eBPF - The Silent Platform Revolution from Cloud Native

Daniel Borkmann (Isovalent)

SIGCOMM 2023 / 1st Workshop on eBPF and Kernel Extensions

Agenda

1) What led us to eBPF and why is it crucial for cloud native
2) eBPF as a new type of software
3) The eBPF verifier and open problems
4) Use cases, ongoing developments and open problems
5) Outlook & how can the academic community contribute

1) What led us to eBPF and why is it crucial for cloud native

OS era pre eBPF

“One-size fits all” OS model - code changes must be generic to all type of workloads and not cause significant regressions

- From supercomputers to small embedded devices all run Linux

Kernel UAPI is the defacto standard for user space applications

Innovation in user space happens within that UAPI boundary (unless taking high cost of kernel bypass or custom kernel module)

Not double but triple edged-sword: stability vs flexibility vs performance

OS era pre eBPF

“One-size fits all” OS model - code changes must be generic to all type of workloads and not cause significant regressions

- From supercomputers to small embedded devices* all run Linux

Kernel UAPI is the defacto standard for user space applications

Innovation in user space happens within that UAPI boundary (unless taking high cost of kernel bypass or custom kernel module)

Not double but triple edged-sword: stability vs flexibility vs performance

*** Linux has made it to Mars**

[Ingenuity]

A look at networking ~10 years ago

Networking 2013: The Year of SDN

Software-defined networking and network virtualization grabbed the industry by the throat in 2013, dominating almost every conversation about the future of networking. The year also was marked by major M&A deals, the arrival of 802.11ac products, and the ongoing Huawei saga. Here, we take you back through some of the memorable moments in networking in 2013.

By **Marcia Savage**

DECEMBER 17, 2013

SDN products hitting production both in HW and SW world

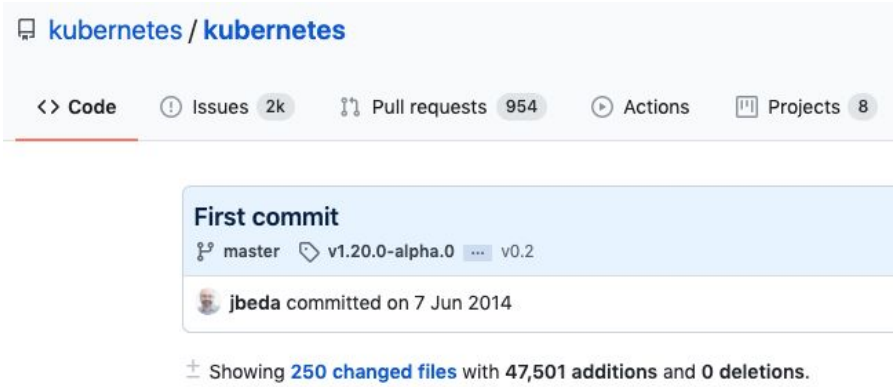
Open vSwitch, iptables, traffic control (“tc”) with limited capabilities for network programming

- Best option for more custom use cases was packet mark (skb → mark) or kernel module

Long developer-user feedback loop: 2+ years delta from upstream kernel to LTS / production

[The Year of SDN]

A look outside of networking ~10 years ago



openstack

docker

Emerge of compute platforms for public consumption utilising containers

GIFEE movement: Google Infrastructure for Everyone Else

- Kubernetes, TensorFlow, gRPC, Prometheus, Bazel, etc
- API-driven building blocks for solving problems at scale

Since then development shift towards “Cloud Native”

“Cloud Native” environment - what are the needs?

Scale
- Increasing density of deployments, nodes, clusters for better resource utilisation
- Distributed and loosely coupled, drives the need for performance and efficiency

Churn
- Handling of metadata association as Pods/containers come and go
- Network policies follow the Pods as they are instantiated
- IP addresses are no longer a stable identity

Day 2 operations
- Reliability of components, consistency, self-healing
- Ability to observe the state of the overall dynamic system

[CNCF Cloud Native Definition v1.0]

Where did this trend leave the OS kernel?

“The Good”: Linux kernel is the common denominator, but ...

“The Bad”: Tools developed ~20 years ago are not API driven, old kernel infrastructure developed back then not scalable for cloud native needs anymore

“The Ugly”: OS kernel bypass solutions were hyped back then, because the kernel “cannot keep up” with certain use cases or hardware anymore

Constrained in kernel UAPI and inflexibility, innovation moved further up the stack (e.g. DPDK).

[e.g. netfilter archeology]

The problem with kernel bypasses

The rise of cloud native approaches to building scalable distributed systems has put the industry at a critical juncture:

- Bypasses are in conflict with scalability, flexibility and multi-tenancy
- Often fine-tuning on per-host basis required → hard to maintain at scale given the need for uniform servers
- Kubernetes and containers standardise the Linux socket API as the common communication bus → bypasses require rewriting existing applications

Can we challenge the status quo and provide a strong foundation within the OS layer itself?

The answer: Programmability within the OS layer. Why?

Accelerate the speed of innovation, experimentation and research

Provide the technology to enable developers to deploy with unprecedented speed and scale

The answer: Programmability within the OS layer. Why?

Accelerate the speed of innovation, experimentation and research

Provide the technology to enable developers to deploy with unprecedented speed and scale

(= eBPF's mission)

2) eBPF as a new type of software

How to get there?

eBPF: a universal assembly language

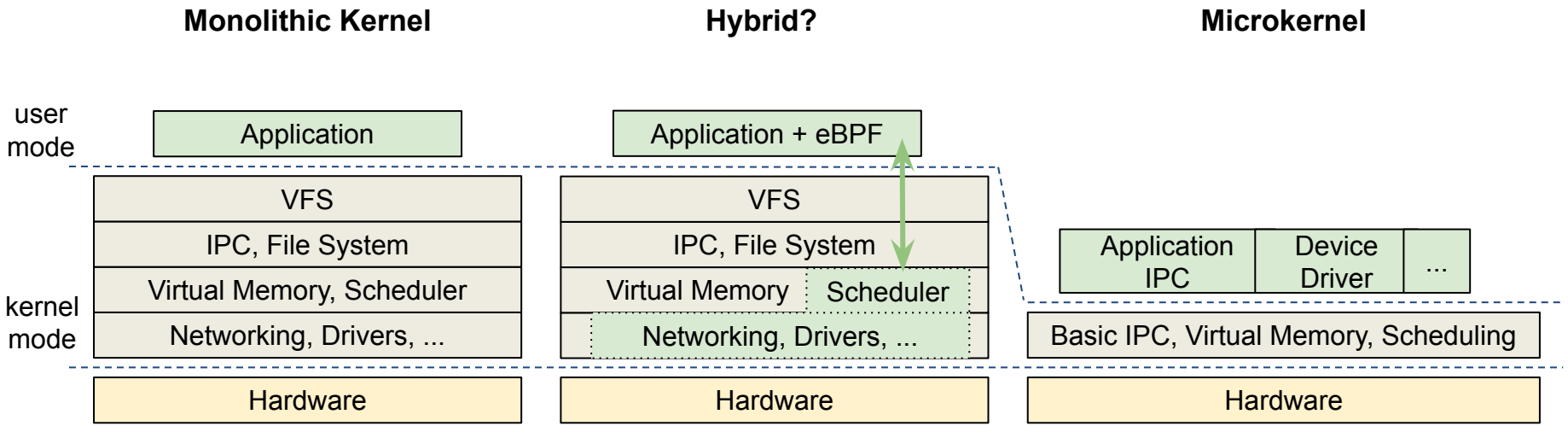
A strictly typed assembly language

Safe to run inside the kernel and for hardware

Stable instruction set

Extensions are backwards compatible

eBPF as new type of software



eBPF as new type of software

Linus Benedict Torvalds Jan 30, 1992, 12:14:26 AM

In article <12...@star.cs.vu.nl> a...@cs.vu.nl (Andy Tanenbaum) writes:
>
>1. MICROKERNEL VS MONOLITHIC SYSTEM

True, linux is monolithic, and I agree that microkernels are nicer. With a less argumentative subject, I'd probably have agreed with most of what you said. From a theoretical (and aesthetical) standpoint linux looses. If the GNU kernel had been ready last spring, I'd not have bothered to even start my project: the fact is that it wasn't and still isn't. Linux wins heavily on points of being available now.

[Tanenbaum vs Torvalds]

eBPF as new type of software

Linus Benedict Torvalds

"[e]BPF has actually been really useful, and the real power of it is how it allows people to do specialized code that isn't enabled until asked for."

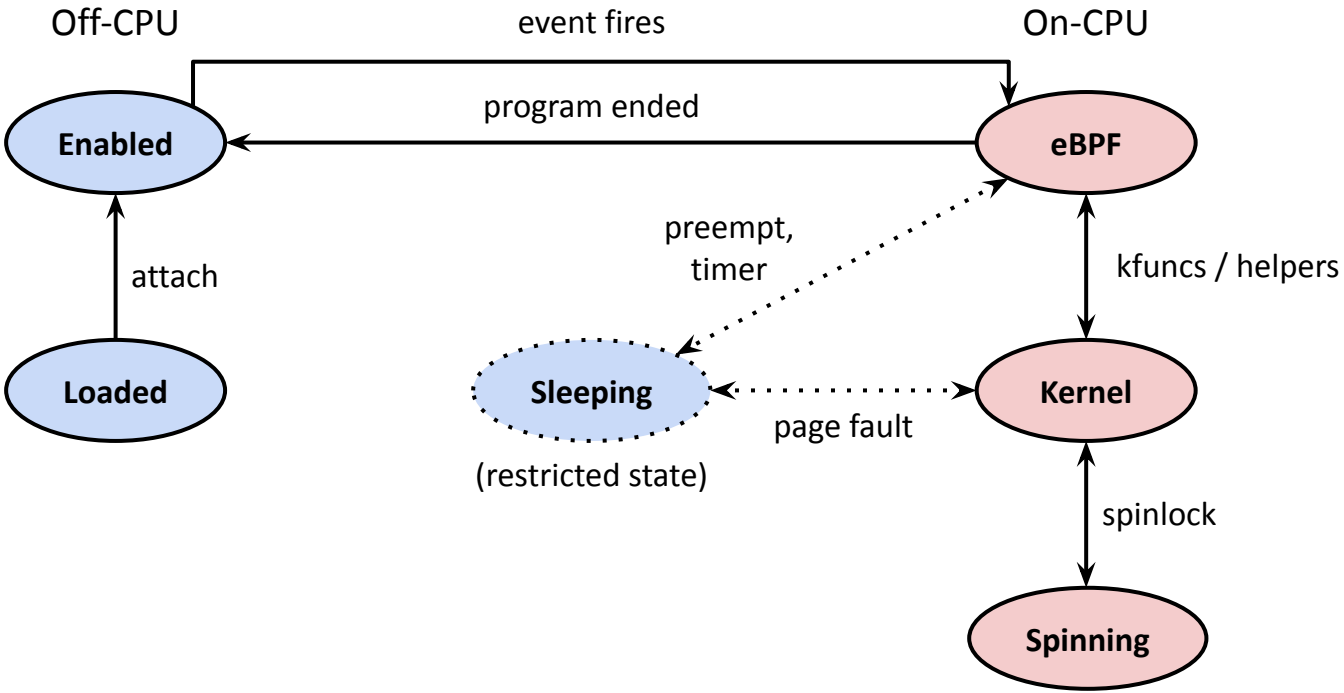
[Interview: Linus Talks about Coming Back to Work]

eBPF as new type of software

	Execution model	User defined	Compilation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none (code reviews)	panic	direct
eBPF	event	yes	JIT, CO-RE	verifier, JIT	error message	restricted helpers, kfuncs

