An eBPF introduction

The in-kernel virtual machine

Intro

- Lots of slides -> I’ll have to hurry a bit #speedrun
- ~~Feel free to ask questions at any time! ;)~~
    - But longer questions at the end / after the talk please
- Background: My master’s thesis (~ 2 years ago)
- The images are not from me (s. hyperlinks for the sources)
- I’ll make the slides available

What is eBPF?

- Origin: eBPF = extended Berkeley Packet Filter (s. next slide)
- Nowadays: a technology / new type of software
- An in-kernel virtual machine (VM)
  - A bit like running Java bytecode in the Java Virtual Machine (JVM)
  - Analogy: Similar to the JavaScript support of web browsers (-> programmability)
- “eBPF is a revolutionary technology with origins in the Linux kernel that can run **sandboxed programs** in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules.”

The origin of eBPF

tcpdump -i lo host 127.0.0.1 and port 80

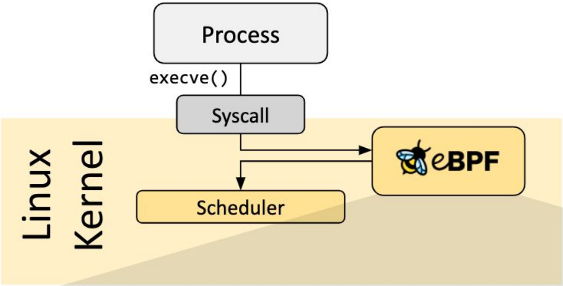
- BSD Packet Filter (BPF): 1992 - For network packet filters (monitoring)
- Originally: BPF, now called cBPF (classic Berkeley Packet Filter)
  - Main use: Filtering packets. Userspace program (tcpdump) can supply the filter
  - Implementation: A bytecode interpreter for an in-kernel VM
  - ISA: 32 bit (few fixed-length instr.), accumulator, index, and 16 “scratch memory store” regs.
- Available on most Unix-like systems, not just on Linux
  - E.g. {Free,Net}BSD (origin), DTrace on (Open)Solaris / illumos (Linux: bpftrace)
- eBPF is the successor of cBPF
  - extended Berkeley Packet Filter (since Linux 3.18)
  - General purpose RISC IS (designed for writing programs in a subset of C; helper functions)
  - 11 64-bit registers (32 bit subregisters, r10: ro frame pointer), PC, and 512 byte stack
  - Nowadays: Only eBPF (cBPF transparently translated to eBPF)
- BPF is now a technology / new type of software

History (eBPF constantly grows)

- 1992: BSD Packet Filter (BPF paper; ISA + register-based pseudo-machine)
- 2011: 3.0: cBPF JIT
- 2014: 3.15/3.18: eBPF
- 2014: LLVM
- 2015: 3.19: Socket tracepoint
- 2015: 4.1: Traffic control (TC) classifier tracepoint
- ~2015: BCC: BPF Compiler Collection
- 2019: 5.3: Bounded loops
- 2019: GCC
- 2021: eBPF foundation (Linux Foundation announcement)

eBPF programs are event-driven (attached to a code path)

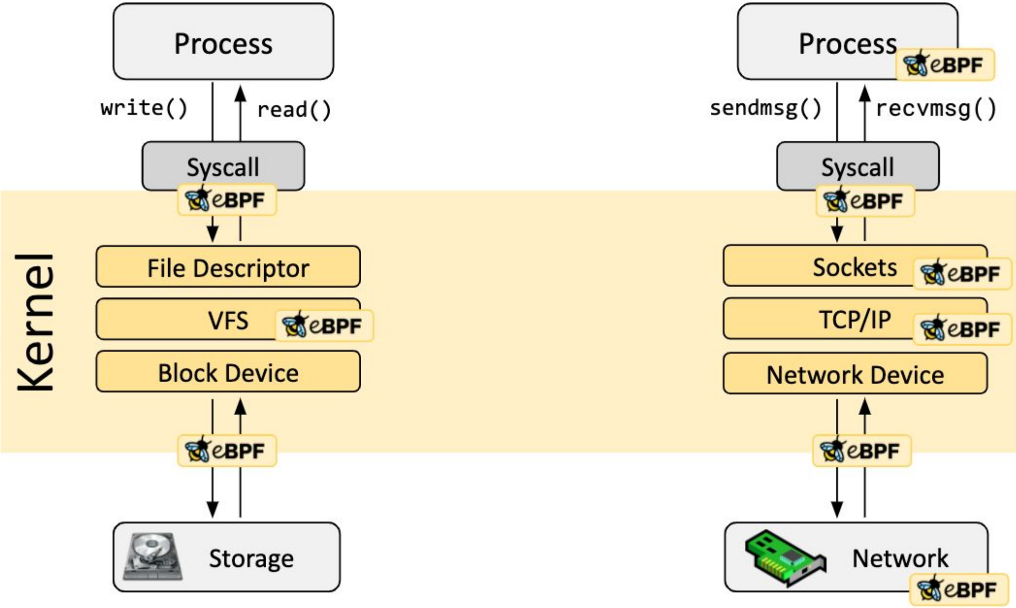
- Hooks/tracepoints: Network events (new packet), system calls (application), kernel tracepoints, etc. or even custom kernel/user probes ({k,u}probe)




```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

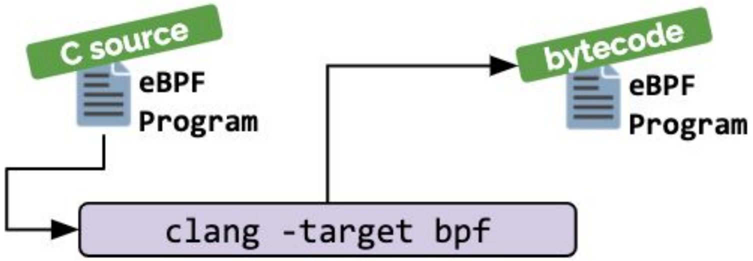


Why eBPF?

