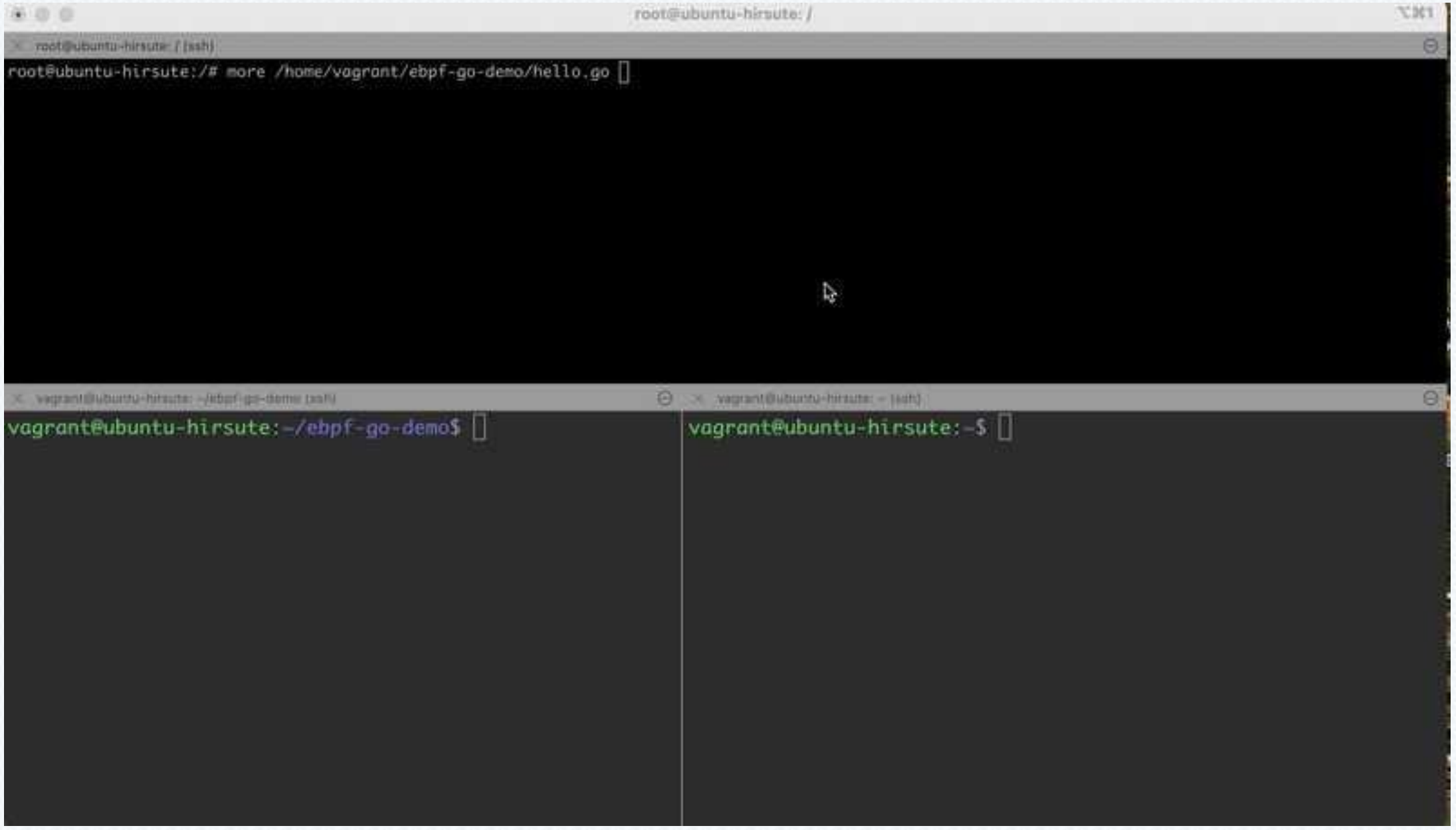
深入浅出 BPF 技术

-- 狄卫华
Revised by 肖卿俊

目 录

热身阶段 - 小的 demo


root@ubuntu-hirsute: /
root@ubuntu-hirsute:/# more /home/vagrant/ebpf-go-demo/hello.go
vagrant@ubuntu-hirsute:~/ebpf-go-demo$
vagrant@ubuntu-hirsute:~$

eBPF：过去50年操作系统最大的变革！！！

1. eBPF 是什么？

1. BPF 的由来

BPF = Berkeley Packet Filter

来自于 Steven McCanne && Van Jacobson 1993 年公开发表的论文：

《The BSD packet filter: a new architecture for user-level packet capture》。

通过引入的 BPF 技术，将数据包过滤技术性能提升 20 倍以上。主要是通过：

- 新的虚拟机（VM）设计，可以有效地工作在基于寄存器结构的 CPU 之上；
- BPF 程序在内核层进行高效过滤，避免了将数据复制到用户空间进行处理；

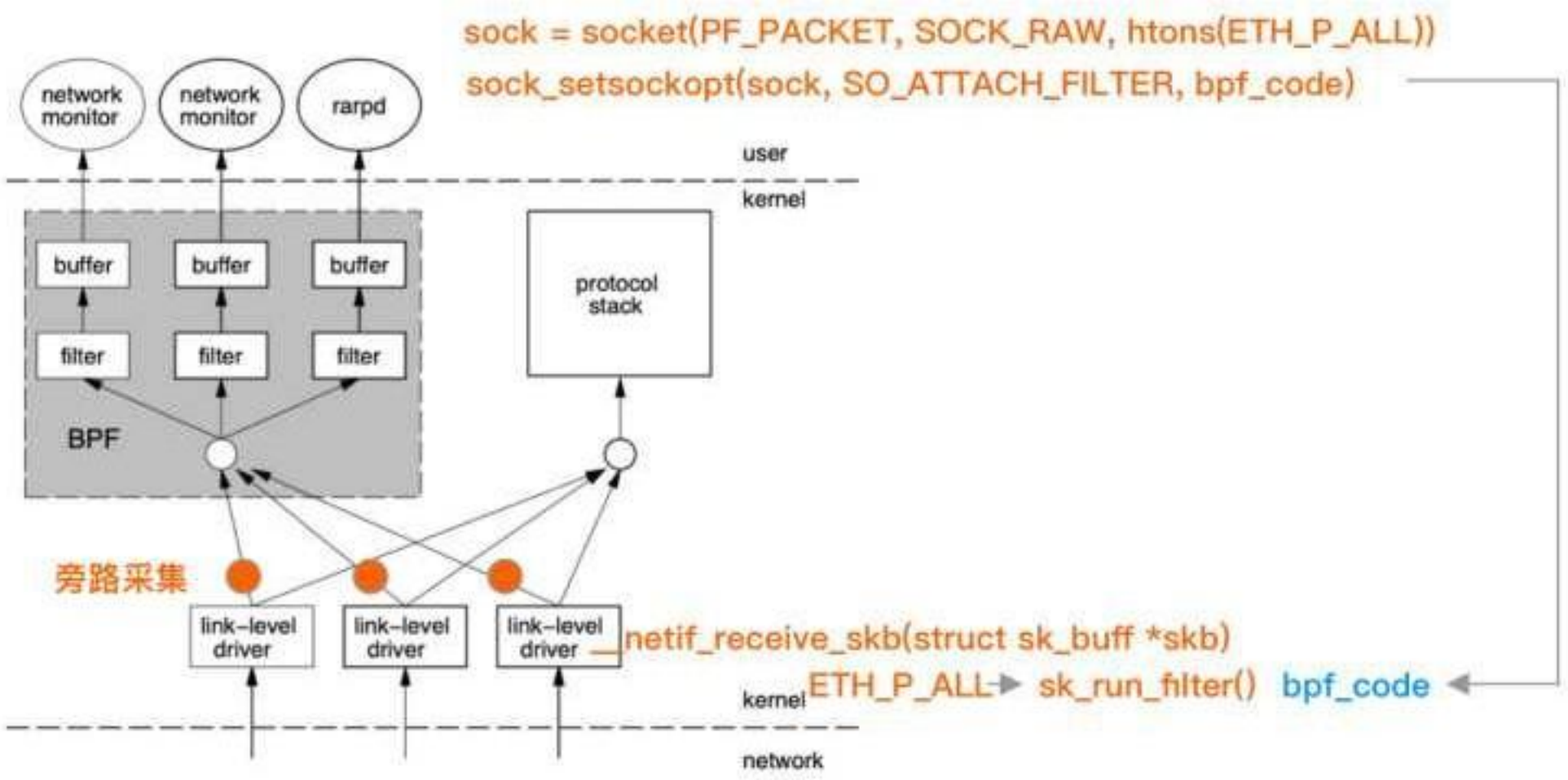
VM 的设计非常简洁， 2 个 32 位的寄存器 A/X，16 个内存位：M[0－15]，33 个指令集。

1. eBPF 是什么？

1.1 BPF 的由来

1. eBPF 是什么？

1.1 BPF 的由来



1. eBPF 是什么？

1.1 BPF 的由来

```
$ tcpdump -d 'ip and tcp port 8080'
(1)     ldh     [12]
(2)     jeq     #0x800          jt 2 jf 12
(3)     ldb     [23]
(4)     jeq     #0x6            jt 4    jf 12
(004) ldh     [20]
(005) jset    #0x1fff         jt 12 jf 6
...
```

1. eBPF 是什么？

1.1 BPF 的由来

1997 年合并入 Linux 2.1.75 版本（BPF 开始位于 BSD 系统）

2011 年加入 BPF JIT 编译器

2012 年为 seccomp 添加 BPF过滤器

当前 BPF 的应用场景：

- **tcpdump** 格式的报文过滤
- Linux 网络流量控制 TC Qos **cls_bpf**
- **seccomp-bpf** 沙盒程序系统调用过滤
- Netfilter iptables **xt_bpf**