[extended from Gregg, BPF Internals, LISA'21]

eBPF program state model



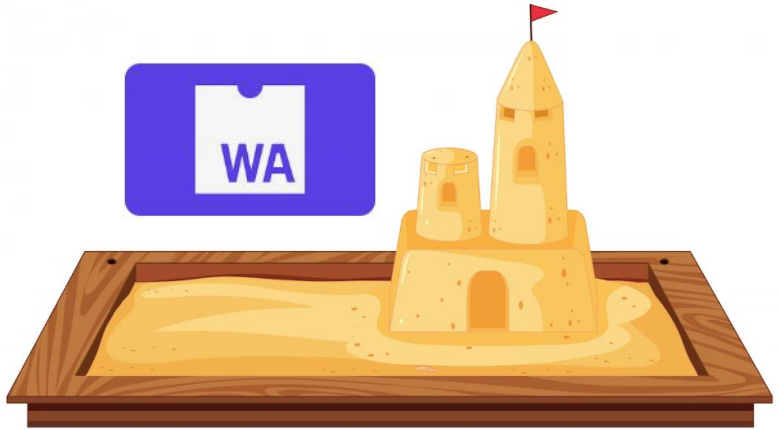
[extended from Gregg, BPF Internals, LISA'21]

Related work: eBPF compared to sandboxing (Wasm, etc)

eBPF program is understood before execution (and determines that the program is safe to run in a trusted environment).

Sandbox restricts execution environment - it doesn't understand what's running inside the sandbox.

eBPF is not a sandbox!



[image: by brgfx on Freepik]

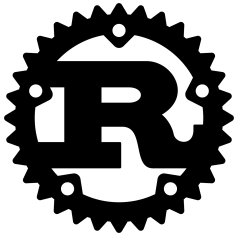
Related work: eBPF compared to P4

Both P4 and eBPF allow for network programmability. P4 is suited towards network hardware such as switches and eBPF targets general purpose CPUs instead of network hardware architectures.

The scope of eBPF is much broader and expands beyond networking into tracing, security, scheduling, etc. P4 is a DSL - eBPF is an assembly language. P4 compiler front ends exist which compile to eBPF.

P4 language is not safe in any sort of verified way, but rather the language is constrained so that one should not be able to express/compile things which would “crash” the system.

Related work: eBPF compared to Rust in the kernel



eBPF is tightly coupled with applications *and* the trusted environment.

The application often acts as the control plane, the trusted environment where the eBPF program runs as the the data plane.

Rust is targeted to improve kernel components and focuses on replacing code which often received less scrutiny such as device drivers.

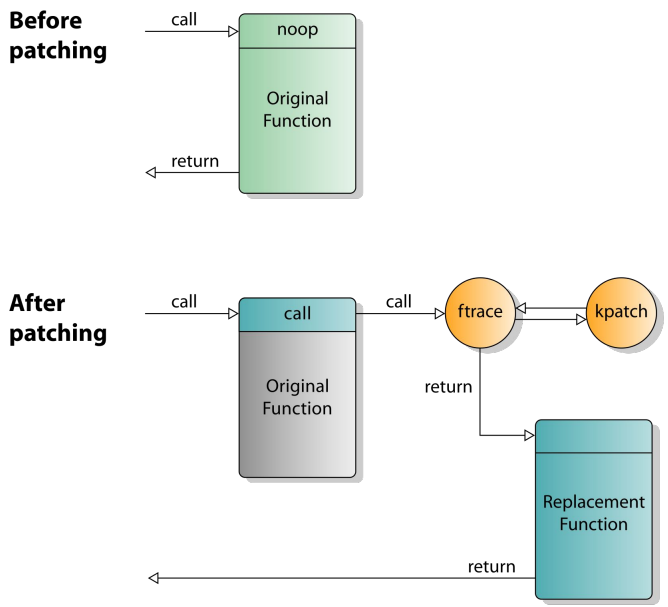
Both efforts do not compete, but rather complement each other.

Related work: eBPF compared to Kernel Live Patching (KLP)

Both modify kernel behavior at runtime, and can attach a replacement function.

KLP is focussed on critical and important security patches and has no user space control plane. The patchlet is standalone, short-lived and does not go through a verifier for safety checks.

eBPF scope is broader and often tightly coupled with a control plane application. eBPF does not have unlimited access to kernel internals.



Navigating stability versus flexibility in eBPF

We cannot predict all future requirements

- Need strong enough fundamentals to support future use cases
- Not so opinionated that we limit the solution space

Navigating stability versus flexibility in eBPF

We cannot predict all future requirements

- Need strong enough fundamentals to support future use cases
- Not so opinionated that we limit the solution space

Mature the concepts and interactions with the model

- Ecosystem libraries are maturing for a range of languages
- Verifier and language extensions to support intentional safety
- Better lifecycle and ownership handling with consistent concepts (e.g. BPF links)
- Consistent multi-program attachment points (e.g. BPF mprog)

Navigating stability versus flexibility in eBPF

We cannot predict all future requirements

- Need strong enough fundamentals to support future use cases
- Not so opinionated that we limit the solution space

Mature the concepts and interactions with the model

- Ecosystem libraries are maturing for a range of languages
- Verifier and language extensions to support intentional safety
- Better lifecycle and ownership handling with consistent concepts (e.g. BPF links)
- Consistent multi-program attachment points (e.g. BPF mprog)

Critical point: Where the model meets kernel data and kernel code

- BPF helpers and transition to now BPF kfuncs
- BPF kptrs for access into kernel data structures
- More minimalistic BPF UAPI with “to be stabilized” components

Innovation requires a strong foundation

IETF (e)BPF working group has been created in July 2023 to document the current state

- BPF instructions set architecture (ISA)
- Verifier expectations and building blocks for allowing safe execution [... of BPF]
- BPF Type Format (BTF)
- [...] conventions and guidelines for producing portable BPF program binaries
- Cross-platform BPF map types allowing native data structure access
- Cross-platform BPF helper types [...]
- Cross-platform BPF program types [...]
- Reference architecture and framework document

Innovation requires a strong foundation

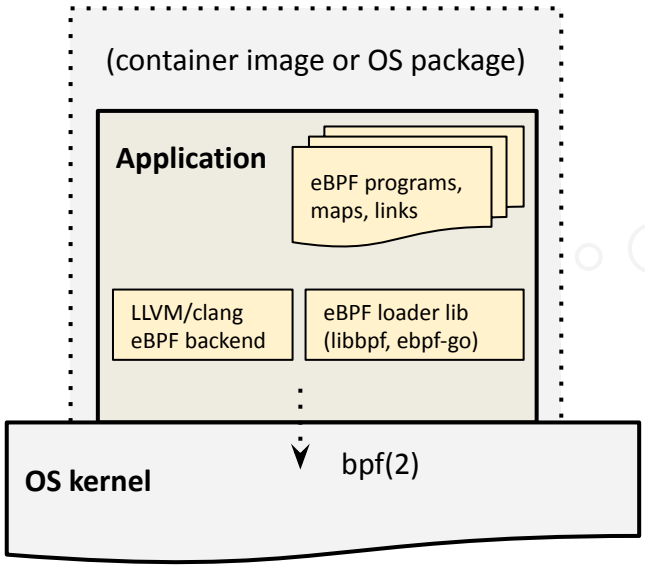
Standardisation effort helps the eBPF ecosystem to grow in multiple areas:

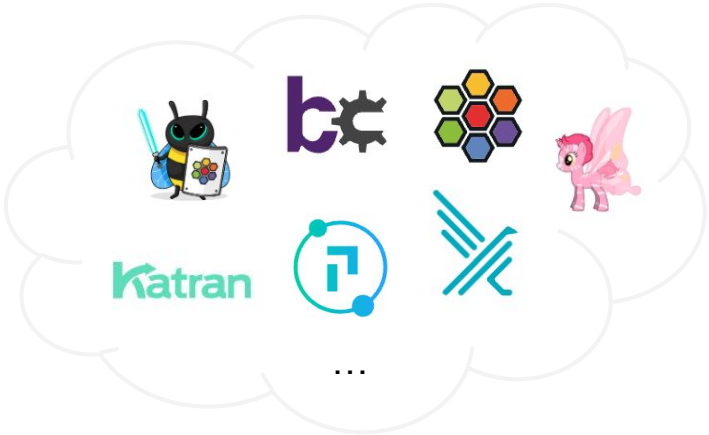
- Consistency and better coordination for LLVM and gcc's eBPF backends
- Hardware vendors have a stable reference for building eBPF offloads (e.g. Android, NICs, NVMe)
- Cross-platform compatibility with eBPF for Windows
- Emerge of new use cases for eBPF as hardware assist e.g. in confidential computing

Thoughts on deployment models of eBPF

Most common approach for large projects:

- Tight integration into the “control plane” application

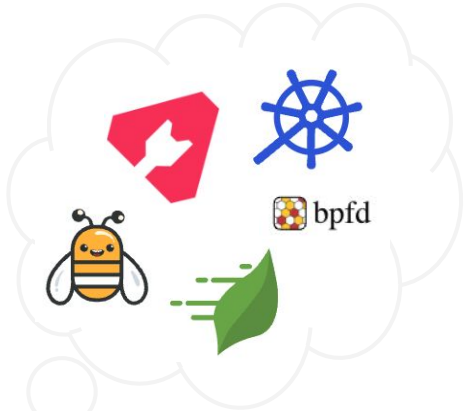


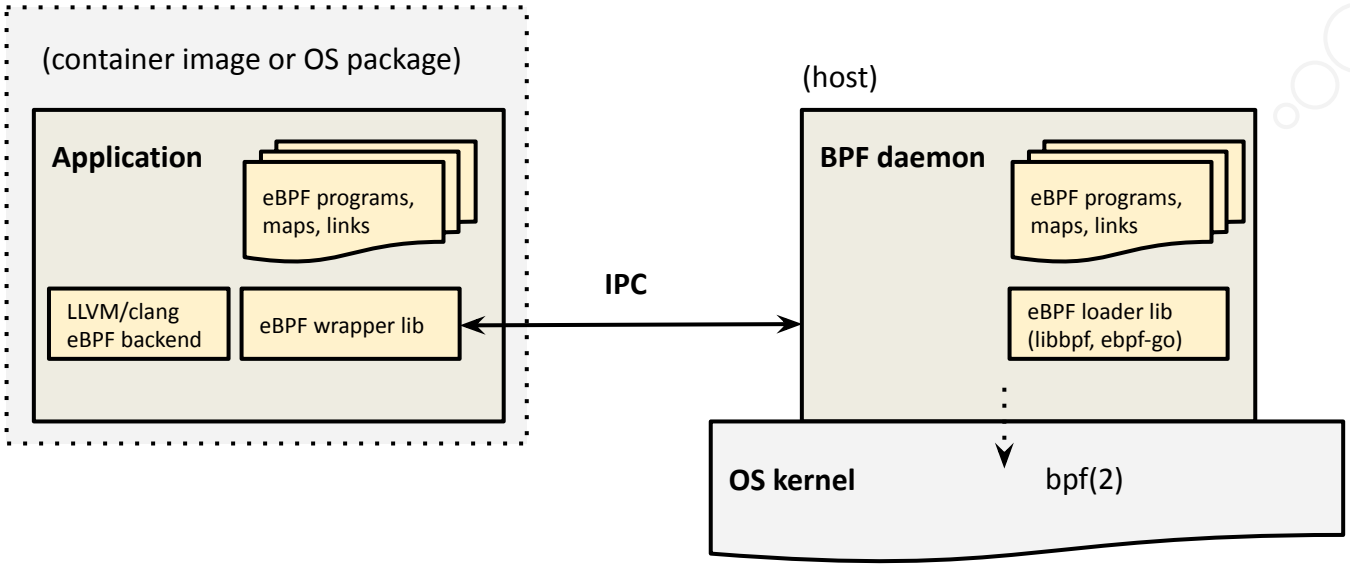


Thoughts on deployment models of eBPF

Alternative approaches:

- RPC between “control plane” application and BPF daemon for delegation





Thoughts on deployment models of eBPF

There is no general one-size-fits-all model: It depends on the use-case and production environment

Biggest “Con”s for tight integration model:

- Multi-user hooks need more coordination as there is no common “protocol” for resolving interdependencies between multiple applications
- Application needs access rights to bpf(2) syscall

Thoughts on deployment models of eBPF

There is no general one-size-fits-all model: It depends on the use-case and production environment

Biggest “Con”s for tight integration model:

- Multi-user hooks need more coordination as there is no common “protocol” for resolving interdependencies between multiple applications
- Application needs access rights to bpf(2) syscall

Biggest “Con”s for daemon delegation model:

- One more critical component in production which could fail and might be hard to debug
- Aside from kernel dependency new features also need daemon integration & deployment which limits flexibility to consume new features fast
- Support burden when loaders must assume both deployment models
- Dealing with atomic up- and downgrades of the application and daemon
- OS distributions / providers each come up with their own daemons

Thoughts on deployment models of eBPF

There is no general one-size-fits-all model

Any daemon based arbitration is not flexible enough to be a *generic* loader solution for all types of BPF programs.

Kernel side needs to ensure that there are enough mechanisms to suit the various needs and use-cases, but does not dictate one loader model over the other.

Mechanisms can range from: Capabilities, BPF token, BPF signing, LSMs, etc

Further discussion on production experience from Meta, Google, Cilium

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency
- Shorter production feedback loops for experimentation and research

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency
- Shorter production feedback loops for experimentation and research
- Largely decouples changing kernel behavior from underlying kernel

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency
- Shorter production feedback loops for experimentation and research
- Largely decouples changing kernel behavior from underlying kernel
- Avoids reinventing the wheel by utilising building blocks from kernel itself (e.g. FIB)

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency
- Shorter production feedback loops for experimentation and research
- Largely decouples changing kernel behavior from underlying kernel
- Avoids reinventing the wheel by utilising building blocks from kernel itself (e.g. FIB)
- Enables low-overhead, always-on deep visibility into the kernel

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud native infrastructure software development
  - Breaking long development cycles and therefore long developer/user feedback loop from patching the kernel
- Use case specific kernel awareness (e.g. Kubernetes metadata based security)
- Shifting data processing closer to the source and therefore significantly increasing efficiency
- Shorter production feedback loops for experimentation and research
- Largely decouples changing kernel behavior from underlying kernel
- Avoids reinventing the wheel by utilising building blocks from kernel itself (e.g. FIB)
- Enables low-overhead, always-on deep visibility into the kernel
- BPF flavor of C language as a safer choice for kernel programming

eBPF making the kernel a composable platform for cloud native

Problems eBPF solved today (extract):

- Significantly speeding up cloud n
  - Breaking long development
    from patching the kernel
- Use case specific kernel awarene
- Shifting data processing closer to
- Shorter production feedback loo
- Largely decouples changing kern
- Avoids reinventing the wheel by
- Enables low-overhead, always-on deep visibility into the kernel
- BPF flavor of C language as a safer choice for kernel programming

```c
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);
    return 0;
}
```

OK in C.
NOT OK in BPF C.

3) The eBPF verifier and open problems

eBPF verifier

Static code analyzer walking in-kernel copy of BPF program instructions

- ➔ **Ensuring program termination**
  - DFS traversal to check program is a DAG
  - Preventing *un*bounded loops
  - Preventing out-of-bounds or malformed jumps
- ➔ **Ensuring memory safety**
  - Preventing out-of-bounds memory access
  - Preventing use-after-free bugs and object leaks
  - Also mitigating vulnerabilities in the underlying hardware (Spectre)
- ➔ **Ensuring type safety**
  - Preventing type confusion bugs
  - BPF Type Format (BTF) for access to (kernel's) aggregate types
- ➔ **Preventing hardware exceptions (division by zero)**
  - For unknown scalars, instructions rewritten to follow aarch64 spec

eBPF verifier

Works by simulating execution of *all* paths of the program

- ➔ **Follows control flow graph**
  - For each instruction computes set of possible states (BPF register set & stack)
  - Performs safety checks (e.g. memory access) depending on current instruction
  - Register spill/fill tracking for program's private BPF stack
- ➔ **Back-edges in control flow graph**
  - Bounded loops by brute-force simulating all iterations up to a limit
- ➔ **Dealing with potentially large number of states**
  - Path pruning logic compares current state vs prior states
    - Current path "equivalent" to prior paths with safe exit?
  - Function-by-function verification for state reduction
  - On-demand scalar precision (back-)tracking for state reduction
  - Terminates with rejection upon surpassing "complexity" threshold

eBPF verifier

BPF register state tracking

BPF reg		
	type	u32
	id	u32
	off	s32
	var_off	tnum
	s64min	s64
	s64max	s64
	u64min	u64
	u64max	u64
	s32min	s32
	s32max	s32
	u32min	u32
	u32max	u32
	...	

uninit, scalar, ptr_to_* types. Types can be composable, e.g. or'ed with ptr_maybe_null.

Identifier for state propagation (e.g. learned bits from conditions)

Fixed part of pointer offset (pointer types only).

tnum		
	value	u64
	mask	u64

Represents knowledge of actual value for scalars (known and unknown bits).

Determined signed and unsigned 64 and 32-bit (sub-register) bounds.

Coupled to the var_off tnum, holding a lower and upper bound of the unknown value.

Used to determine if any memory access using this register will result in a bad access.

eBPF verifier

Short primer on BPF tristate numbers (tnums)

tnum	value	u64
	mask	u64

for each bit position P

P.value	P.mask	P.state
0	0	0
1	0	1
0	1	X
1	1	NaN

Example, 4 bit tnum:

010X ⟶ v = 0100, m = 0001
010X represents S { 0100, 0101 } ⟶ { **4, 5** }

XXXX ⟶ v = 0000, m = 1111
XXXX represents S { 0000, 0001, ..., 1111 } ⟶ { **0, 1, ..., 15** }

tnum and 64/32 bit min/max bounds relation:

Both needed, verifier propagates & refines knowledge between them.

Example state:

R ⟶ { 64 bit bounds [1, 0x77ffffff],
32 bit bounds [0, 0x7fffffff],
tnum v = 0, m = 0x77ffffff }

eBPF verifier

Short primer on BPF tristate numbers (tnums)

tnum	value	u64
	mask	u64

```
def tnum_add(tnum P, tnum Q):

   u64 sv := P.v + Q.v
   u64 sm := P.m + Q.m
   u64 Σ := sv + sm
   u64 χ := Σ ⊕ sv
   u64 η := χ | P.m | Q.m
   tnum R := tnum(sv & ~η, η)
   return R
```

Example, 4 bit tnum addition:

10X0 ⟶ v = 1000, m = 0010 ⟶ { **8, 10** }
\+ 10X1 ⟶ v = 1001, m = 0010 ⟶ { **9, 11** }
= 10XX1 ⟶ v = 10001, m = 00110 ⟶ { **17, 19, 21, 23** }

[Vishwanathan et al., Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers]

eBPF verifier

Short primer on BPF tristate numbers (tnums)

tnum	value	u64
	mask	u64

```
def  our_mul(tnum P, tnum Q):

   ACC_v := tnum(P.v * Q.v, 0)
   ACC_m := tnum(0, 0)

   while P.value or P.mask:
      # LSB of tnum P is a certain 1
      if (P.v[0] == 1) and (P.m[0] == 0):
        ACC_m := tnum_add(ACC_m, tnum(0, Q.m))
      # LSB of tnum P is uncertain
      else if  (P.m[0] == 1):
        ACC_m := tnum_add(ACC_m, tnum(0, Q.v|Q.m))
      # Note: no case for LSB is certain 0
      P := tnum_rshift(P, 1)
      Q := tnum_lshift(Q, 1)

   tnum R := tnum_add(ACC_v, ACC_m)
   return R
```

Example, 4 bit tnum multiplication:

X01 ⟶ v = 001, m = 100 ⟶ { **1, 5** }
* X10 ⟶ v = 010, m = 100 ⟶ { **2, 6** }
= XXX10 ⟶ v = 00010, m = 11100 ⟶ { **2, 6, 10, 14, 18, 22, 26, 30** }

[Vishwanathan et al., Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers]

eBPF verifier

Toy example

```
struct {
  uint8_t index;
  int32_t value;
  int32_t array[256];
} s;

s.array[s.index] = -s.value;
```

eBPF verifier

Toy example

```
struct {
  uint8_t index;
  int32_t value;
  int32_t array[256];
} s;

s.array[s.index] = -s.value;
```

BPF bytecode

```
; r0 points to s
r1 = *(u8*)(r0 + offsetof(s, index))
r2 = *(u32*)(r0 + offsetof(s, value))
r0 += offsetof(s, array)
r1 *= sizeof(int32_t)
r0 += r1
r2 = -r2
*(u32*)(r0) = r2
```

eBPF verifier

BPF bytecode:

```
; r0 points to s
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
```

```
; r1 ∈ {0, 4, 8, ..., 1020}
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1


r2 = -r2
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
; r2 [NO CONSTRAINTS]

; simplifies BPF verifier
```

eBPF verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1


r2 = -r2
*(u32*)(r0) = r2
```

; bpf_reg_state[]:

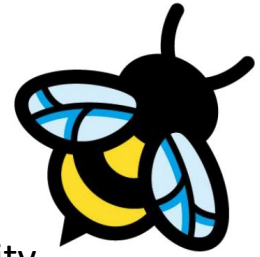
```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
; r2 [NO CONSTRAINTS]