- **Safe**/secure/sandboxed: Bytecode verified before running it (no DOS, accidental crashes, arbitrary memory access, etc.)
- **Fast**/performant (close to natively compiled in-kernel code): Just-in-time (JIT) compiler converts BPF instructions into native code that runs in kernel-space
  - Programs can (limitations!) even be offloaded to HW
- Flexible: General purpose enough for many use-cases
- Stable API and ABI (unlike kernel modules)
- Portable: Even user-mode interpreters (via pcap API, implemented by libpcap on Linux; or via uBPF) that support Linux and non-Linux systems; and Windows support
  - CO-RE (Compile Once Run Everywhere): Support multiple kernel versions without recompiling
- A lot of existing tools and well supported (libraries, applications, languages, etc.)
- Compilers for higher-level languages: C, a subset of P4, Rust (aya), even Python, etc.

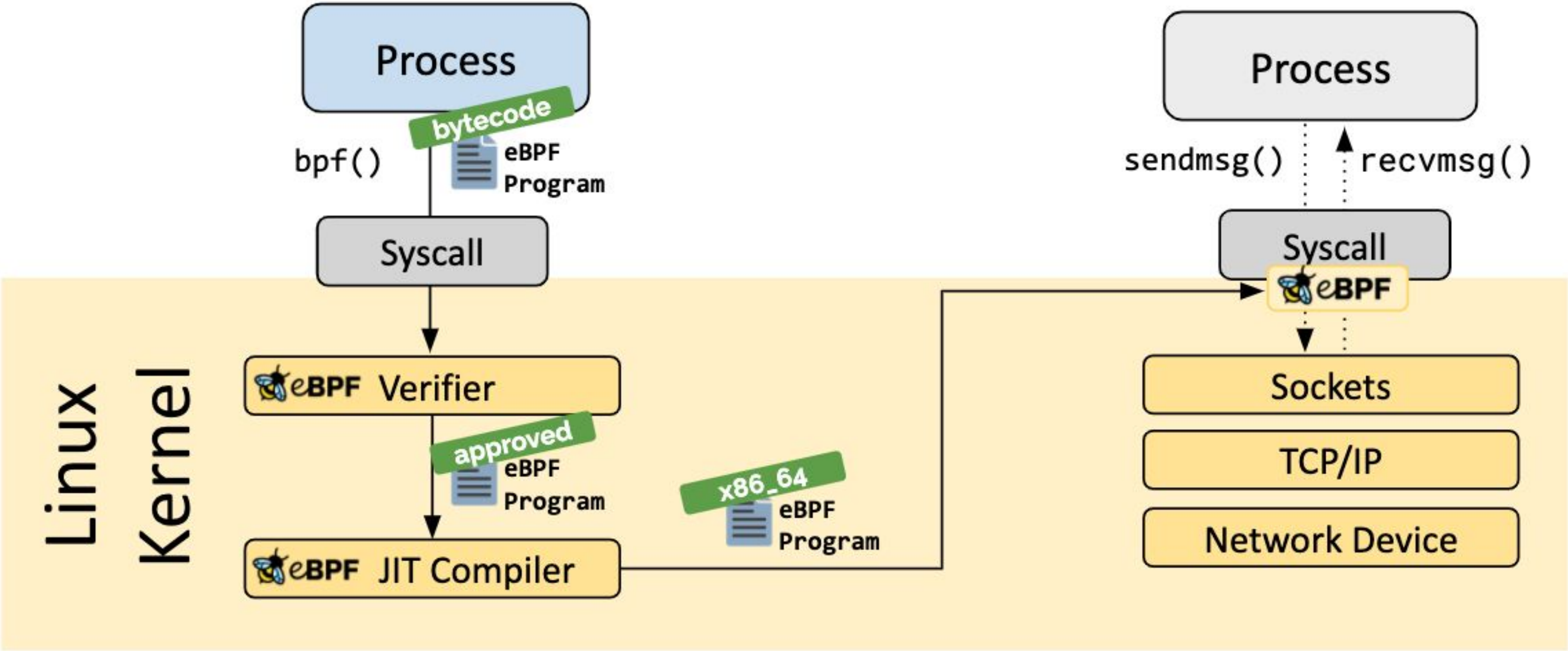
Using higher-level languages

- Lots of different languages supported (C, Go, Rust, Python, Lua, etc.)