1. eBPF 是什么？

1.2 eBPF 后起之秀

1. eBPF 是什么？

1.2 eBPF 后起之秀

eBPF = extended Berkeley Packet Filter

2014 年初，Alexei Starovoitov 实现了 eBPF。eBPF 最早出现在 Linux 内核 3.18 内核中。

此后，BPF 就被称为经典 BPF，缩写 cBPF（classic BPF），cBPF 现在已经基本废弃。

现在，Linux 内核只运行 eBPF，内核会将加载的 cBPF 字节码透明地转换成 eBPF 再执行。

BPF => cBPF

BPF ⇔ eBPF

BPF ⇔ (eBPF, cBPF)

1. eBPF 是什么？

1.2 eBPF 后起之秀

2014 年 6 月，eBPF 扩展到用户空间，这也成为了 eBPF 技术的转折点。

Alexei：“这个补丁展示了 eBPF 的潜力”。

eBPF 演进为一个通用执行引擎，应用场景不在局限于网络数据包过滤，逐步扩展到内核各子模块，在观测、网络和安全等领域崭露头角，成为顶级的内核模块。

1. eBPF 是什么？

2. eBPF 后起之秀

eBPF 相对于 cBPF 的增强如下：

- 处理器原生指令集建模，因此**更接近底层处理器架构**， 性能相比 cBPF 提升 4 倍；
- 指令集从 33 个扩展到了 114 多个，依然保持了足够的简洁；
- 寄存器从 2 个 32 位寄存器扩展到了 11 个 64 位的寄存器 （其中 1 个只读的栈指针）；
- 引入 bpf_call 指令和寄存器传参约定，实现零（额外）开销内核函数调用；
- 虚拟机的最大栈空间是 512 字节（cBPF 为 16 个字节）；
- 引入了 map 结构，用于用户空间程序与内核中的 eBPF 程序数据交换；
- 最大指令数初期为 4096，现在已经将这个限制放大到了 100 万条；

1. eBPF 是什么？

3. eBPF 生产超能力

- **稳定：**有循环次数和代码路径触达限制，保证程序可以固定时间结束；
- **高效：**可通过 JIT 方式翻译成本地机器码，执行效率高效；
- **安全：**验证器会对 eBPF 程序的可访问函数集合和内存地址有严格限制，不会导致内核 Panic；
- **热加载/卸载（持续交付）：**可热加载/卸载 eBPF 程序，无需重启 Linux 系统；
- **内核内置：**eBPF 自身提供了稳定的 API；
- VM 通用引擎足够简洁通用 （87 个指令集，11 个寄存器， x86-64 2000个指令集）；

数据与功能分离

1. eBPF 是什么？

1.3 eBPF 生产超能力

eBPF 与内核模块对比：

维度	Linux 内核模块	eBPF
kprobes/tracepoints	支持	支持
安全性	可能引入安全漏洞或导致内核 Panic	通过验证器进行检查，可以保障内核安全
内核函数	可以调用内核函数	只能通过 BPF 辅助函数调用
编译性	需要编译内核	不需要编译内核，引入头文件即可
运行	基于相同内核运行	基于稳定 ABI 的 BPF 程序可以编译一次，各处运行
与应用程序交互	打印日志或文件	通过 perf_event 或 map 结构
数据结构丰富性	一般	丰富
入门门槛	高	低
升级	需要卸载和加载，可能导致处理流程中断	原子替换升级，不会造成处理流程中断
内核内置	视情况而定	内核内置支持

1. eBPF 是什么？

4. eBPF 实践

eBPF 的国外巨头实践

- Facebook：Katran 开源负载均衡器， L4LB、DDoS、Tracing
- Netflix：BPF 重度用户，例如生产环境 Tracing。
- Google：Android、服务器安全以及其他很多方面，GKE 默认使用 Cilium 作为网络基础
- Cloudflare：L4LB、DDoS
- Apple：使用 Falcon 识别安全风险
- AWS：使用 eBPF 作为 RPC 观测工具等

……

4. eBPF 实践

eBPF 的国内实践

- 字节跳动：百万主机可观测性探测和 ACL 访问控制列表
- 阿里云 CNI 网卡采用 eBPF 技术观测和故障演练拓扑
- 腾讯：Cilium 作为 TKE 的底层引擎
- 网易：轻舟平台 eBPF 和 Cilium 的实践
- 携程： Cilium + BGP 云原生网络实践

……

1. eBPF 是什么？

5. eBPF 开源项目

- Katran[网络-4层负载均衡]：Facebook 开源的高性能 4 层负载均衡器。Katran 是一个 C++ 库和 eBPF 程序，用于建立一个高性能的第四层负载平衡转发平面。Katran 利用 Linux 内核中的 XDP 基础设施，为快速数据包处理提供内核内设施。

- Cilium[网络/安全/观测]：Isovalent 开源项目，提供由 eBPF 驱动的网络、安全和可观察性。它是专门为 Kubernetes 世界带来 BPF 的优势而设计的，以解决容器工作负载的新的可扩展性、安全性和可视性要求。Isovalent 公司 2020 年 11 月凭借 Cilium 产品获得由 Google/Cisco/Andreessen Horowitz 等公司牵头的 2900 万美元投资。

1. eBPF 是什么？

5. eBPF 开源项目　　Linux IO Visor 基金会管理项目

- BCC［开发工具］：是一个建立在 eBPF 基础上的高效内核跟踪和操作程序的工具包，它包括几个有用的命令行工具和例子。BCC 简化了用 C 语言编写 BPF 程序的过程，包括一个围绕 LLVM 包装器，以及 Python 和 Lua 的前端绑定。

- BPFTrace［开发工具］：是一个用于 Linux eBPF 的高级跟踪语言，语法类似 awk 脚本语言。bpftrace 使用 LLVM 作为后端，将脚本编译成BPF 字节码，并可使用现有的 Linux 跟踪能力和附件点进行交互的库。

- Kubectl-Trace［调试工具］：是一个 kubectl 插件，可以在 Kubernetes 集群中调度 bpftrace 程序的执行。kubectl-trace 不需要在 Kubernetes 集群中直接安装任何组件就可以执行 bpftrace 程序。当指向一个集群时，它安排了一个名为 trace-runner 的临时容器，执行 bpftrace。

1. eBPF 是什么？

5. eBPF 开源项目

- Tracee［安全］：由 Aqua Security 安全公司开源，是一个用 于 Linux 的运行时安全和取证工具。它使用 Linux eBPF 技术在运行时跟踪系统和应用程序，并分析收集的事件以检测可疑的行为模式。

- Falco［安全］：Sysdig 公司开源，为云原生运行时安全项目，是事实上的Kubernetes 威胁检测引擎。Falco 是第一个作为孵化级项目加入CNCF 的运行时安全项目。Falco 就像一个安全摄像头，实时检测意外行为、入侵和数据盗窃。

- eBPF Exporter［可观测］：CloudFlare 公司开源，将 eBPF 技术与 Prometheus 结合。

- Pixie：［可观测］New Relic 公司开源， CNCF 沙盒项目。 Pixie 是一个开源的 Kubernetes 应用程序的可观察性工具。Pixie 使用 eBPF 来自动捕获遥测数据，而不需要手动装置。

1. eBPF 是什么？

5. eBPF 开源项目

- inspektor-gadget：[调试工具]是一个工具（或小工具）的集合，用于调试 Kubernetes 应用程序。虽然它最初是为 Kinvolk 的开源 Kubernetes 发行版 Lokomotive 设计的，在其他 Kubernetes 发行版上也同样适用，底层基于 BCC 代码。（备注： 已经被微软收购）

- bmc-cache （memcache + eBPF) /tcptracer-bpf/bpfilter ……

1. eBPF 是什么？

6. eBPF 限制

- eBPF 程序不能调用任意的内核函数，只限于内核模块中列出的 BPF 辅助函数，函数支持列表也随着内核的演进在不断增加；最新进展是支持了直接调用特定的内核函数调用；

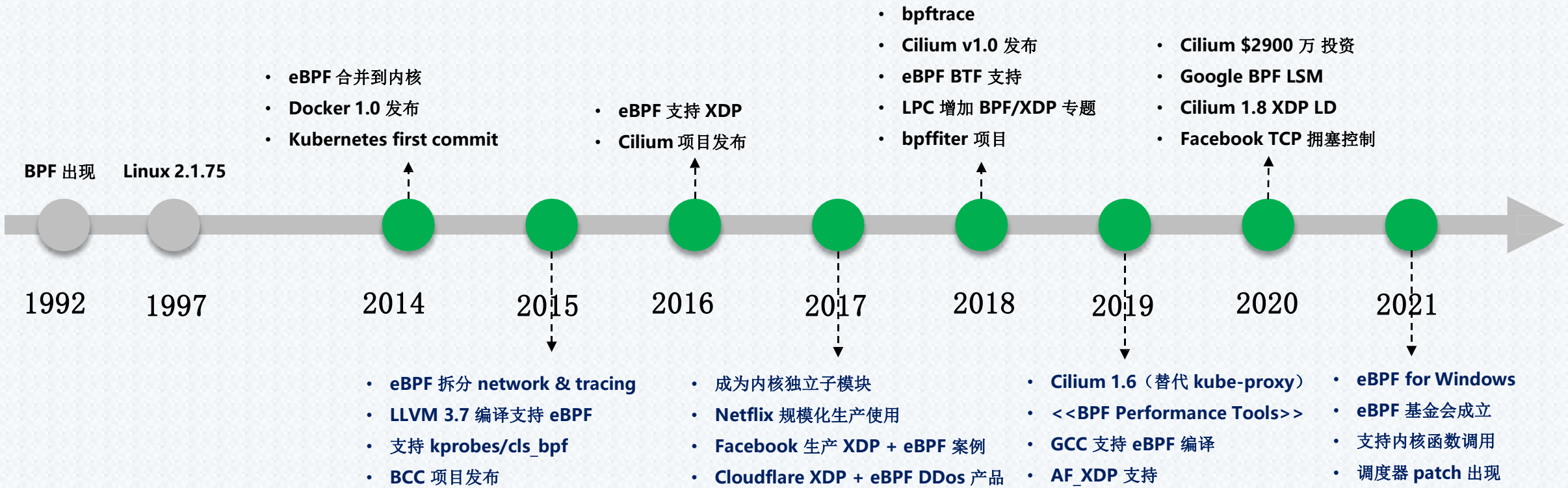
- eBPF 程序不允许包含无法到达的指令，防止加载无效代码，延迟程序的终止；

- eBPF 程序中循环次数限制且必须在有限时间内结束，Linux 5.3 在 BPF 中包含了对有界循环的支持，它有一个可验证的运行时间上限；

- eBPF 堆栈大小被限制在 MAX_BPF_STACK，截止到内核 Linux 5.8 版本，被设置为 512eBPF 字节码大小最初被限制为 4096 条指令，截止到内核 Linux 5.8 版本， 当前已将放宽至 100 万指令 ，对于无特权的 BPF 程序，仍然保留 4096 条限制 (BPF_MAXINSNS)；新版本的 eBPF 也支持了多个 eBPF 程序级联调用，可以通过组合实现更加强大的功能；