; safe for all simulated r0 values
```

eBPF verifier and open problems

Challenges

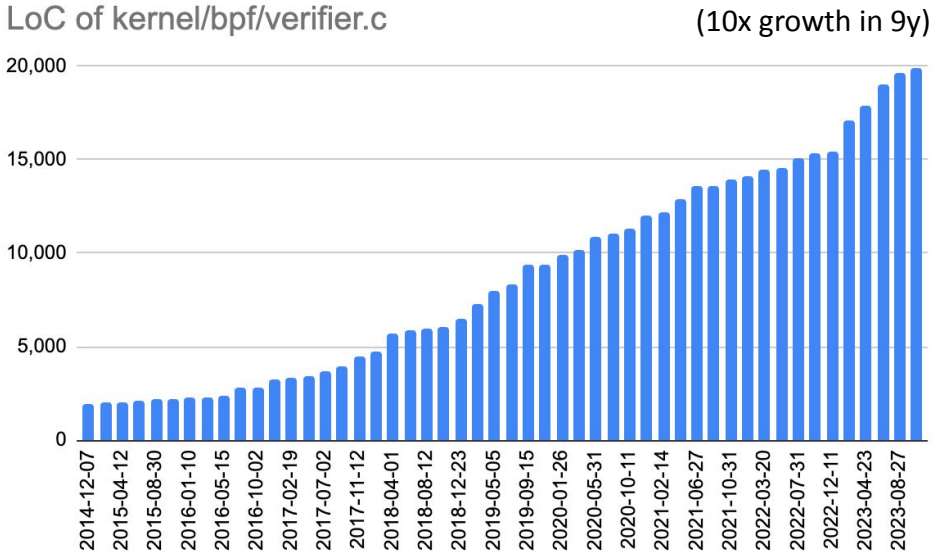
- ➔ **Attractive target for exploitation when exposed to non-root**
  - ◆ Growing verifier complexity
  - ◆ Programmability can be abused to bypass mitigations once in OS kernel
- ➔ **Reasoning about verifier correctness is non-trivial**
  - ◆ Especially Spectre mitigations
  - ◆ Only partial formal verification (e.g. tnums, JITs)
- ➔ **Occasions where valid programs get rejected**
  - ◆ LLVM/gcc vs verifier “disconnect” to understand optimizations
  - ◆ Restrictions when tracking state
- ➔ **“Stable ABI” for BPF program types (with some limitations)**
  - ◆ BPF programs in production should not break upon OS kernel upgrade
- ➔ **Performance vs security considerations**
  - ◆ Verification of complex programs must be efficient to be practical
  - ◆ Mitigations must be practical as performance of programs crucial

eBPF verifier and open problems



Challenges (cont)

- ➔ Allowing both 32-bit and 64-bit operations in BPF programs contributes to complexity
- ➔ Introspection into internal verifier state is limited (e.g. around path pruning decisions)
- ➔ Under active development to support new BPF features (e.g. exceptions)



eBPF verifier and open problems

More recent enhancements (extract):

- kfuncs and kptr infrastructure for user BPF objects
- Open-coded iterators to support loops
- mcpu=v4 instruction set extensions

```
struct bpf_iter_num it;
int *v;

bpf_iter_num_new(&it, 2, 5);
while ((v = bpf_iter_num_next(&it))) {
    bpf_printk("X = %d", *v);
}
bpf_iter_num_destroy(&it);
```

Longer-term objectives which must be tackled:

- Security and complexity of verifier
  - Still lacks a strong formal foundation as a whole
  - Catching code bugs and ensuring long-term maintainability
- Ease of use and less false positive program rejections
  - Allowing users to write more natural C to lower adoption barrier
  - Scalability with large programs containing many paths
- Unprivileged BPF

eBPF verifier and open problems

Unprivileged BPF:

- Currently disabled by default in all major distributions
- Not possible any time soon due to CPU HW vulnerabilities (Spectre & co)
- Mitigations for v1/v2/v4 implemented but severely limit flexibility and performance

Potential compromise is “trusted unprivileged BPF”:

- Why? Allowlist exceptions for trusted and verified production use cases
- Permissive LSM hooks: Rejected upstream given LSMs enforce further restrictions
- BPF token concept to delegate subset of BPF via token fd from trusted privileged daemon
- BPF signing as gatekeeper: application vs BPF program (no one-size-fits-all)
- RPC to privileged BPF daemon: Limitations depending on use cases/environment

4) Use cases*, ongoing developments and open problems

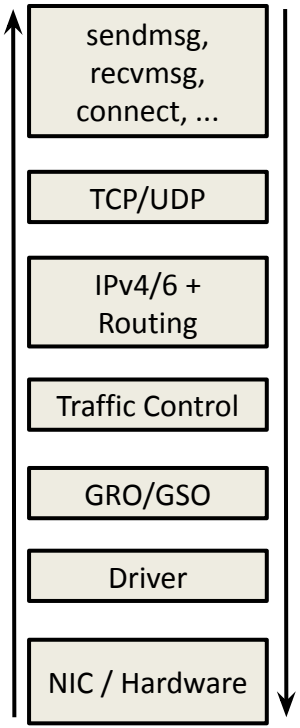
* just a small subset & not covering tracing due to time constraints

Networking & eBPF

Networking & eBPF

Linux networking stack in a nutshell

Next: Overview (extract) of different hooks & open problems



Networking & eBPF

XDP

General information:

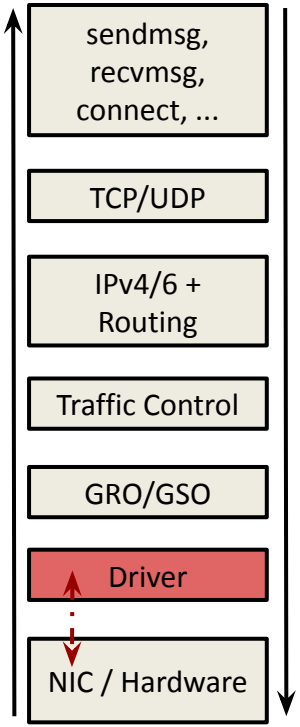
- Running inside the driver on ingress before the skb is formed
- Maximum speed, useful for L4LB, Gateways, DDoS scrubbing
- All relevant 10/40/100G+ drivers support it today
- Custom encap/decapsulation possible

Ongoing Improvements:

- Multi-buffer support for >4k MTUs
- XDP hints to utilise NIC offload capabilities with BPF

Open Problems:

- Pre-GRO, meaning, BPF program is run for every packet instead of batched
  - Data accessed in BPF program might not be in CPU cache yet in some cases
  - If co-located on cluster worker node, consider deferring host-local traffic to tcx layer early
- Support for multi-program attachment in XDP layer still on TODO
- Some drivers require ethtool RX/TX channel reconfiguration before use
- “Generic XDP” only for experimentation/CI as it is slow (turns off GRO, linearizes skb)



Networking & eBPF

tcx (& legacy tc cls_bpf)

General information:

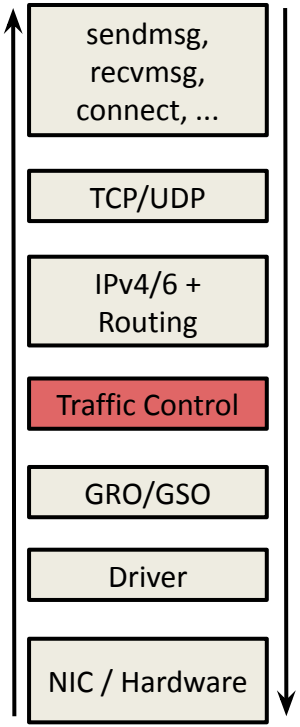
- Driver generic, running on ingress after GRO and egress before GSO
- Based on skb and therefore most feature-rich, flexible and fast hook
- Ideal for container networking (e.g. Cilium) or any kind of traffic which eventually terminates in (or originates from) a local socket - complementing XDP
- EDT for bandwidth shaping, full access to kernel fib, NAT46x64, …
- Can interoperate and exchange information with XDP programs

Recent Improvements:

- tcx recently merged as the future tc BPF core datapath
- Reusable bpf_mprog multi-attach layer for control dependencies

Open Problems:

- The kernel networking stack has a lot of protocol specific meta data encoded in the skb
  - Custom encapsulation might not work with GSO/TSO
- Harder to use than XDP with the various skb/checksum-specific helpers
- Full tc and XDP program offloading to hardware limited to just one vendor so far



Networking & eBPF

cgroup socket hooks

General information:

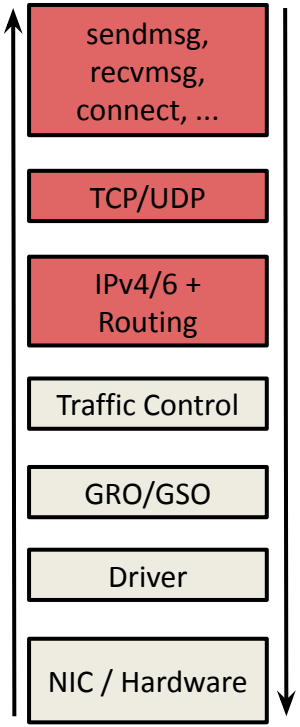
- sendmsg, recvmsg, connect, bind, get[sock,peer]name, [get,set]sockopt hooks for IPv[46] TCP and UDP to implement application-specific socket level policies
- Based on struct sock_addr / socket as input, no skb available
- Useful for implementing proxies or L4LBs for east/west cluster traffic (e.g. Cilium), complements XDP for north/south cluster traffic by moving processing closer to event source
- Moves per-packet NAT to connect-time NAT

Recent Improvements:

- In-kernel socket support such as nfs required special handling

Open Problems:

- Bypasses expectations from legacy infrastructure, e.g. some projects such as Istio install iptables rules into datapath which do not match service VIP given DNAT happened already at connect

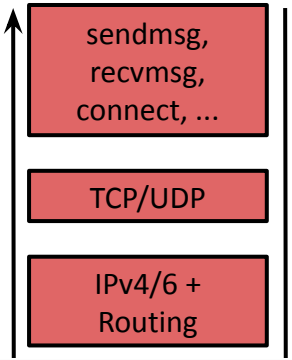


Networking & eBPF

TCP congestion control

General information:

- struct_ops programs to implement congestion control algorithms (e.g. DCTCP, BBR/BBRv2, new ones)
- Allows for very short feedback loop for experimentation in data center networks with atomic replacement during live traffic rather than deploying / rolling out new kernels
- Location aware / destination-specific congestion control algorithms with BPF