  - Even special languages (DSLs), e.g. P4 (p4c-ebpf (TC), p4c-xdp, p4c-ubpf), BCC (toolkit and library), and bpftrace (high-level tracing language)
- Compilers with BPF target: LLVM (2014; also BCC) and GCC (2019)
- Before that: eBPF assembly -> bpf_asm/bpfc/ubpf -> eBPF bytecode
- LLVM example:



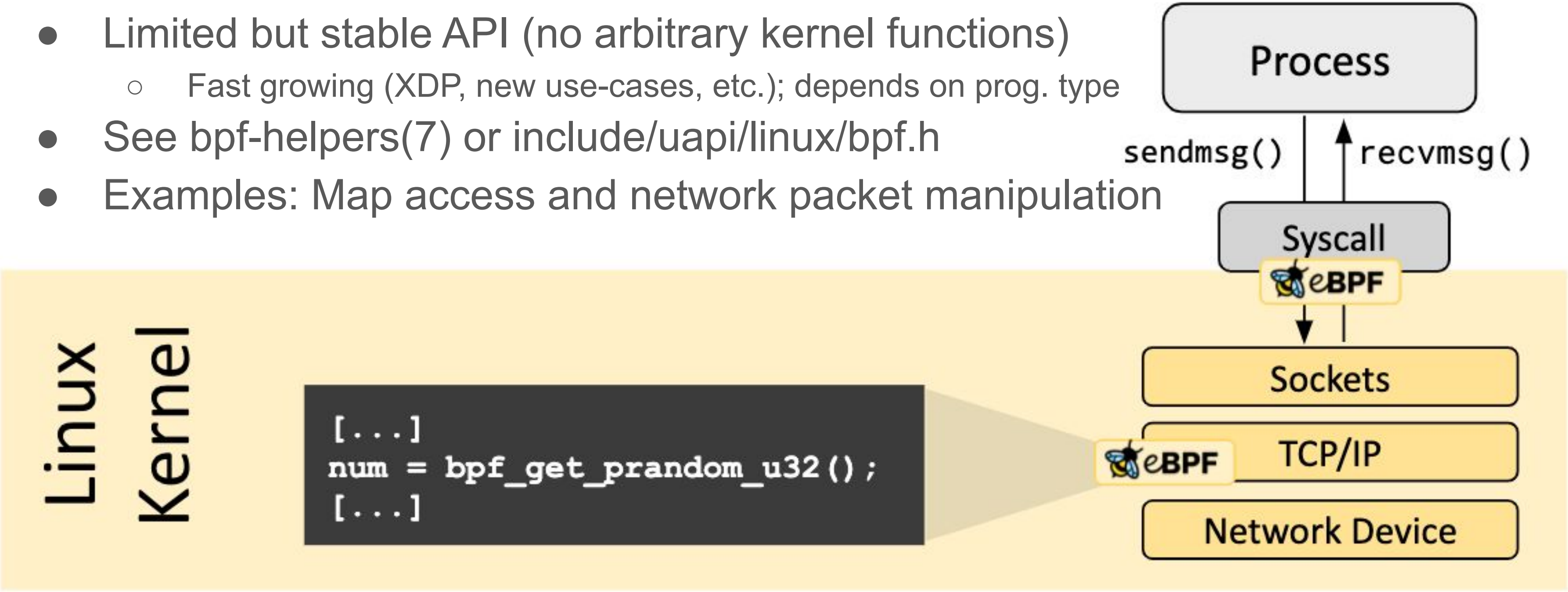
Loading and verification

- Loading: Manually (bpf() syscall / library) or via iproute2 (ip/tc), etc.



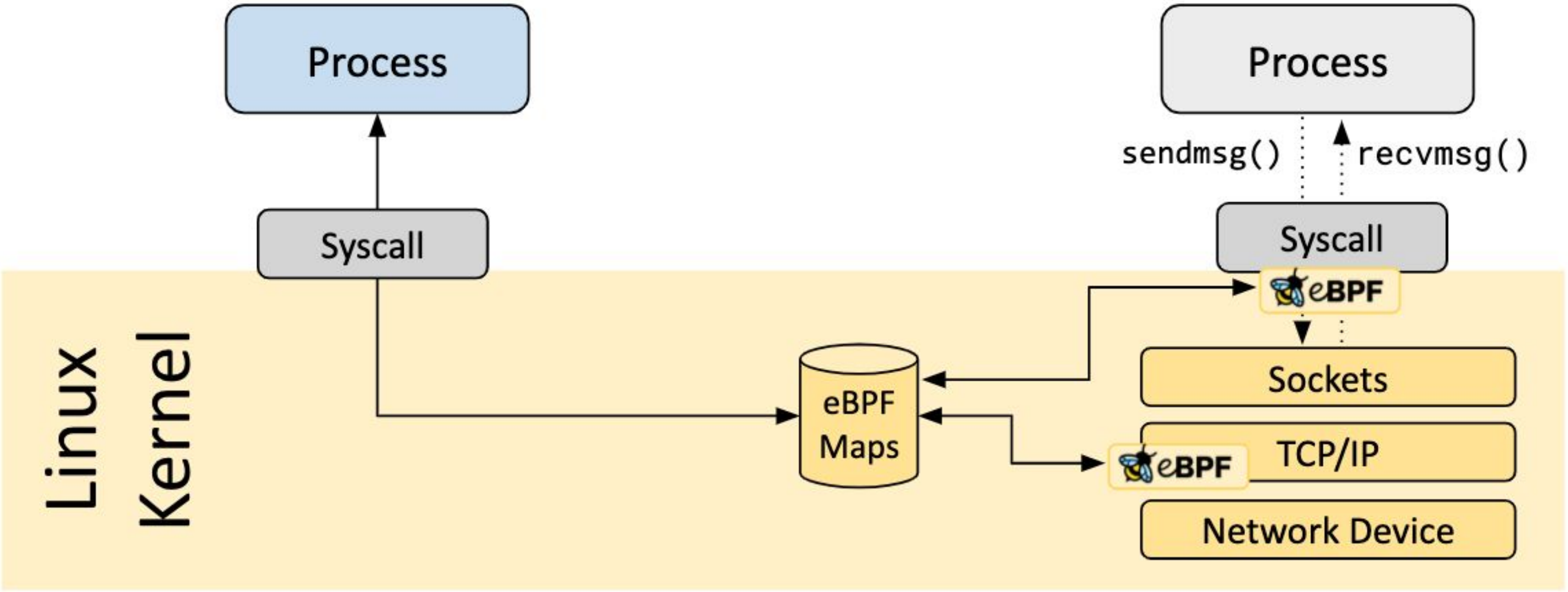
eBPF helper functions

- Limited but stable API (no arbitrary kernel functions)
  - Fast growing (XDP, new use-cases, etc.); depends on prog. type
- See bpf-helpers(7) or include/uapi/linux/bpf.h
- Examples: Map access and network packet manipulation

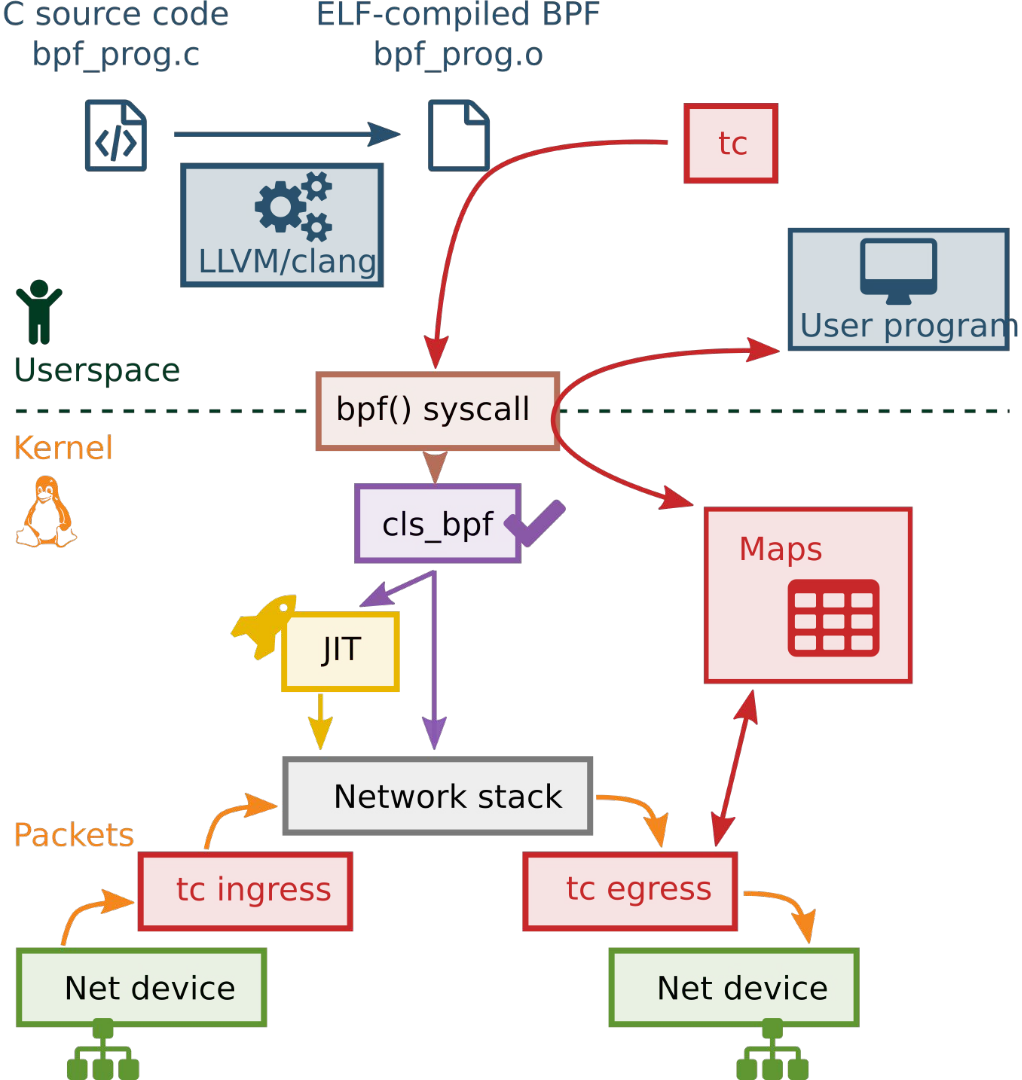


State: eBPF maps (key/value store)

- Can be accessed from eBPF program(s) and user space