1. eBPF 是什么？

1.7 eBPF 年鉴



1. eBPF 是什么？

1.8 eBPF 总结

eBPF：过去50年操作系统最大的变革！！！

1. eBPF 是什么？

1.8 eBPF 总结

347 contributors (Jan 2014 to Jul 2020):

- 588 Daniel Borkmann (Isovalent; maintainer)
- 421 Andrii Nakryiko (Facebook)
- 401 Alexei Starovoitov (Facebook; maintainer)
- 224 Yonghong Song (Facebook)
- 209 Jakub Kicinski (Facebook)
- 183 Martin KaFai Lau (Facebook)
- 179 Stanislav Fomichev (Google)
- 165 John Fastabend (Isovalent)
- 161 Quentin Monnet (Isovalent)
- 130 Jesper Dangaard Brouer (Red Hat)
- 117 Andrey Ignatov (Facebook)
- [...]

Large scale BPF production users:

1. eBPF 是什么？

eBPF 内核社区截至 2020 年 7 月份的一些数据：

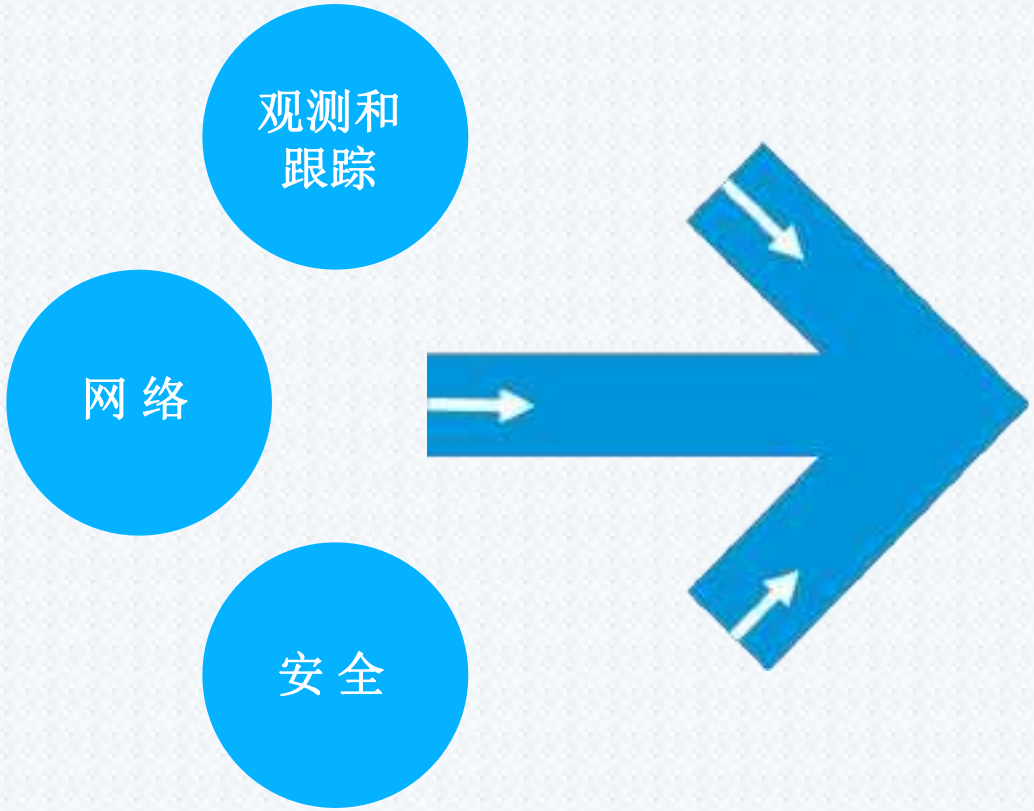
- 347 个贡献者，贡献了 4,935 个 patch 到 BPF 子系统；
- BPF 内核邮件列表日均 50 封邮件（高峰经常超过日均 100）；
  - 23,395 since Feb 2019
- 每天合并 4 个新 patch，接受率 30% 左右；
- 30 个程序类型，27 种 BPF map 类型，141 个 BPF 辅助函数，超过 3,500 个测试；
- 2 个 BPF kernel maintainers & team of 6 senior core reviewers；
  - 主要贡献来自：Isovalent（Cilium），Facebook and Google

毫无疑问，这是内核里发展最快的子系统！

目 录

2. eBPF 的场景有哪些？

2.1 综述



观测和跟踪

将 eBPF 程序附加到跟踪点以及内核和用户应用探针点的能力，使得应用程序和系统本身的运行时行为具有前所未有的可见性。eBPF 不依赖于操作系统暴露的静态计数器和测量，而是实现了自定义指标的收集和内核内聚合，并基于广泛的可能来源生成可见性事件。

网络

可编程性和效率的结合使得 eBPF 自然而然地满足了网络解决方案的所有数据包处理要求。eBPF 的可编程性使其能够在不离开 Linux 内核的包处理上下文的情况下，添加额外的协议解析器，并轻松编程任何转发逻辑以满足不断变化的需求。JIT 编译器提供的效率使其执行性能接近于本地编译的内核代码。

安全

看到和理解所有系统调用的基础上，将其与所有网络操作的数据包和套接字级视图相结合，可以采用革命性的新方法来确保系统的安全。虽然系统调用过滤、网络级过滤和进程上下文跟踪等方面通常由完全独立的系统处理，但 eBPF 允许将所有方面的可视性和控制结合起来，以创建在更多上下文上运行的、具有更好控制水平的安全系统。