```
__u32 BPF_STRUCT_OPS(bictcp_recalc_ssthresh, struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        ca->epoch_start = 0;    /* end of epoch */

        /* Wmax and fast convergence */
        if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
                ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
                        / (2 * BICTCP_BETA_SCALE);
        else
                ca->last_max_cwnd = tp->snd_cwnd;

        return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

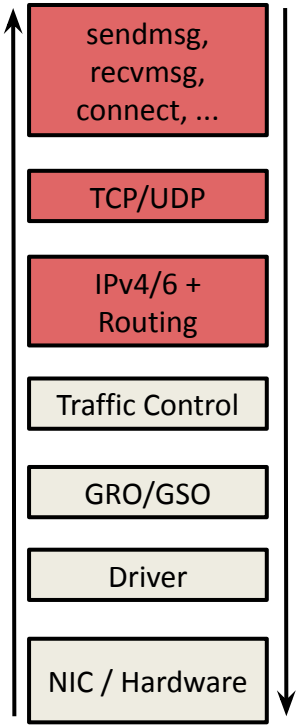
```
static u32 bictcp_recalc_ssthresh(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        ca->epoch_start = 0;    /* end of epoch */

        /* Wmax and fast convergence */
        if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
                ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
                        / (2 * BICTCP_BETA_SCALE);
        else
                ca->last_max_cwnd = tp->snd_cwnd;

        return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

Networking & eBPF

sk_lookup hook

General information:

- BPF program can pick a socket different from the skb's 5-tuple
- Attach scope is per network namespace
- Redirect to a listening socket in case of TCP, any socket for UDP
- Use case is to overcome limitations of bind(2) API e.g. binding socket to a subnet (x.y.z.0/24, port 80) and to steer packets destined to an IP on any port (a.b.c.d/32, any port) to a single socket



Networking & eBPF

sk_lookup hook

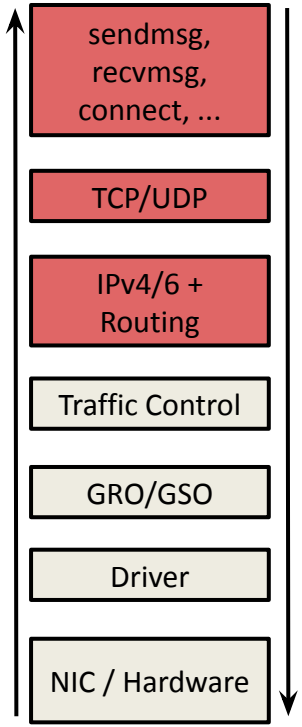
General information:

- BPF program can pick a socket different from the skb's 5-tuple
- Attach scope is per network namespace
- Redirect to a listening socket in case of TCP, any socket for UDP
- Use case is to overcome limitations of bind(2) API e.g. binding socket to a subnet (x.y.z.0/24, port 80) and to steer packets destined to an IP on any port (a.b.c.d/32, any port) to a single socket

sk_reuseport hook

General information:

- BPF program can pick a socket out of a reuseport socket group for better application scalability
- reuseport socket group contains socket bound to the same IP/port
- By default the kernel picks the socket based on the skb hash
- Use case is to silo packet processing to local CPUs/NUMA node for web server or proxies (Envoy) and for socket migration



Brief deep dive into recent developments

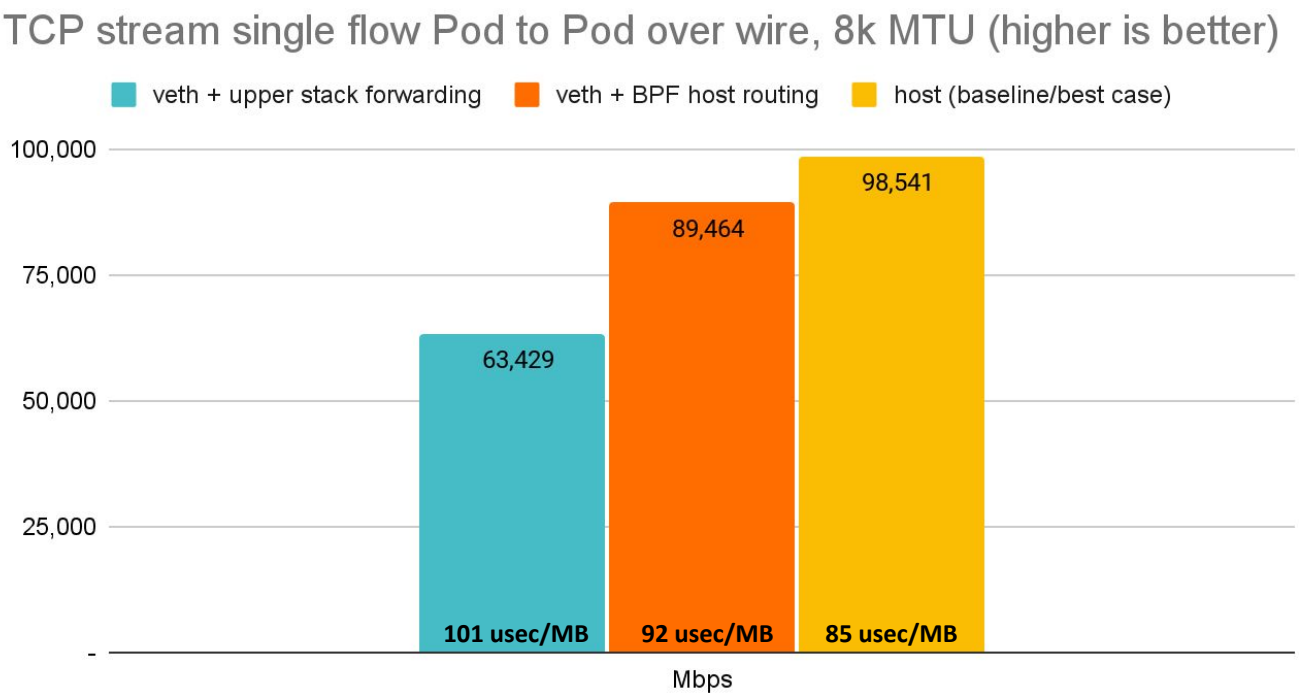
meta device for BPF

Why replacing veth driver for container networking?

- Achieve **same performance** for application inside Kubernetes Pod (netns) compared to application residing inside host namespace

- Just because we move applications into netns should not incur performance penalty, but it currently still does

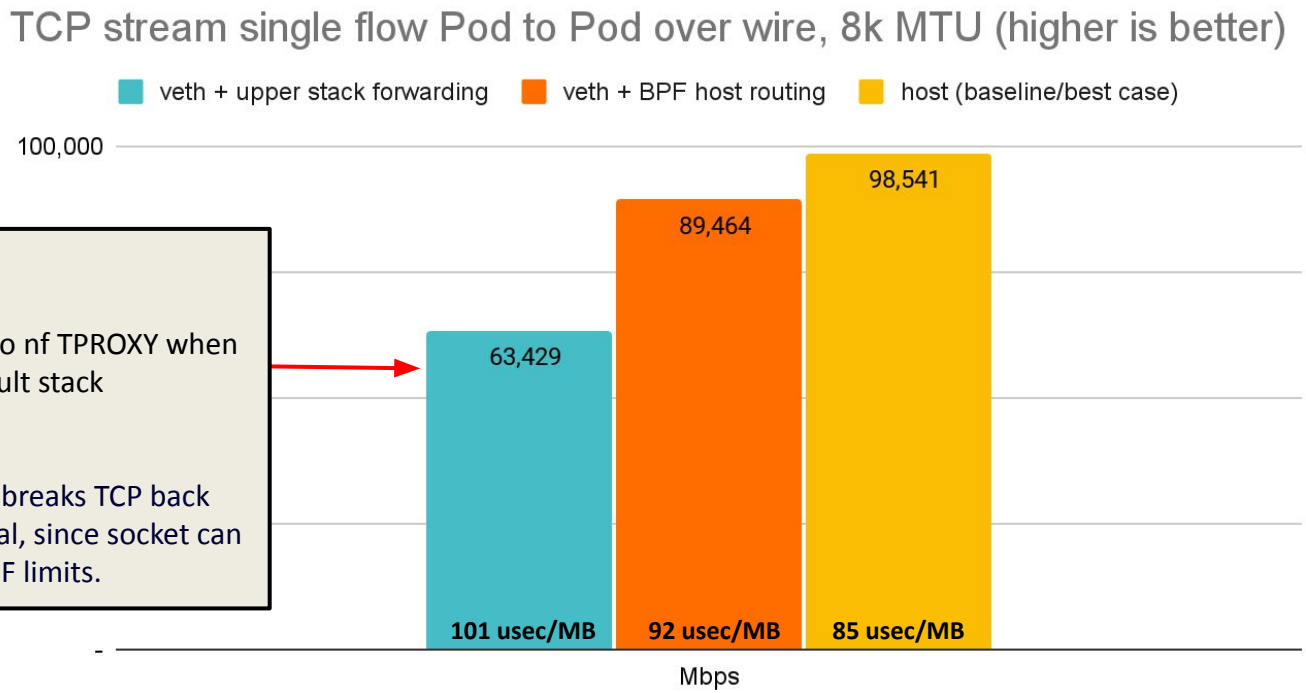
veth + stack vs BPF host routing case, results:



* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

veth + stack vs BPF host routing case, results:



Upper Stack:

skb_orphan due to nf TPROXY when packet takes default stack forwarding path.

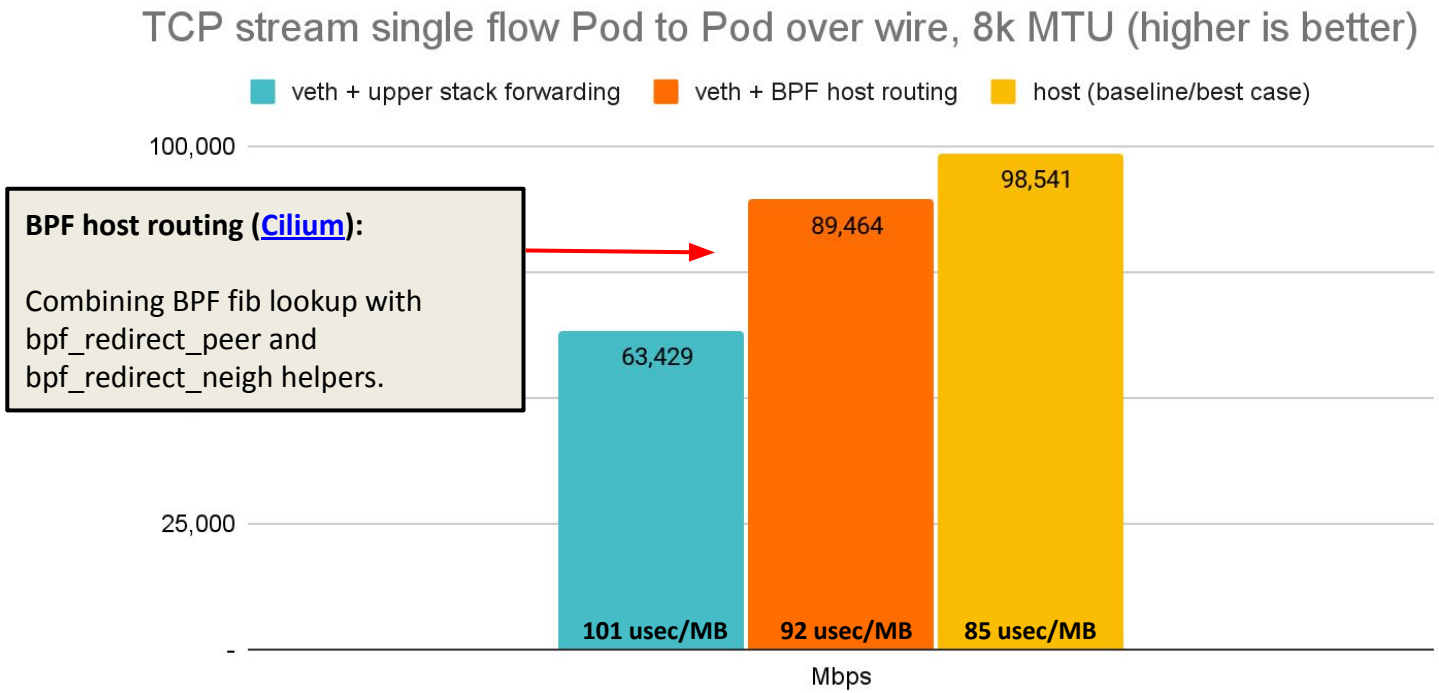
Doing it too soon breaks TCP back pressure in general, since socket can evade SO_SNDBUF limits.

* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU

Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

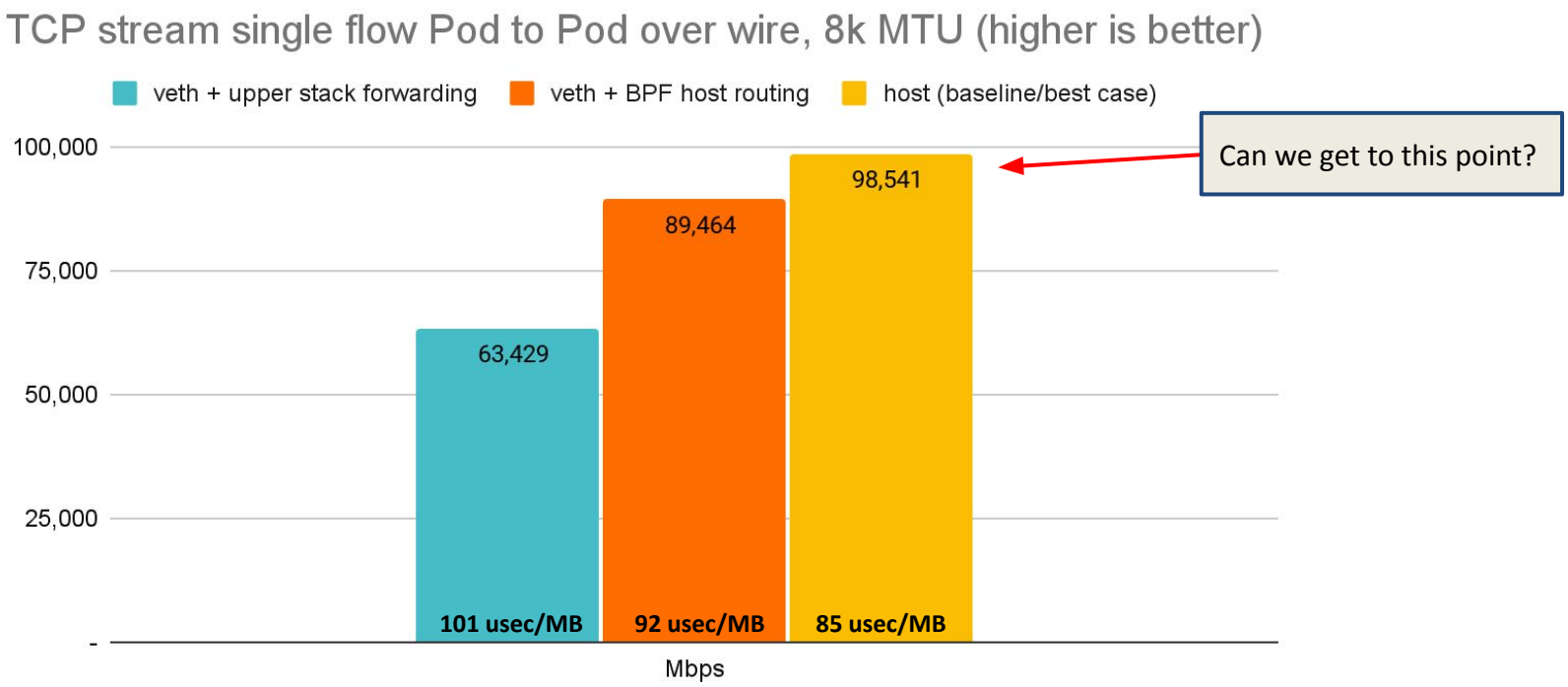
veth + stack vs BPF host routing case, results:



* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

veth + stack vs BPF host routing case, results:



* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

meta netdevices:

Core Idea:

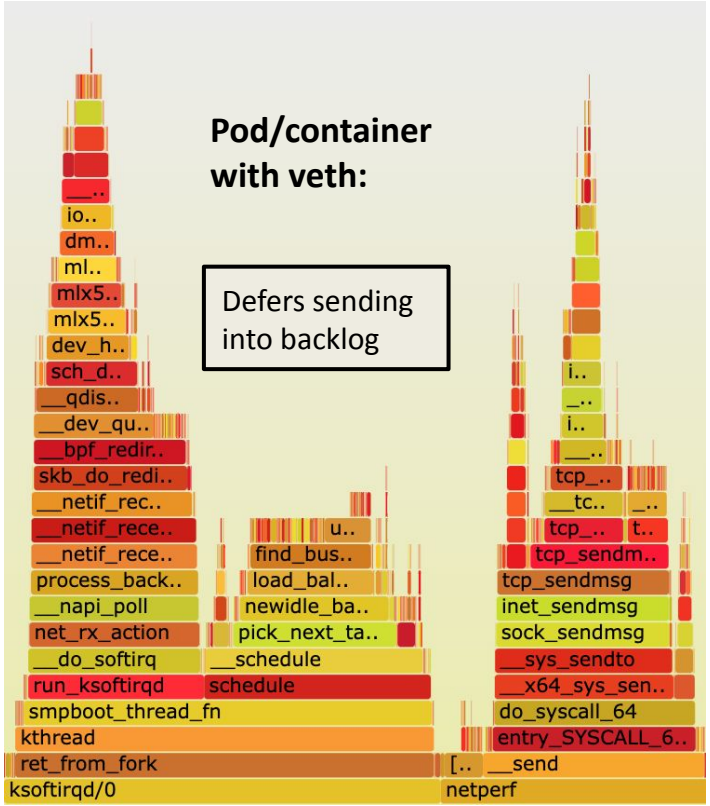