- Many (10+) different types exist (e.g. arrays and hash tables (optional: LRU))

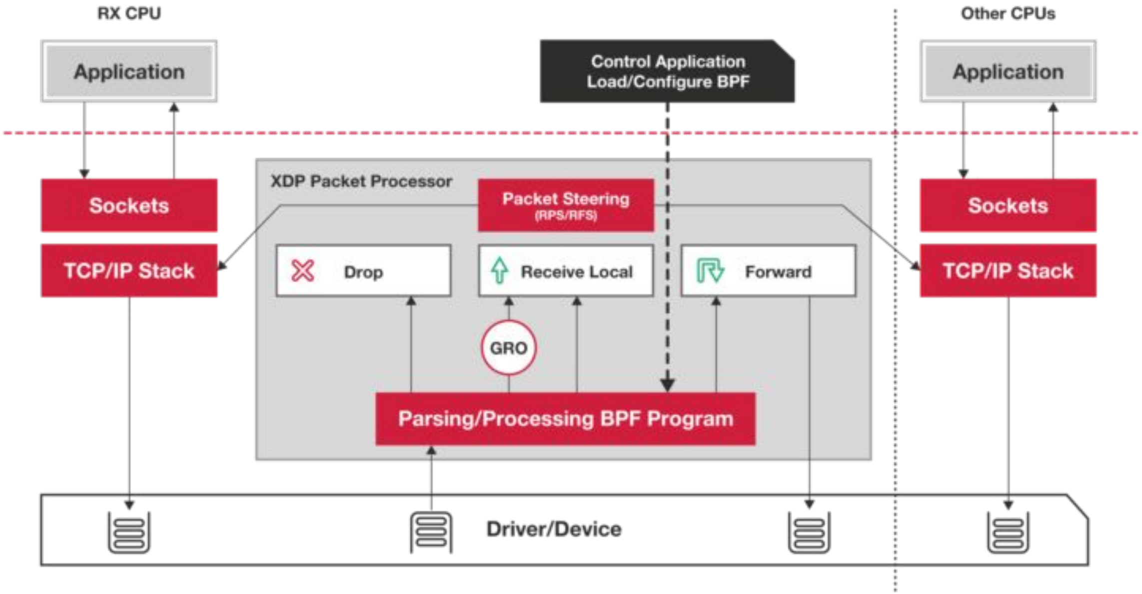


Example


C source code
bpf_prog.c
ELF-compiled BPF
bpf_prog.o
LLVM/clang
tc
User program
Userspace
bpf() syscall
Kernel
cls_bpf
Maps
JIT
Network stack
Packets
tc ingress
tc egress
Net device
Net device

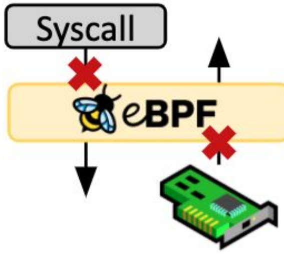
XDP: eXpress Data Path

- High-performance packet processing
- eBPF program runs at the lowest level of the (RX) network stack
    - Immediately after packet is received (i.e. before any parsing/processing)

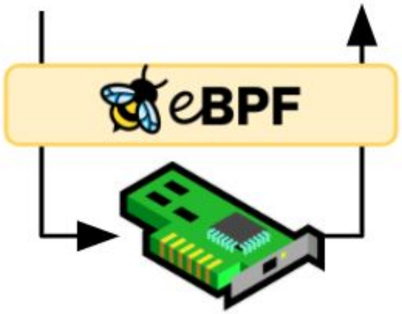


Main use cases (currently)

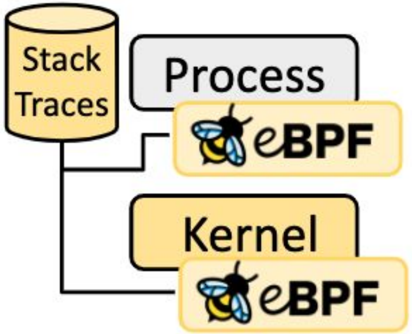
- Security



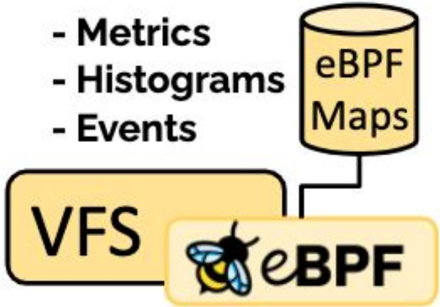
- Networking



- Tracing and profiling



- Overvability and monitoring



Problems

- When using C: High chance for compiling an invalid eBPF program