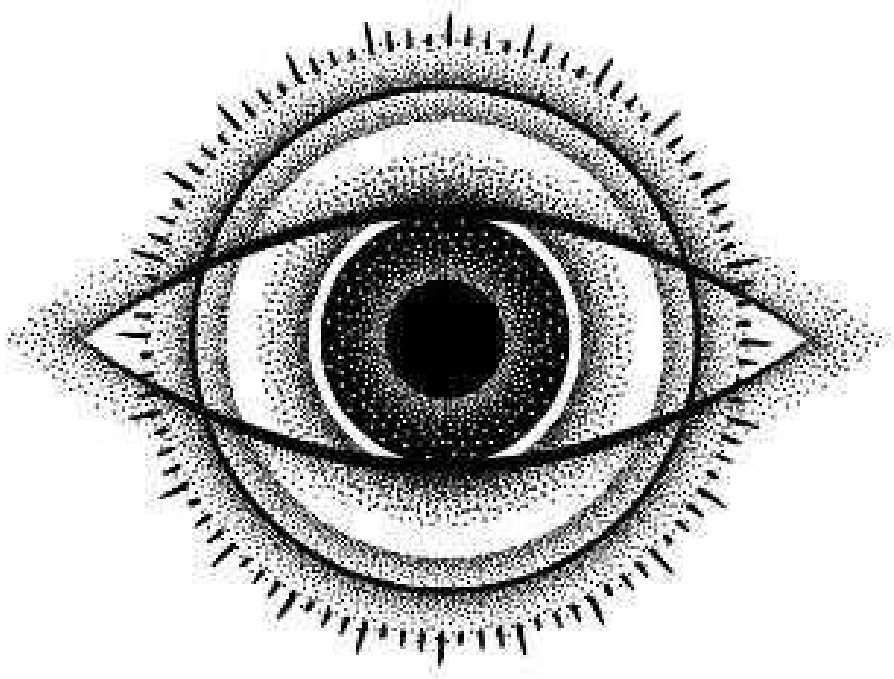
2. eBPF 的场景有哪些？

2.2 观测和跟踪

核 磁 共 振

2. eBPF 的场景有哪些？

2.2 观测和跟踪

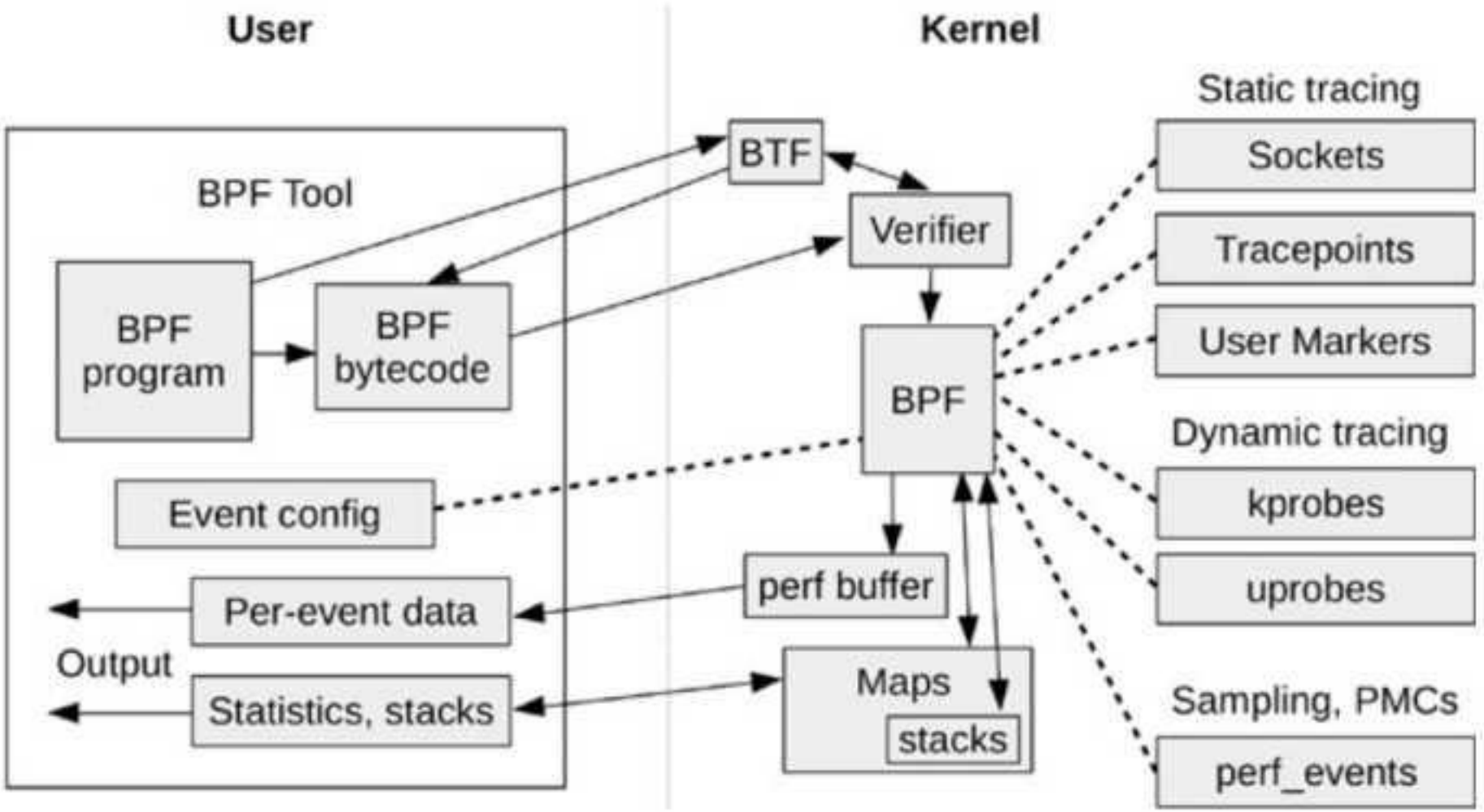


上帝之眼

2. eBPF 的场景有哪些？

2.2 观测和跟踪

eBPF 跟踪架构



2. eBPF 的场景有哪些？

2. 观测和跟踪

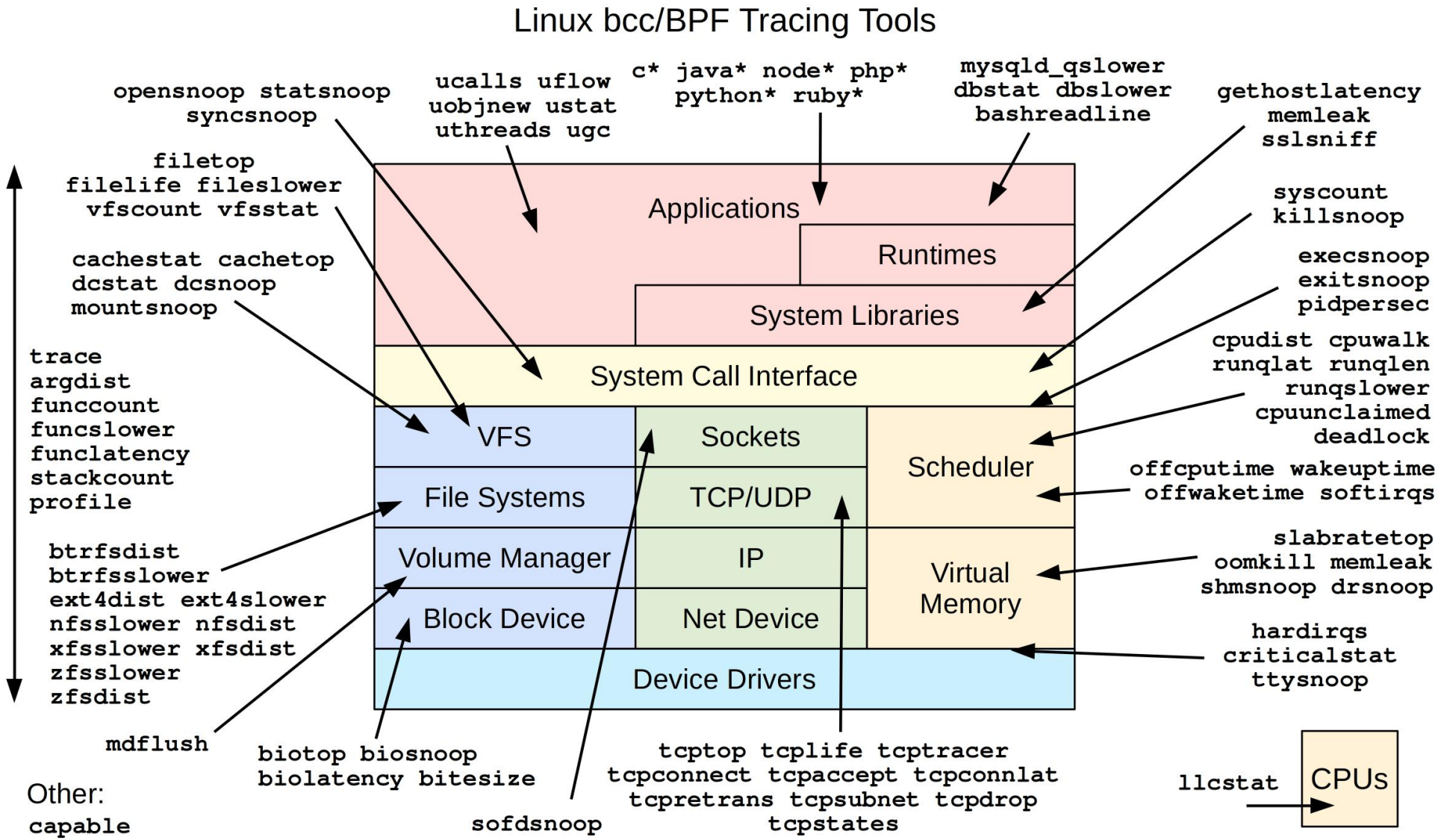
跟踪的事件对象支持：

- **kprobes/kretprobes**：实现内核中动态跟踪。 可跟踪 Linux 内核中的函数入口或返回点，非稳定 ABI 接口；(5.5 fentry/fexit 替代，性能和可用性更好）
- **uprobes/uretprobes**：用户级别的动态跟踪。与 kprobes 类似，只是跟踪的函数为用户程序中的函数；
- **tracepoints**：内核中静态跟踪。tracepoints 是内核开发人员维护的跟踪点，能够提供稳定的 ABI 接口，但是由于是研发人员维护，数量和场景可能受限；
- **perf_events**：定时采样和 PMC；

对于内核文件 /proc/kallsyms 暴露的函数列表，都可以认为使用 kprobes 进行跟踪，内核中 inline 的函数和部分明确屏蔽 kprobe 跟踪的函数无法跟踪，可以理解基本上 Linux 的内核所有函数都可使用 kprobe 跟踪，当前 5.x 内核中导出函数数量在 13万+。

2. eBPF 的场景有哪些？

2.2 观测和跟踪



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

2. eBPF 的场景有哪些？

2.2 观测和跟踪

memleak

memleak 跟踪内存分配和删除请求，并收集每次分配的调用栈。然后，memleak 可打印一份详情，说明哪些地方未进行匹配的释放。

```
# ./memleak -p $(pidof allocs)
Attaching to pid 5193, Ctrl+C to quit.
[11:16:33] Top 2 stacks with outstanding allocations:
        80 bytes in 5 allocations from stack
                main+0x6d [allocs]
                __libc_start_main+0xf0 [libc-2.21.so]

[11:16:34] Top 2 stacks with outstanding allocations:
        160 bytes in 10 allocations from stack
                main+0x6d [allocs]
                __libc_start_main+0xf0 [libc-2.21.so]
```