- BPF shifted from tcx (veth case) into the device driver, so BPF logic is part of device xmit itself
  - Program has the knowledge & ability to redirect from there to physical device

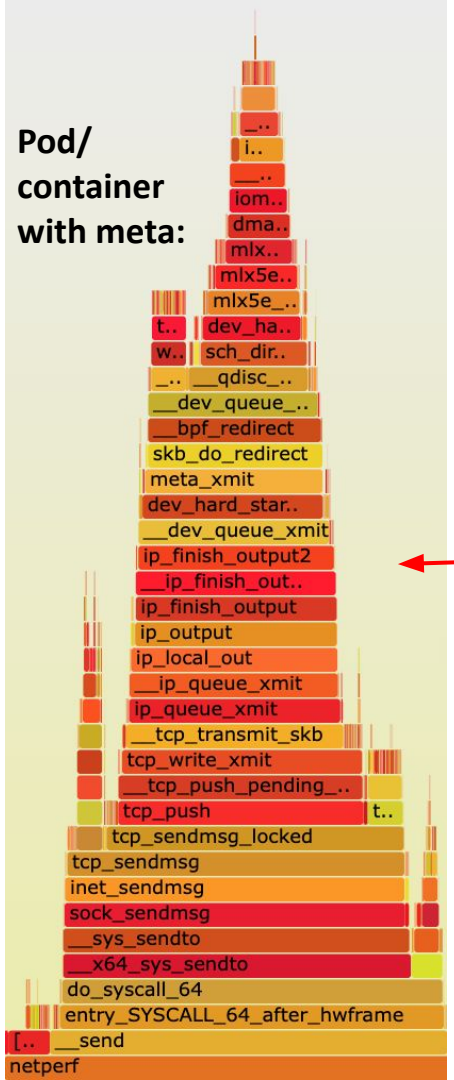
- Performance: no per-CPU backlog when BPF redirects traffic from Pod to outside the node
  - This shifts softirq time over to sys time: process scheduler can make better decisions
  - Reduces latency by simplifying transmit processing loop from two to one

- Program management: reuse of BPF multi-prog attach API (bpf_mprog)
- Operates in L3 or L2 mode with option to blackhole traffic when no BPF is attached

[PoC code for meta device]

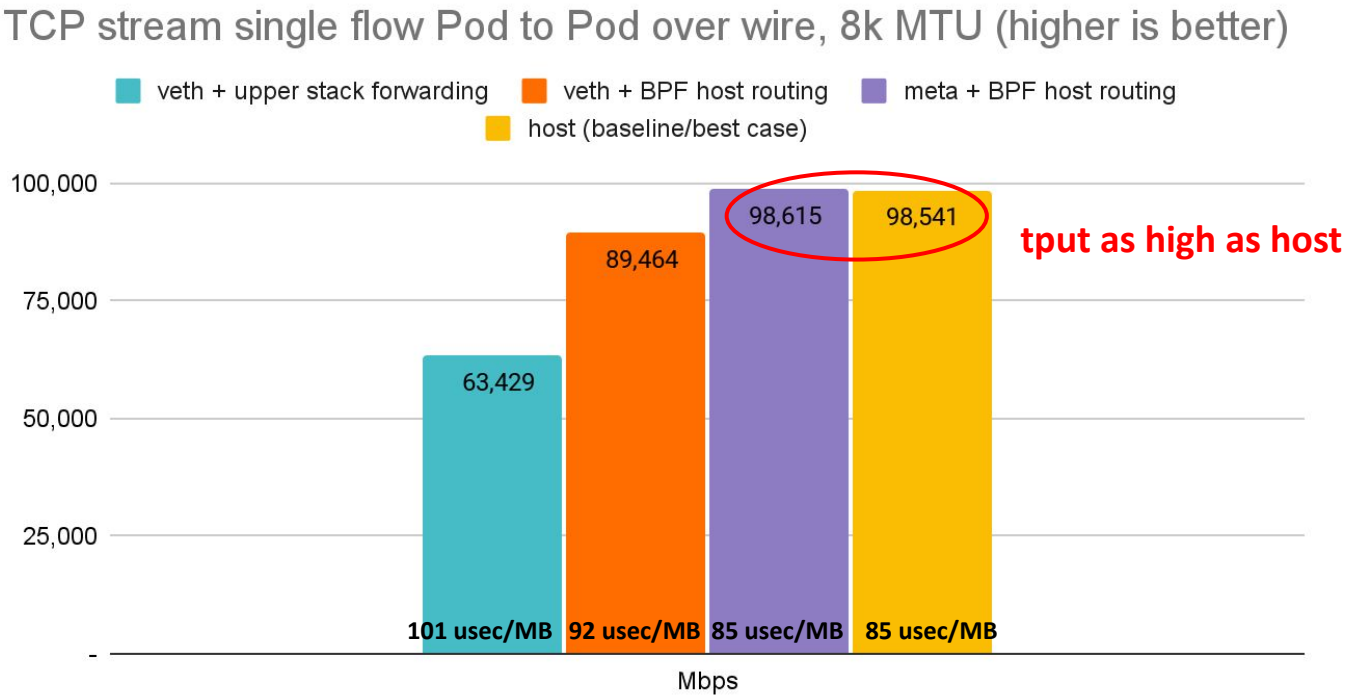
veth vs meta: backlog queue





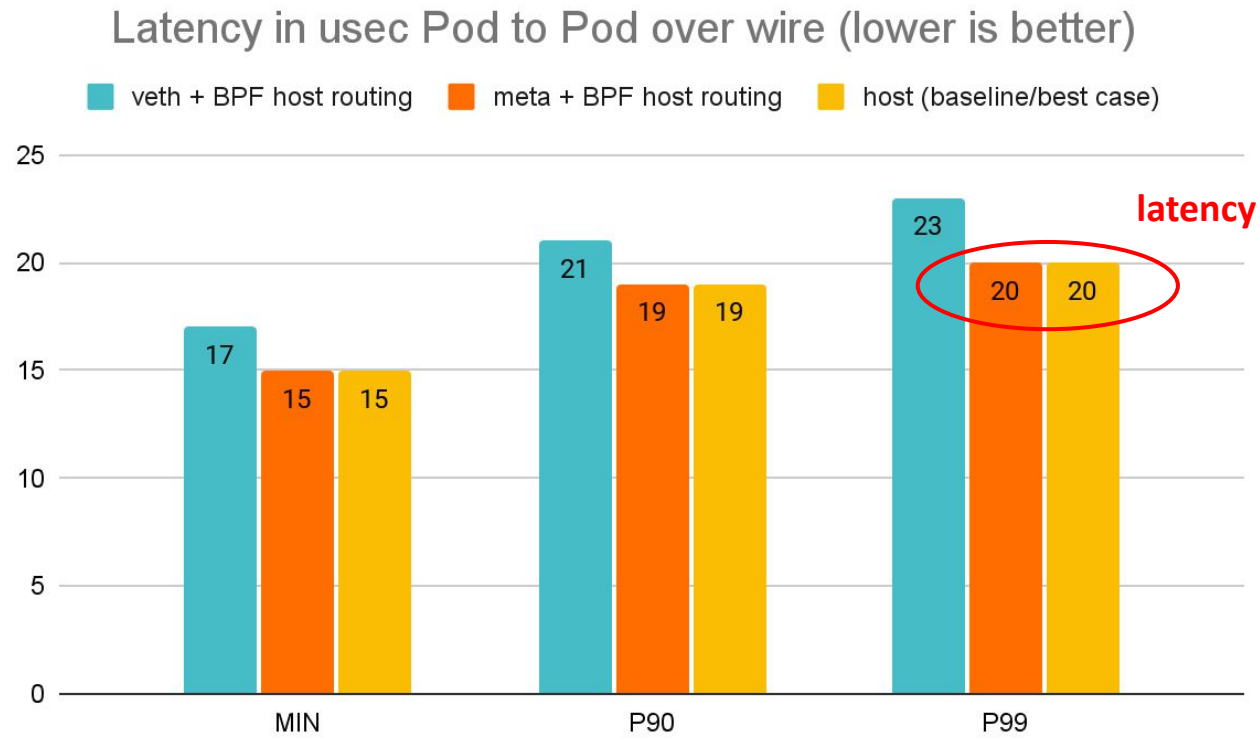
Remains in process context all the way

meta + BPF host routing case, results:



Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

meta + BPF host routing case, results:



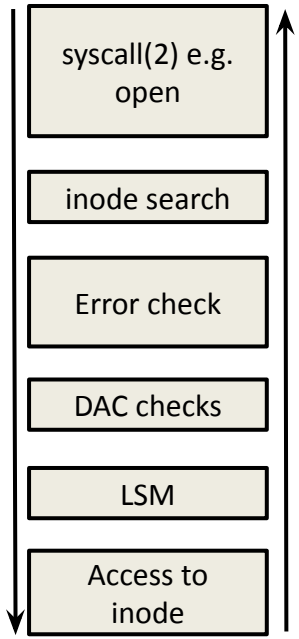
Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off
netperf -t TCP_RR -H <remote pod> -- -O MIN_LATENCY,P90_LATENCY,P99_LATENCY,THROUGHPUT

Security & eBPF

Security & eBPF

Linux security stack in a nutshell

Next: Overview (extract) of different hooks & open problems



Security & eBPF

seccomp-BPF

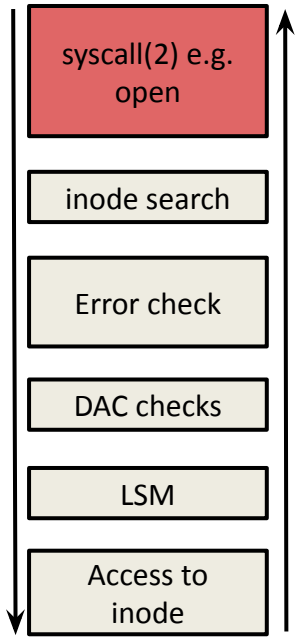
```
struct seccomp_data {
        int nr;
        __u32 arch;
        __u64 instruction_pointer;
        __u64 args[6];
};
```

General information:

- Minimised instruction set based on ”classic” BPF which gets translated to eBPF under the hood
- Runs on every syscall
- Several attempts to extend seccomp to native eBPF (e.g. Daniel Gruss et al.) but received pushback from upstream: no compelling story for unprivileged eBPF at this point

Open Problems:

- Only ”classic” BPF supported today, no BPF maps or other helpers
- Limited to only 4k instructions, often containing search-tree with allowed syscall numbers
- No deep argument parsing possible due to TOCTOU race



Security & eBPF

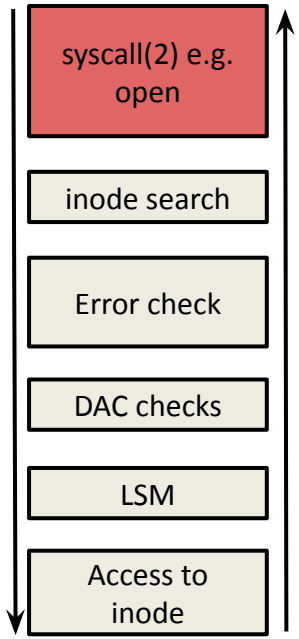
BPF tracing with bpf_override_return and bpf_send_signal

General information:

- BPF tracing can attach to kernel functions with ALLOW_ERROR_INJECTION() and trigger errors via fault injection capabilities through bpf_override_return() helper
- SIGKILL can be triggered to the ‘current’ task via bpf_send_signal() helper
- Can be attached to run on every syscall
- Solves seccomp-BPF limitations except the TOCTOU race when accessing syscall argument structures
- Used by various security projects (e.g. Tetragon) for enforcement

Open Problems:

- Needs CONFIG_FUNCTION_ERROR_INJECTION enabled in distribution kernels
- Typically not application but rather orchestrator sets up security policies given extended capabilities are needed (CAP_BPF, CAP_PERFMON)
- No deep argument parsing possible due to TOCTOU race



Security & eBPF

BPF LSMs

General information:

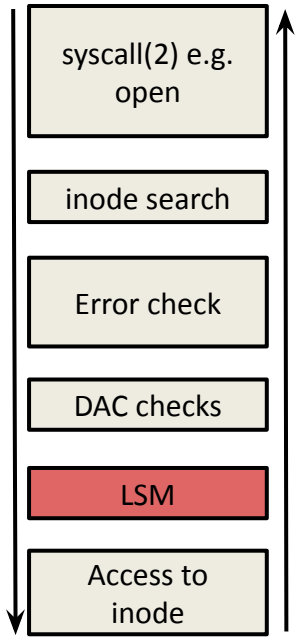
- Linux LSM infrastructure has a rich set of security hooks throughout the whole kernel where security policy can be enforced, used by SELinux, AppArmor
- BPF can be used to implement a LSM since kernel v5.7
- Compared to seccomp and syscall tracing, this solves the TOCTOU race since buffers reside inside the kernel

Ongoing Improvements:

- Small extensions to have per-cgroup LSM functionality, suitable for container workloads
- Initial proposal for extended attribute (xattr) support for BPF LSM programs

Open Problems:

- LSM hooks are invoked as indirect function calls, registered to a linked list at boot time
  - Due to retpoline mitigation they have a high overhead