  - User will only know when loading/running it
  - BCC: Aims to provide a BPF-specific frontend -> feedback from the compiler
- Stable ABI can break when using kernel internal data structures (requires compiling with kernel internal headers) for tracing programs and tracepoints can change -> but nowadays: CO-RE
  - I.e. (some) eBPF programs can still loosely depend on the kernel version
- Restrictions/limitations: The verifier imposes a lot of restrictions (length/size, stack size, termination / finite loops, memory access, no uninitialized variables, finite and limited complexity, etc.) and the API is limited to a small set of helper functions (and cannot be extended via kernel modules)

Problems II

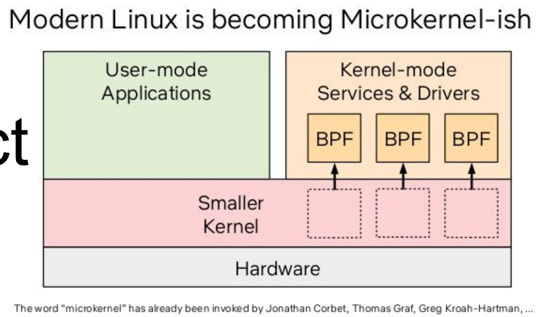
- Limited API (many helper functions but might still not be enough)
  - Cannot be extended via kernel modules and no arbitrary kernel functions