2. eBPF 的场景有哪些？

2.2 观测和跟踪

execsnoop

execsnoop 通过追踪 execve() 系统调用来实现的，可显示参数和返回值的细节。对于发现临时创建和消亡的非常有用处。

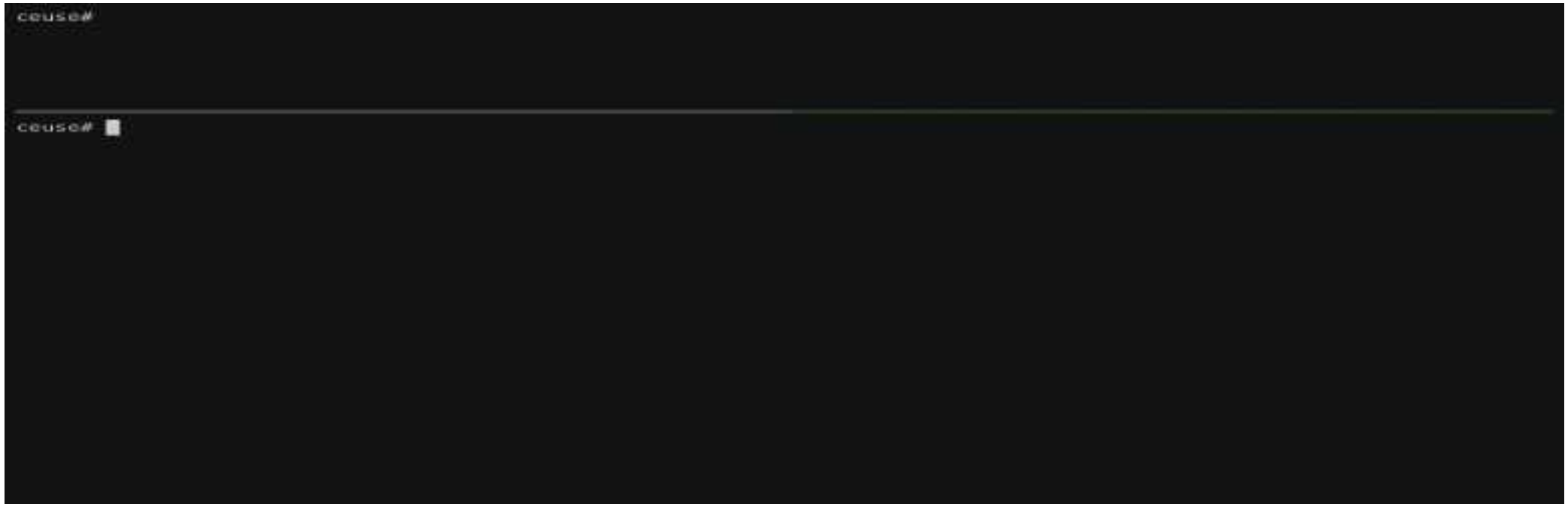
```
# ./execsnoop
PCOMM            PID    PPID   RET ARGS
sleep            3334   21029    0 /usr/bin/sleep 3
sleep            3339   21029    0 /usr/bin/sleep 3
conntrack        3341   1112     0 /usr/sbin/conntrack --stats
conntrack        3342   1112     0 /usr/sbin/conntrack --count
sleep            3344   21029    0 /usr/bin/sleep 3
iptables-save    3347   9211     0 /sbin/iptables-save -t filter
iptables-save    3348   9211     0 /sbin/iptables-save -t nat
```

2. eBPF 的场景有哪些？

2.2 观测和跟踪

pwru

pwru (packet, where are you?) 是一个基于 eBPF 的工具，用于追踪 Linux 内核中的网络数据包，具有方便的过滤功能。Pwru 允许对内核状态进行检查跟踪，方便调试网络连接问题。



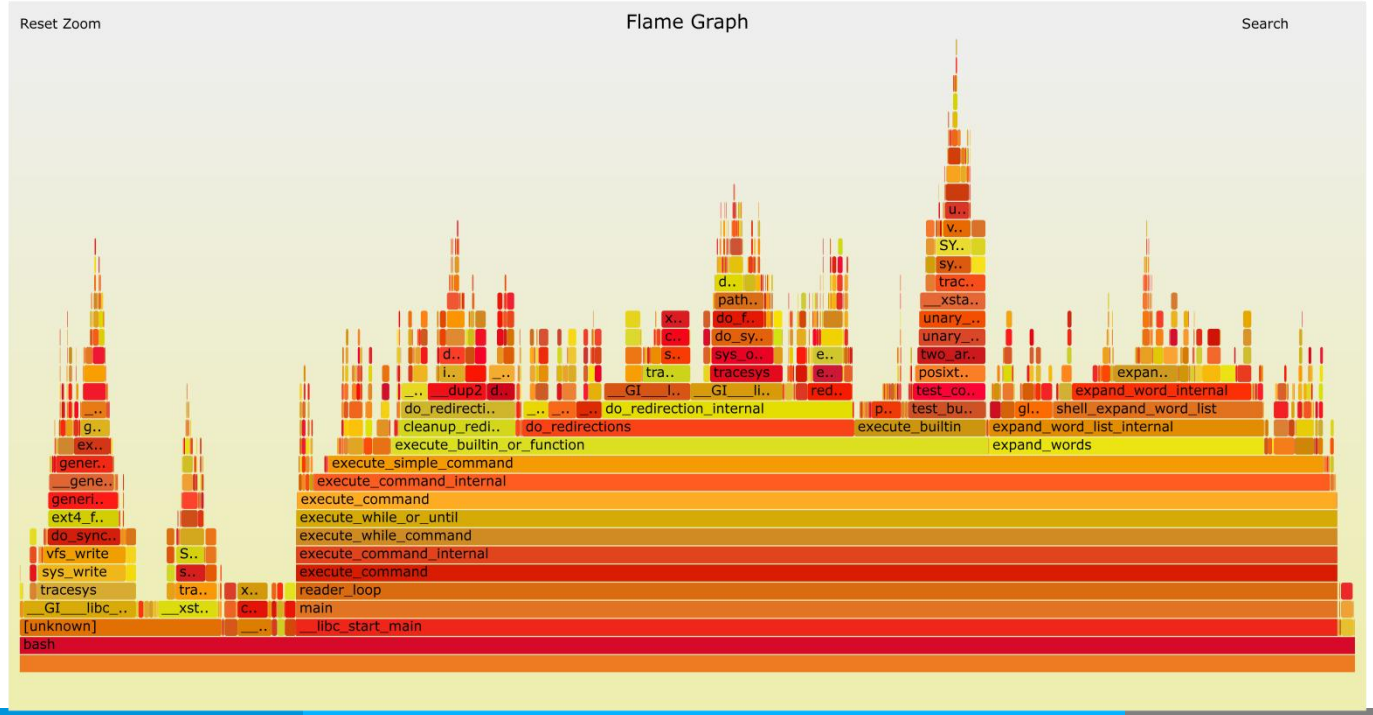
2. eBPF 的场景有哪些？

2.2 观测和跟踪

profile

profile 可在对跟踪程序无任何修改的情况下，搜集 CPU 路径的数据，基于数据采用工具可以轻松地生成火焰图，查找到程序的性能瓶颈。

```
$ profile -af 30 > out.stacks01
$ git clone https://github.com/brendangregg/FlameGraph
$ cd FlameGraph
$ ./flamegraph.pl --color=java < ../out.stacks01 > out.svg
```