  - Proposal to reduce overhead under discussion but slow moving, workload gains of up to 10% have been observed. Blocker for moving into production.
- Still early in development from infrastructure side, but has the most potential



Security & eBPF

New directions: Confidential Computing?

Protection of data in use by performing computation in a hardware-based, attested Trusted Execution Environment (TEE)

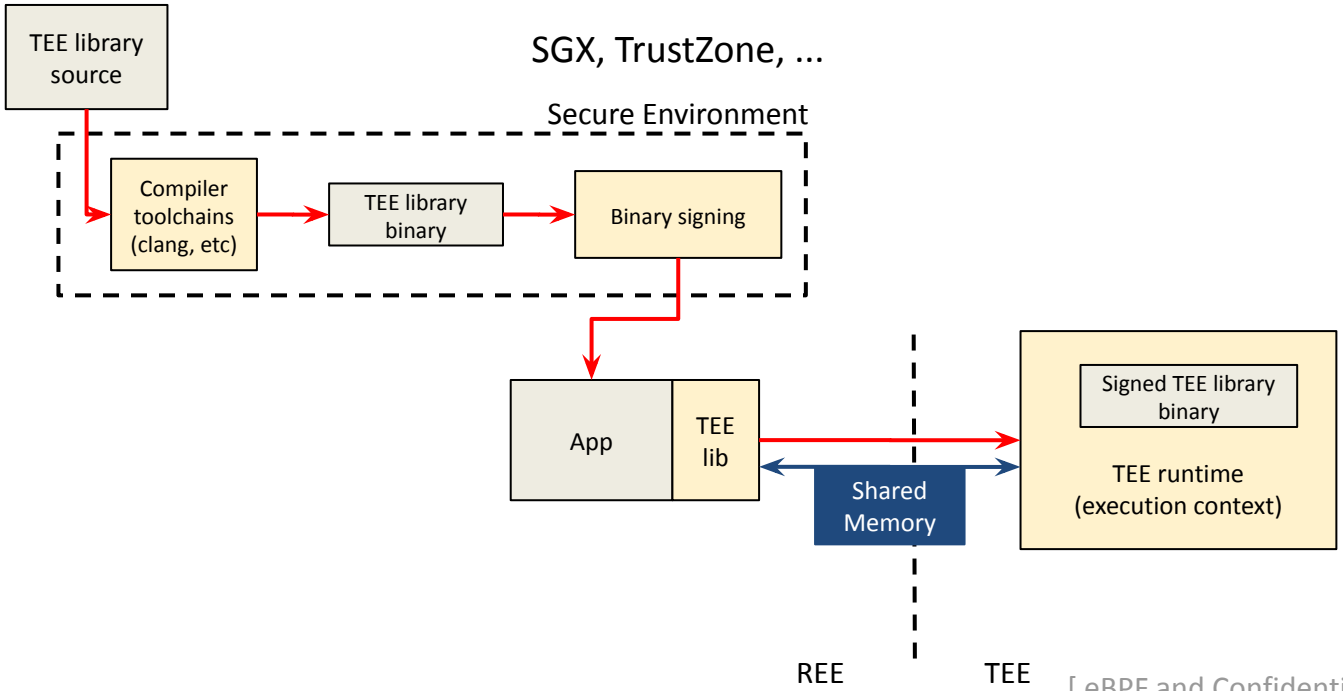
TEE is an environment that provides a level of assurance of data integrity, data confidentiality, and code integrity

- TEE is a privileged system component (compare: user vs kernel space)
- TEE is not part of the normal CPU REE (compare: offload to SmartNICs)

[eBPF and Confidential Computing at bpfconf'23]

Security & eBPF

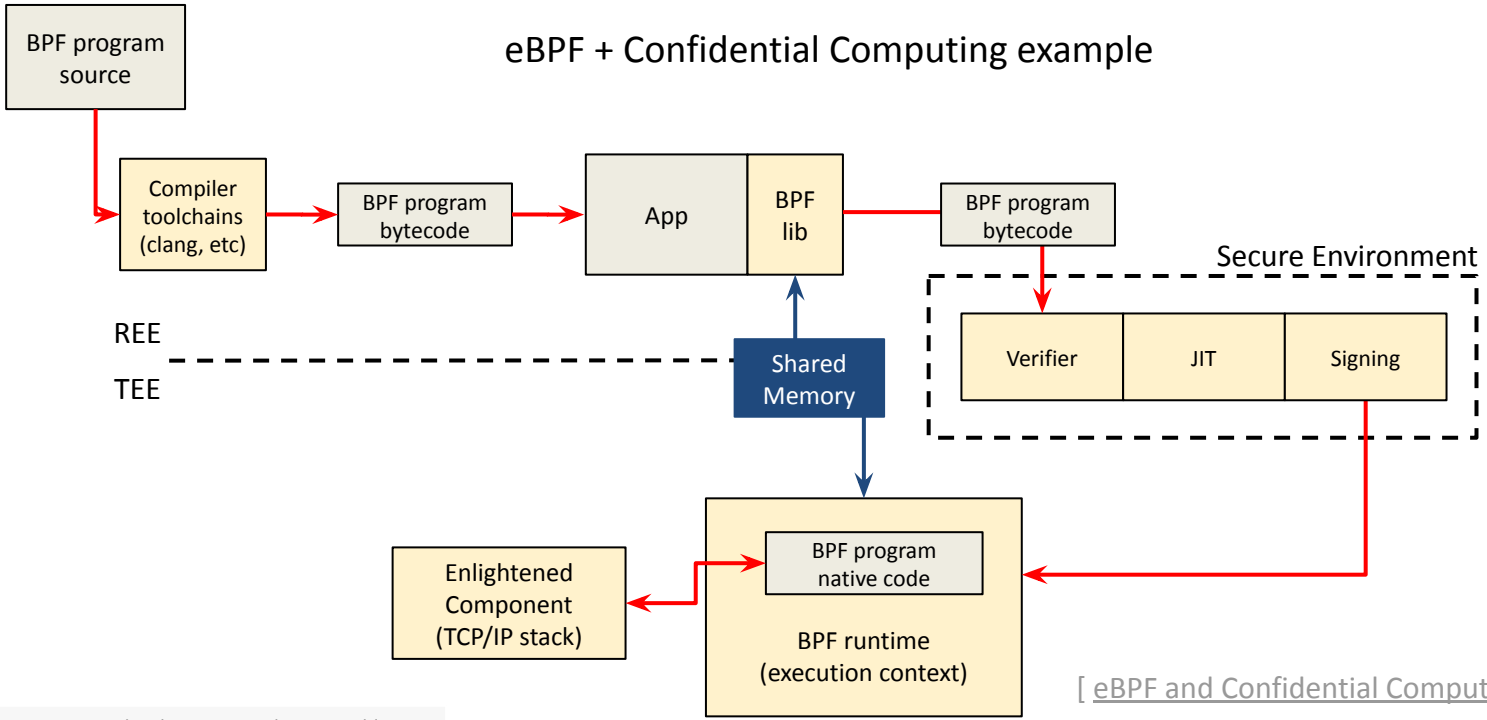
New directions: Confidential Computing?



[eBPF and Confidential Computing at bpfconf'23]

Security & eBPF

New directions: Confidential Computing?

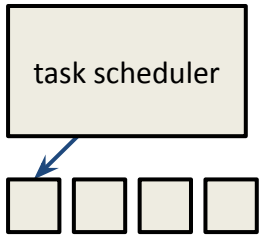


[eBPF and Confidential Computing at bpfconf'23]

Scheduling & eBPF

Scheduling & eBPF

sched_ext



General information:

- New process scheduler class, operating at a lower priority than CFS
- Enables scheduling policies to be written in BPF programs
- Use cases include rapid experimentation, bespoke schedulers that are optimized for specific services, quick rollout of new policies e.g. to work-around CPU bugs until mitigations are applied
- From Facebook production experiments main web service optimized by 1.25-3% RPS, 3-6% P99 latency compared to fine-tuned vanilla CFS

Ongoing Improvements:

- Not upstream yet, under ongoing discussion with maintainers who are mainly concerned about improvements not flowing back into CFS
- Usability improvements to BPF verifier merged

Misc:

- Consolidated effort from Facebook & Google
- Also interest from mobile phone vendors for building energy efficient schedulers

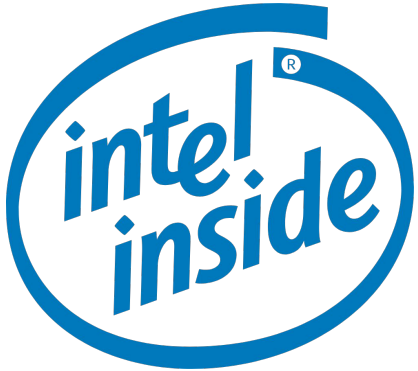
[sched_ext at bpfconf'23]

5) Outlook & How can the academic community contribute?

Outlook: eBPF inside

Remember “Intel inside” back in the days?

- Everybody wanted a laptop or server from Intel “because it was Intel inside”
- Not because they knew exactly what it was, but because they thought what was inside would provide them a better computing experience ...



Outlook: eBPF inside

... for eBPF this is exactly the same case:

- Not everybody knows what eBPF is, but users want products with eBPF inside because of what it enables them to do

As a community we need to ensure to stay on path for eBPF's mission:

- Accelerate the speed of innovation, experimentation and research
- Provide the technology to enable developers to deploy with unprecedented speed and scale

eBPF: A silent (r)evolution from cloud native

- Silent given eBPF is already used under the hood in many projects and products in cloud native enabling enrichment with cloud native context
- eBPF has been in production and is production-proven for more than half a decade
  - Started out in cloud native and now expanding into other industry sectors
- Users do not directly interface with eBPF - might not even know about it - but benefit from it as an underlying tool (“invisible technology, visible benefits”)

[eBPF for all, eBPF - the silent platform revolution]

eBPF kernel community statistics

- 4 core maintainers (Meta, Isovalent)
- 11 core reviewers who are in review rotation (Meta, Google, Isovalent)
- Approx. 50 reviewers per development cycle
- Approx. 280 active developers
- Currently 1 active developer from academia

Linux kernel v6.5 & v6.6 statistics for the eBPF subsystem:

```
Previous cycle:
27 Apr to 28 Jun: 4234 mailing list messages, 62 days, 68 messages per day
664 repo commits (11 commits/day)

Current cycle:
28 Jun to 29 Aug: 5240 mailing list messages, 62 days, 85 messages per day
384 repo commits (6 commits/day)
```

[bpf development stats for 6.6]

How can academic community contribute?

Help building a strong, long-term foundation of the eBPF core parts

- Further formal verification of BPF (& BTF) verifier (latest work covers range analysis)
- In-depth security research / analysis of BPF (& BTF) verifier
- Reduction of verifier complexity
- Usability improvements and language extensions
- Spectre & BPF: How can we better overcome transient execution attacks
- Documenting / defining a BPF memory model & KCSAN integration

- Finding a viable, safe solution for unprivileged BPF
- Supply chain security: Signing in context of dynamic BPF program generation

How can academic community contribute?

Research into new areas for eBPF and eBPF as a platform to innovate

- Confidential computing: Can BPF play an important role here?
- New applications in critical system components (IoT, 5G, embedded/industrial, etc)
- New applications in sustainable computing (e.g. Kepler)
- Better policy decisions due to the distributed nature of BPF
  - Domain-specific process schedulers
  - TCP congestion control for data center networks
  - Kernel security, building novel sandboxes & LSMs
- Profiling BPF with BPF, uprobe/USDT probe speedup (BSC roadmap)

How can academic community contribute?

Research into new areas for eBPF and eBPF as a platform to innovate

- Confidential computing: Can BPF play an important role here?
- New applications in critical system components (IoT, 5G, embedded/industrial, etc)
- New applications in sustainable computing (e.g. Kepler)
- Better policy decisions due to the distributed nature of BPF
  - Domain-specific process schedulers
  - TCP congestion control for data center networks
  - Kernel security, building novel sandboxes & LSMs
- Profiling BPF with BPF, uprobe/USDT probe speedup (BSC roadmap)

Most importantly:

Contribute patches to the upstream eBPF community as part of your research outcome
Exchange ideas, being involved in discussions and participate in both communities

Also: How can the industry facilitate academic research on eBPF?