (not all types)
- List programs per cgroup, per network interface, per tracing hook
- Probe system support for eBPF features
- Dump data from event maps

Packaged for several distributions, source code in kernel repository

Documentation: see man pages

eBPF for Cloud-Native Environments

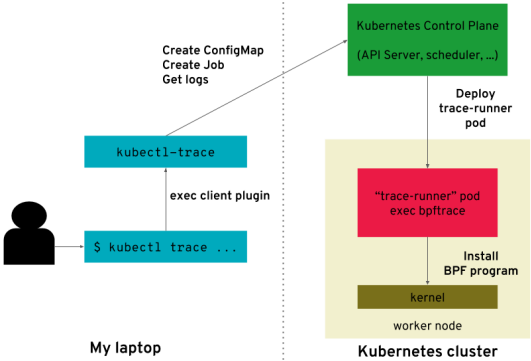
The Force is Strong With eBPF

- **Safety**, **performance**, **observability**, **versatility**
- In the **kernel**, but **flexible**
  - **Available** by default, no add-on required
  - **Stable UAPI**
  - **Updates**: no wait for upstream, no reboot, no loss of packet
- **Container-aware**
  - Multiple hooks
  - Kernel is the ideal location for managing containers
- Create what you need...
  - Don't just "program" a tool, "**create**"
  - Solve **real-world** production challenges
- ... Just what you need
  - Skip unnecessary features
  - Cleaner, faster, **scalable**

Linux kernel, base foundation for cloud-native environments: eBPF brings huge benefits!

Tracing Pods in a Kubernetes Cluster

Kubectl-trace to run bpftrace scripts on Pods



Credits: Lorenzo Fontana

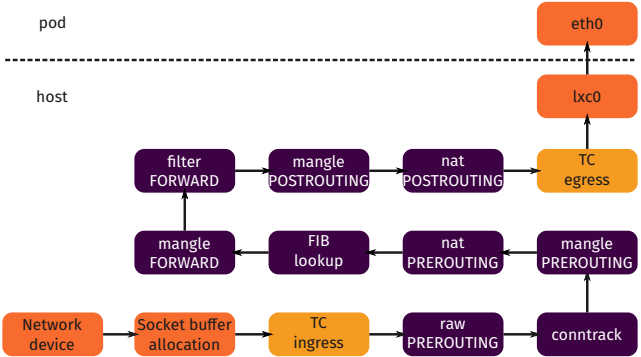
On the same model: Inspektor Gadget for BCC tools

Mastering Networks With Cilium

Cilium: *eBPF-based Networking, Observability, and Security*

Kube-proxy replacement

- Iptables: thousands of rules, linear search / eBPF: Hash map lookups
- Bypass Netfilter/conntrack entirely

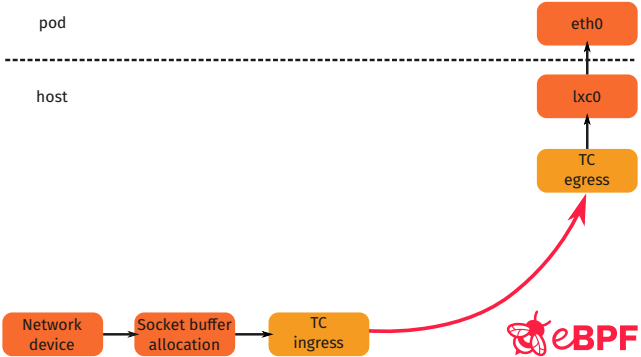


Mastering Networks With Cilium

Cilium: *eBPF-based Networking, Observability, and Security*

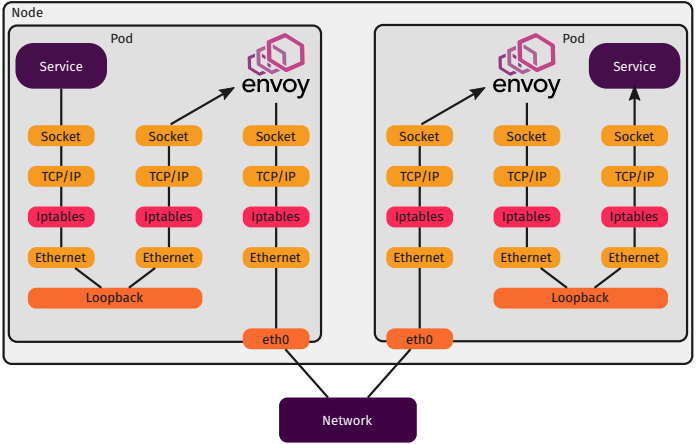
Kube-proxy replacement

- Iptables: thousands of rules, linear search / eBPF: Hash map lookups
- Bypass Netfilter/conntrack entirely