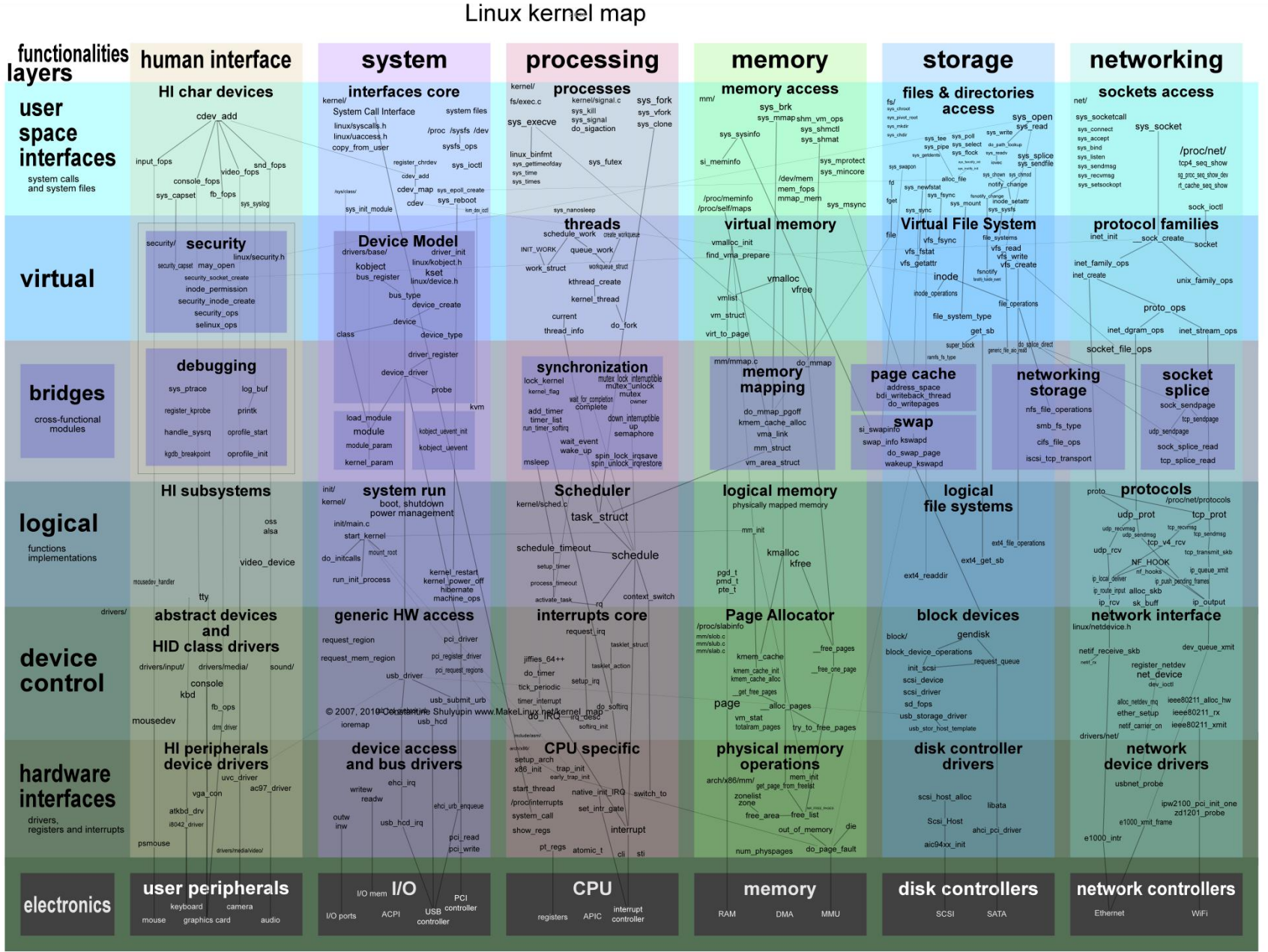
2. eBPF 的场景有哪些？

2.2 观测和跟踪

2. eBPF 的场景有哪些？

2.2 观测和跟踪

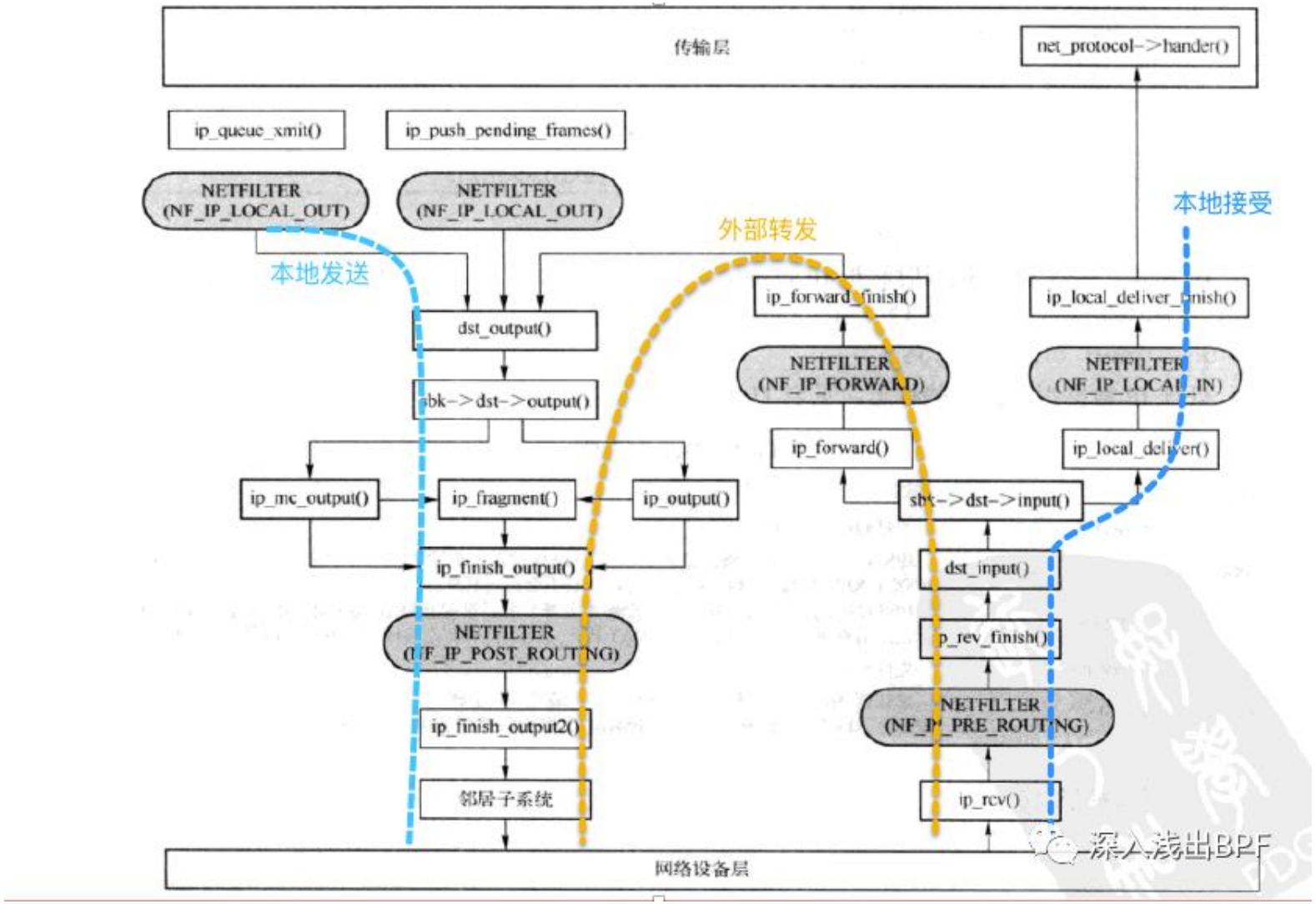
内核全景图



2. eBPF 的场景有哪些？

2.2 观测和跟踪

网络数据包全路径 Trace 跟踪



2. eBPF 的场景有哪些？

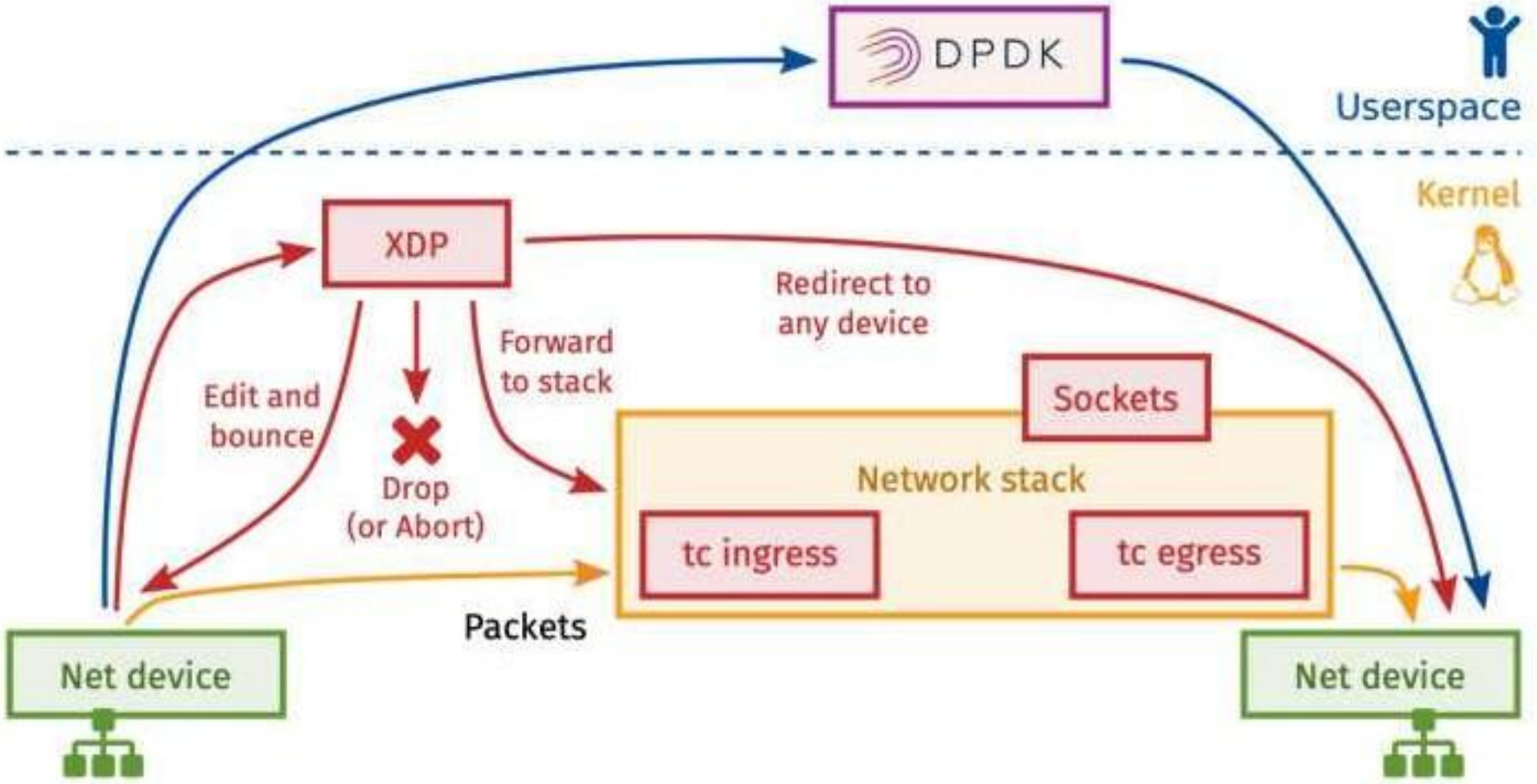
2.2 观测和跟踪

思考题：

- 如何快速定位发送到特定 DNS 请求的进程？
- 如何不需要程序做任何修改得到 Kubernetes 集群的流量拓扑图？

2. eBPF 的场景有哪些？

2.3 网络



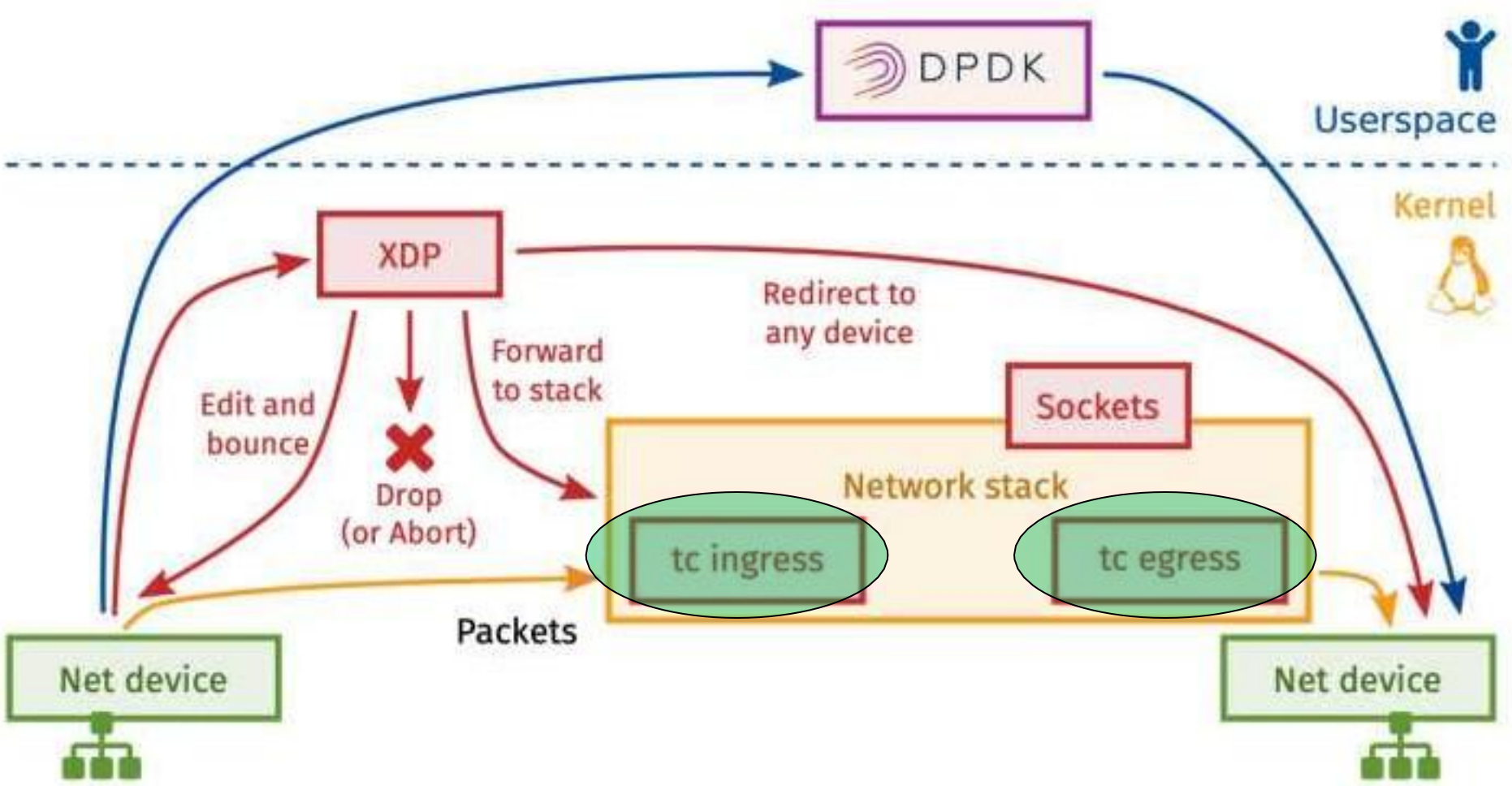
2. eBPF 的场景有哪些？

场景1：网络数据包过滤

BPF_PROG_TYPE_SOCKET_FILTET 类型，与 cBPF 早期的 tcpudump 数据包过滤类似，主要实现实时流量丢弃和特定条件过滤后数据包。

2. eBPF 的场景有哪些？

场景2 - TC 流量控制



2. eBPF 的场景有哪些？

场景2 - TC 流量控制

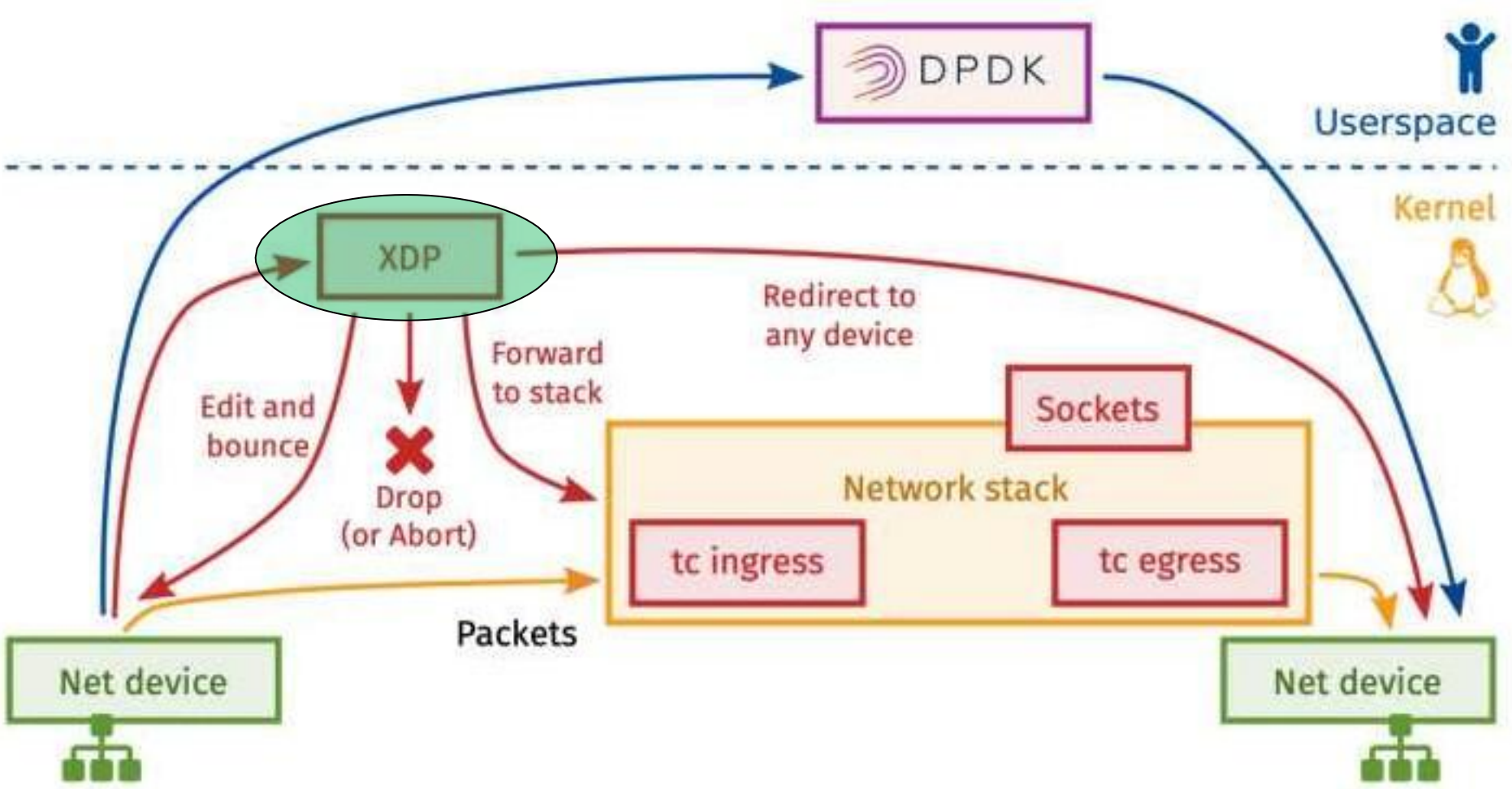
流量控制是一种强大的机制，在所有分类器中，可以编程网络数据路径 cls-bpf 分类器。cls-bpf 可以将 BPF 程序直接挂钩到入口和出口层，从而实现对于数据包流入和流出控制。

网络协议栈会根据挂载在 cls-bpf 中的程序返回值决定数据包的处理方式：

- TC_ACT_OK（0） 终止数据包处理流程，允许处理数据包；
- TCACT_SHOT（2） 终止数据包处理流程，丢弃数据包；