- Lack of documentation
  - Especially official documentation! (Exception: eBPF helper functions)

Example eBPF users, use-cases, and applications

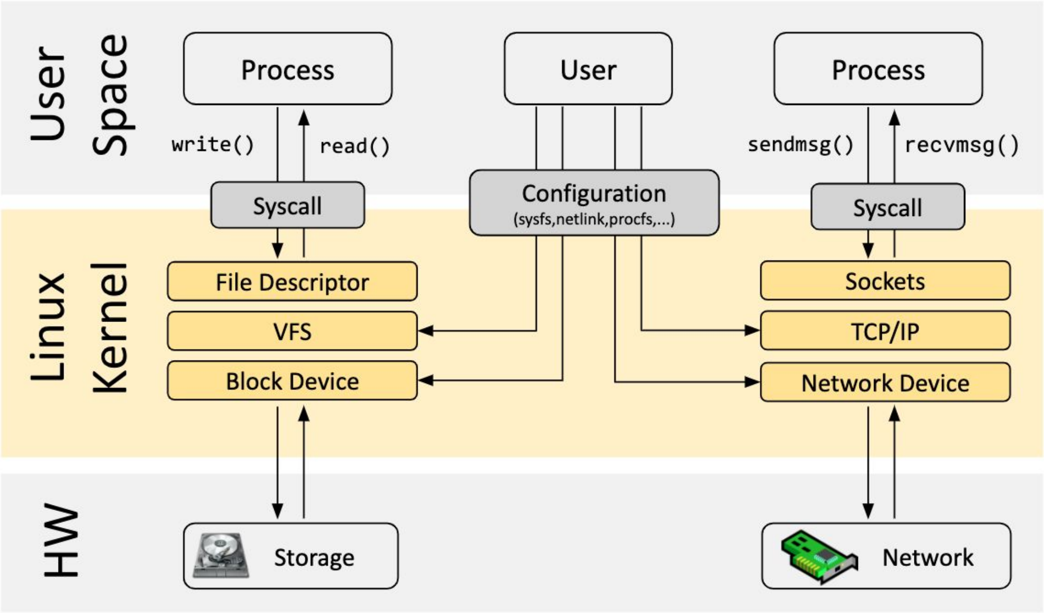
- https://ebpf.io/projects/ (lots of open-source applications/projects!)
- Industry users (https://ebpf.io/case-studies/): Google, Cloudflare, Android, Meta (~40/server), Netflix (~14/server), Red Hat, etc. (also via systemd)
- Use-cases: Networking (SDN, monitoring, firewalls, …), security (seccomp, IDS, containers, observability), kernel debugging, perf analysis, etc.
- A new type of software:

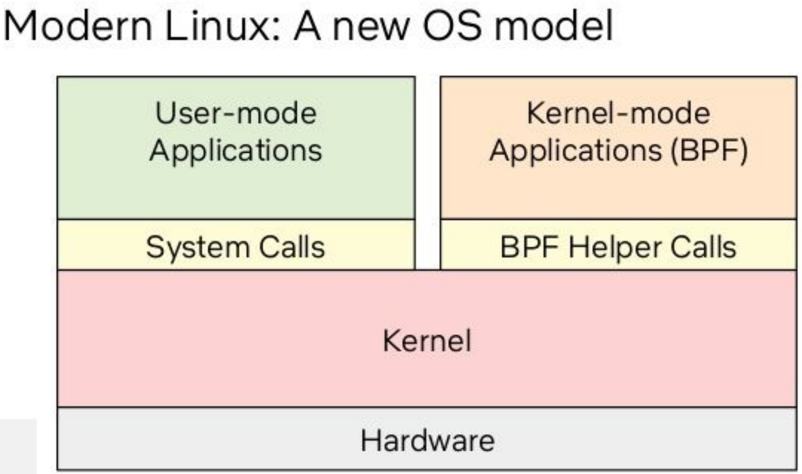
	Execution model	User defined	Compil-ation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers

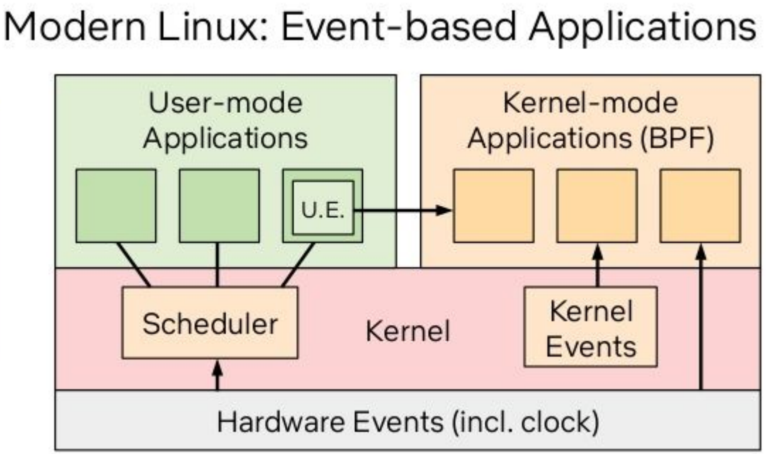
eBPF impact



- Making the Linux kernel reprogrammable
  - Vs. source-code changes, kernel module, etc.