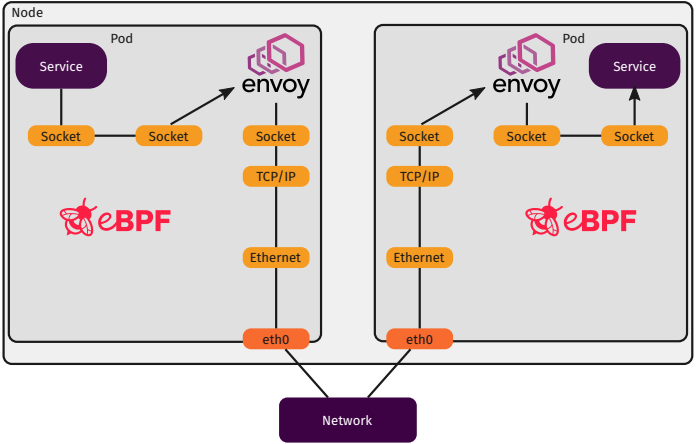
Cilium's Optimized Datapath

Example: Socket connection to Istio proxy for L7 policies



Cilium's Optimized Datapath

Example: Socket connection to Istio proxy for L7 policies



Cilium Use Cases for eBPF

Networking

- Highly efficient and flexible networking
- Routing, overlay, cloud-provider native
- IPv4, IPv6, NAT46
- Multi-cluster routing

Load balancing

- Highly scalable L3-L4 (XDP) load-balancing
- Kubernetes services (replaces kube-proxy)
- Multi-cluster
- Service affinity (prefer zones)

Network security

- Identity-based network security
- API-aware security (HTTP, gRPC, ...), DNS-aware
- Transparent encryption

Observability

- Metrics (network, DNS, security, latencies, HTTP, ...)
- Flow logs (with datapath aggregation)

Servicemesh

- Minimized overhead when injecting servicemesh sidecar proxies
- Istio integration

Credits: Daniel Borkmann

In the Wild

Large scale production users

Facebook, Netflix, Google, Cloudflare, Cilium, ...

Other projects rely on eBPF

Falco, Tracee, Hubble, Weave Scope, Suricata, ...
Falco probes were recently contributed to the CNCF

A Thriving Ecosystem

Increasing number of projects

Start-ups

- New start-ups for continuous profiling, network analytics, security
- Acquisitions
  - Pixie was acquired by New Relic
  - Flowmill was acquired by Splunk

Kernel community

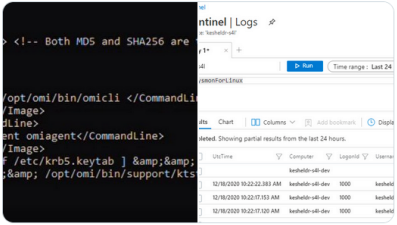
- One of the fastest growing subsystems in Linux
- Dedicated mailing list, 50 emails/day on average
- Three maintainers, five senior core reviewers (Facebook, Isovalent, Google)

First eBPF Summit (October 2020)

On the Rise

Mark Russinovich
@markrussinovich

We're working on eBPF-based Sysmon for Linux that has same filtering and output schema (where applicable) as Sysmon For Windows. Shooting for a preview in February.



7:16 PM · Dec 20, 2020

♡ 1.5K 💬 463 Share this Tweet

Steven Rostedt
@srostedt

BPF will replace Linux #kr2019

10:06 AM · Sep 26, 2019

♡ 87 💬 26 Share this Tweet

Wrapping Up

eBPF brings programmability to the kernel

- Safe, efficient, versatile, scalable
- Ideally located for gathering data or processing packets in cloud-native environments

eBPF tooling

- Tracing/monitoring: BCC, bpftrace
- Development: libbpf, Go libraries
- Introspection, management: bpftool

eBPF on the rise

- Solves real-world problems
  - See Cilium's datapath and network policies
- Big actors run eBPF in production at scale
- Buzzing community

Ride the eBPF wave!

Thank You!

eBPF cilium ISOVALENT

https://ebpf.io
https://cilium.io
quentin@isovalent.com

For more eBPF use cases, watch our other presentations:

- Tomorrow, 13:30 CEST (Networking)
  Uncovering a Sophisticated Kubernetes Attack in Real-Time – Jed Salazar & Natália Réka Ivánkó, Isovalent
- Tomorrow, 14:20 CEST (Security + Identity + Policy)
  How to Break your Kubernetes Cluster with Networking – Thomas Graf, Isovalent

Questions on eBPF after the Q&A session? Community Slack: https://ebpf.io/slack