- TC_ACT_UNSPEC（-1） 使用tc配置的默认操作，类似于从一个分类器返回 -1；
- TC_ACT_PIPE（3） 如果有下一个动作，迭代到下一个动作；
- TC_ACT_RECLASSIFY（1） 终止数据包处理流程，从头开始分类；

2. eBPF 的场景有哪些？

场景3 - XDP



2. eBPF 的场景有哪些？

场景3 - XDP

Intel 公司出品的 DPDK（Data Plane Development Kit）套件，本质上 是一种内核 **ByPass** 的方案，将数据包从网卡驱动层直接通过 ZeroCopy 技术复制到用户空间进行处理。

高性能网络场景下 Linux 内核的窘境：

- 内核在处理网络数据的时候采用软中断方式
- 需要构造 sk_buff 结构体，在大量网络数据包场景下会导致系统性能下降
- 用户态和内核态的频繁的上下文切换。

但是，操作系统被 **ByPass** 之后**应用隔离和安全机制**就都失效了；一起失效的还有各种经过已经充分测试的**配置、部署和管理工具**。

2. eBPF 的场景有哪些？

场景3 - XDP

XDP 提供了高性能网络处理的另一种选择，

具有以下优点：

- XDP 提供了一个**仍然基于操作系统内核**的安全执行环境，在**设备驱动上下文**中执行，可用于定制各种包处理应用。
- XDP 是**内核的一部分**，与现有的内核网络栈完全兼容，二者协同工作。
- XDP 应用通过 C 等高层语言编写，然后编译成特定字节码；出于安全考虑，内核会首先对这些字节码执行静态分析，然后再将它们翻译成 **处理器原生指令**

2. eBPF 的场景有哪些？

场景3 - XDP

XDP 常见的使用场景：

- 三层路由转发
- 四层负载均衡 （Facebook Katran，比 IPVS 性能高 4.3 倍）
- DDoS 防护
- 分布式防火墙
- 访问控制 ACL