Future

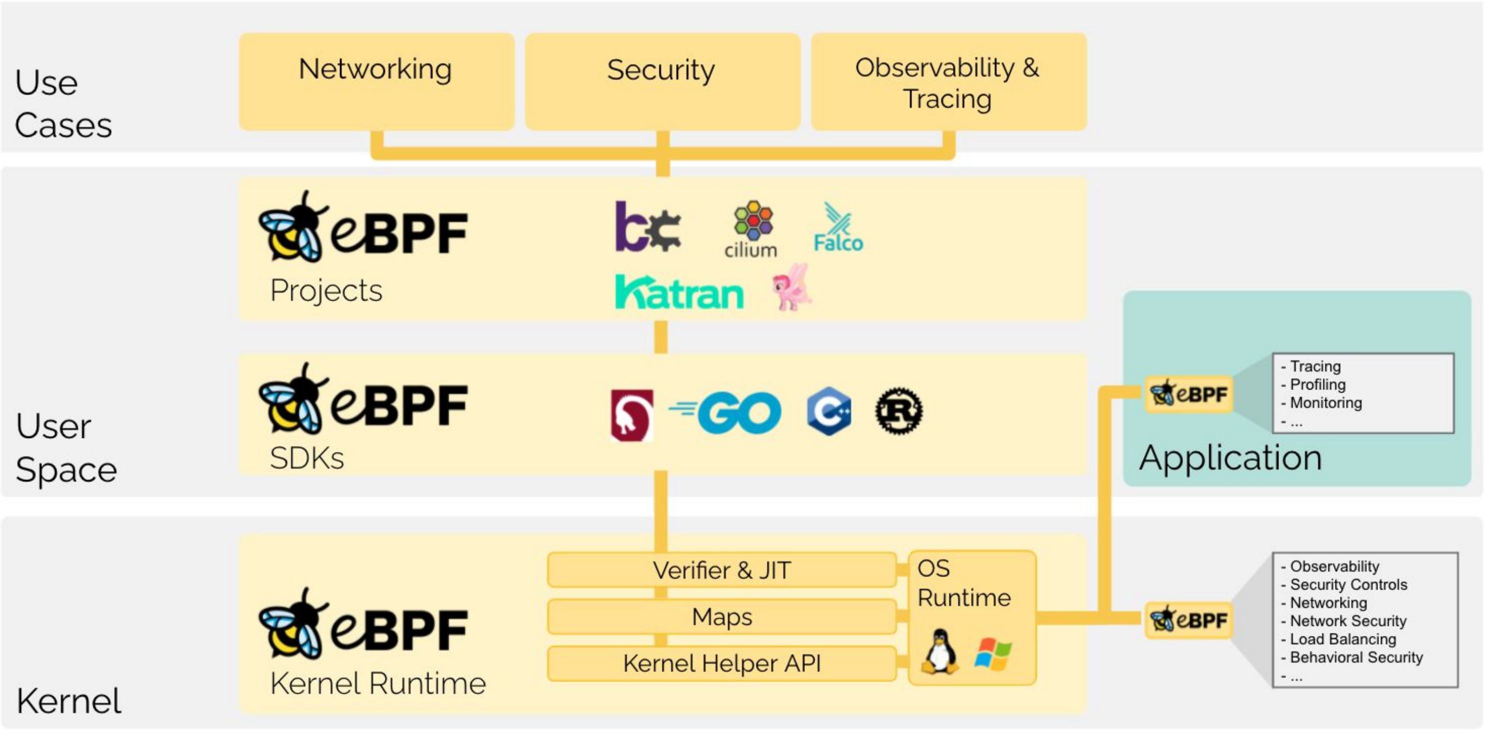
- New features (helper functions, hooks, less verifier restrictions, etc.)
- Many new use-cases and users (e.g. containers, networking, sandboxing, tracing, and even device drivers)
- Unprivileged eBPF?
- Fully reprogrammable Linux kernel?
  - But IMO not a path towards a microkernel (too much complexity and would require replacing kernel subsystems with eBPF programs, a strict privilege level separation, and a clean API)
- Alternative to livepatching?
  - To hotpatch kernel vulnerabilities or bugs (by changing control flow, input sanitization, etc.)
- Potential problems: Could result in less upstreamed fixes, features, etc.
- Potential advantages: Could help to avoid upstreaming special purpose code

End of presentation

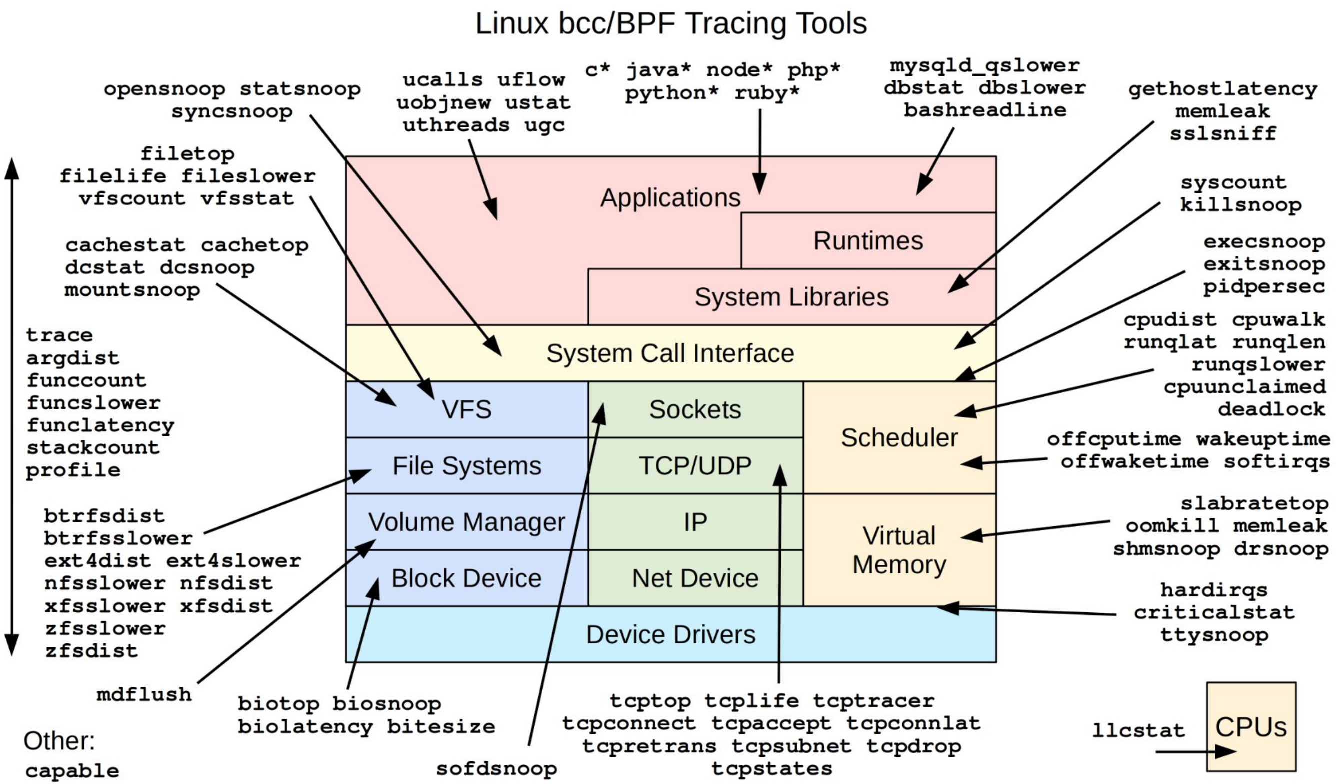
- Thank you!
- Any questions?

Ecosystem overview

- Applications: https://ebpf.io/applications/



BCC tracing tools



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

Unprivileged eBPF

- Allows unprivileged users to load certain types of eBPF programs
  - E.g. socket filters
- Should be safe but in reality still many issues (-> currently many restrictions)
  - Enabled by default but should be disabled (kernel.unprivileged_bpf_disabled)
  - Many CVEs (privilege escalation, kernel crashes / DoS, etc.)
- Requires additional verifier checks / hardening (secure mode)
  - Prevent leaking of kernel pointer values (+ no pointer arithmetic allowed?)
  - Prevent speculative execution attacks
  - Mark memory for eBPF program as read-only + constant blinding (prevents injecting code)
- Has been abandoned as unachievable
  - But still interest from some and attempts to make it work
  - Use-cases: Containers, seccomp, socket filters, etc.
- Future unclear (heated discussions)

Resources

- https://ebpf.io/
- Linux kernel BPF documentation (WIP)
  - Linux Socket Filtering aka Berkeley Packet Filter (BPF)
  - Various man-pages (bpf, bpf-helpers, etc.)
- Cilium project
  - Introduction
  - Documentation/Overview
  - BPF and XDP Reference Guide
- Awesome eBPF

Backup/WIP slides

eBPF verifier checks (WIP - changes too frequently)

- Program terminates (i.e. all loops are bounded)