2. eBPF 的场景有哪些？

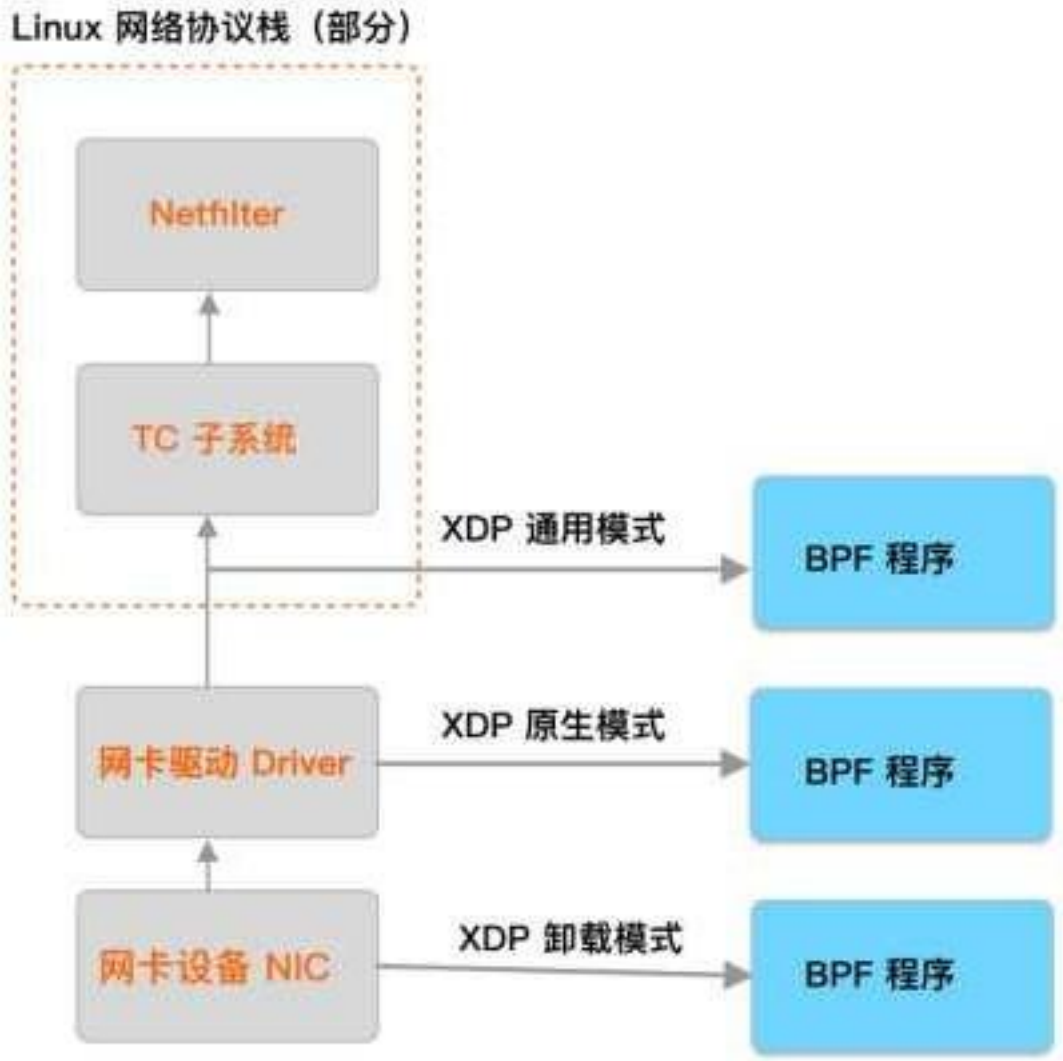
场景3 - XDP

XDP 支持 3 种工作模式：

- **原生 XDP**（Native），默认工作模式，该模式下， XDP BPF 程序在网络驱动的早期接受路径之外运行，需要驱动支持；
- **卸载 XDP**（OffLoad）， XDP 的 BPF 程序直接卸载到网卡，而不是主机的 CPU，比原生 XDP 性能更高；
- **通用 XDP**（Generic），如果没有原生或者卸载 XDP 的功能，可以使用通用 XDP 进行测试，内核 4.12 版本以后支持，可在 veth 设备上测试；该函数处理在 **receive_skb()** 之后；

2. eBPF 的场景有哪些？

场景3 - XDP



2. eBPF 的场景有哪些？

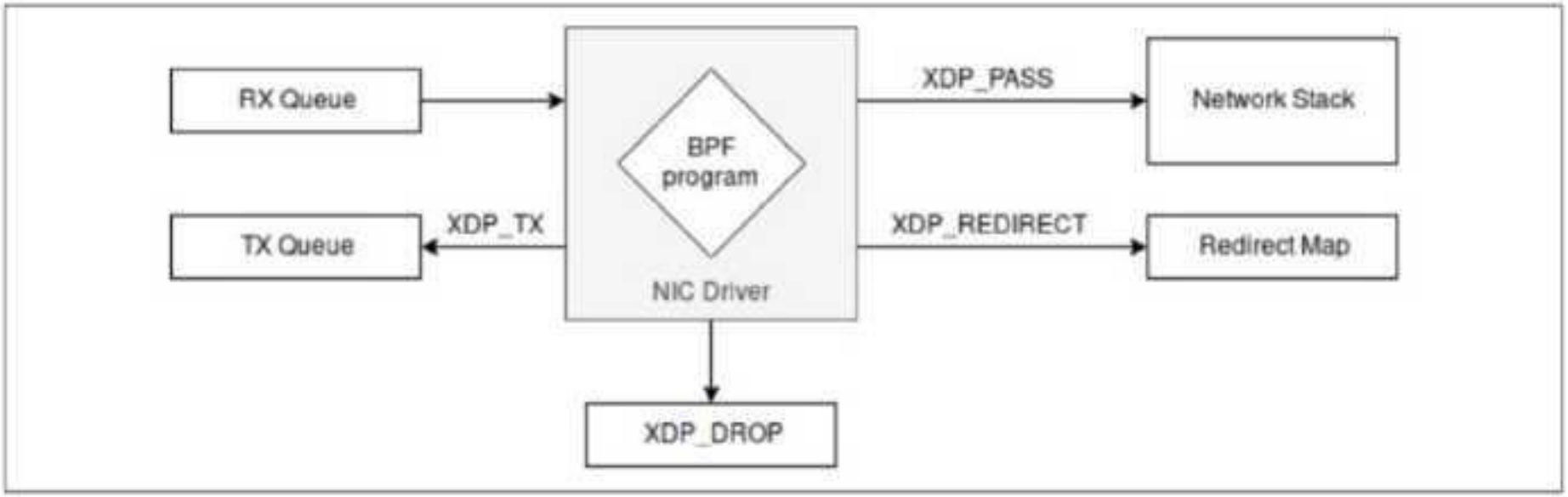
场景3 - XDP

XDP BPF 程序处理结果：

- **XDP_DROP** 丢弃数据包，这通常发生在驱动程序最早的 RX 阶段，对于降低 Dos 场景而言，尽早丢弃包是关键，这可以尽量少的占用 CPU 的处理时间；
- **XDP_TX**：转发数据包，可能发生在数据包被修改前或修改后，将处理后的数据包发回给相同的网卡；
- **XDP_PASS**：将数据包传递到普通网络协议栈处理，与没有 XDP BPF 程序运行的效果一致；
- **XDP_REDIRECT**：与 XDP_TX 类似，只是转发的目的地可以为其他 CPU 处理队列、不同的网卡或转发到特定的用户空间 socket（AF_XDP），具体依赖于 Redirect Map 的设置；
- **XDP_ABORTED** 表示 BPF 程序错误，并导致数据包对丢弃，可通过 trace_xdp_exception 跟踪点进行额外监控；

2. eBPF 的场景有哪些？

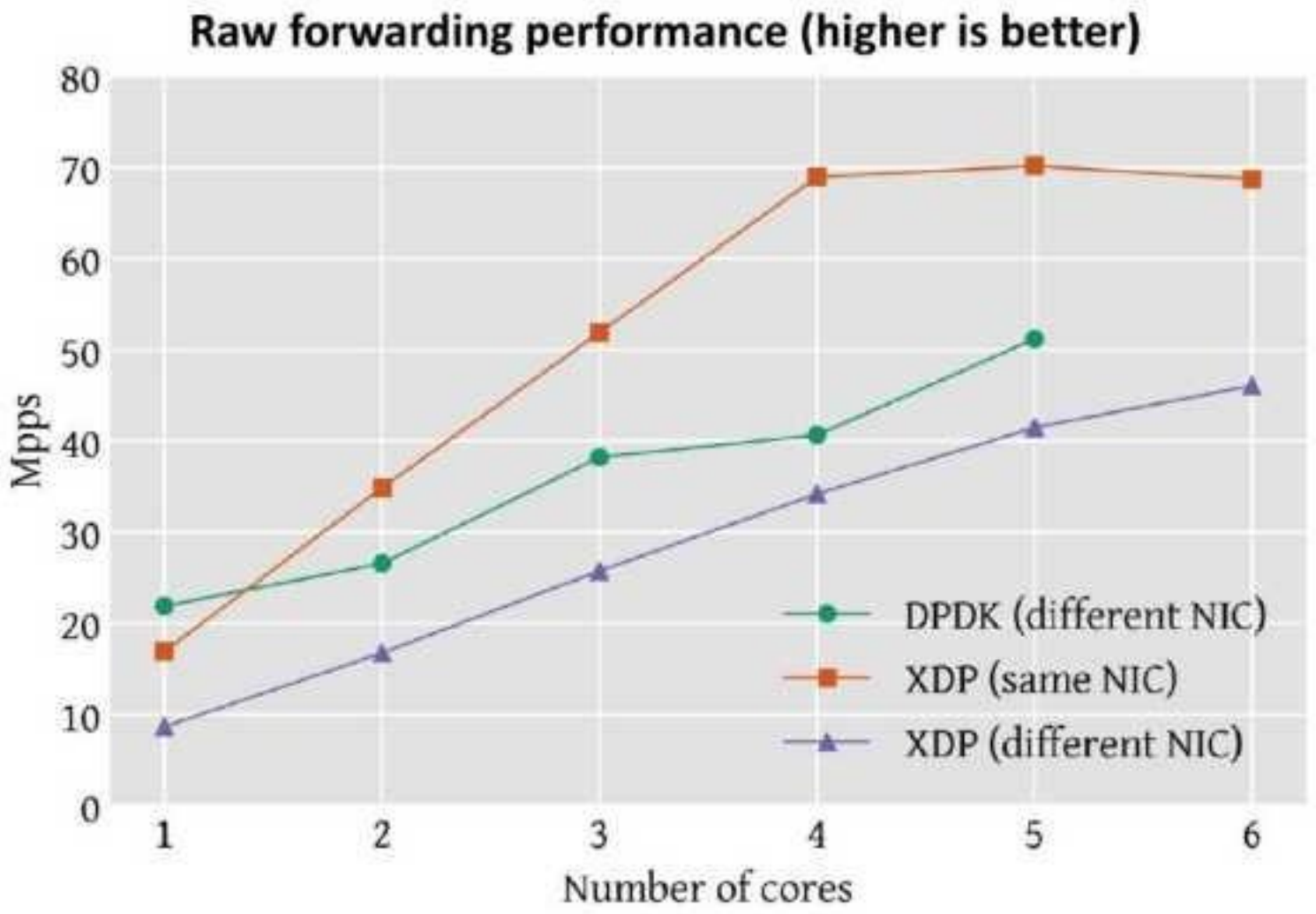
场景3 - XDP



2. eBPF 的场景有哪些？

场景3 - XDP

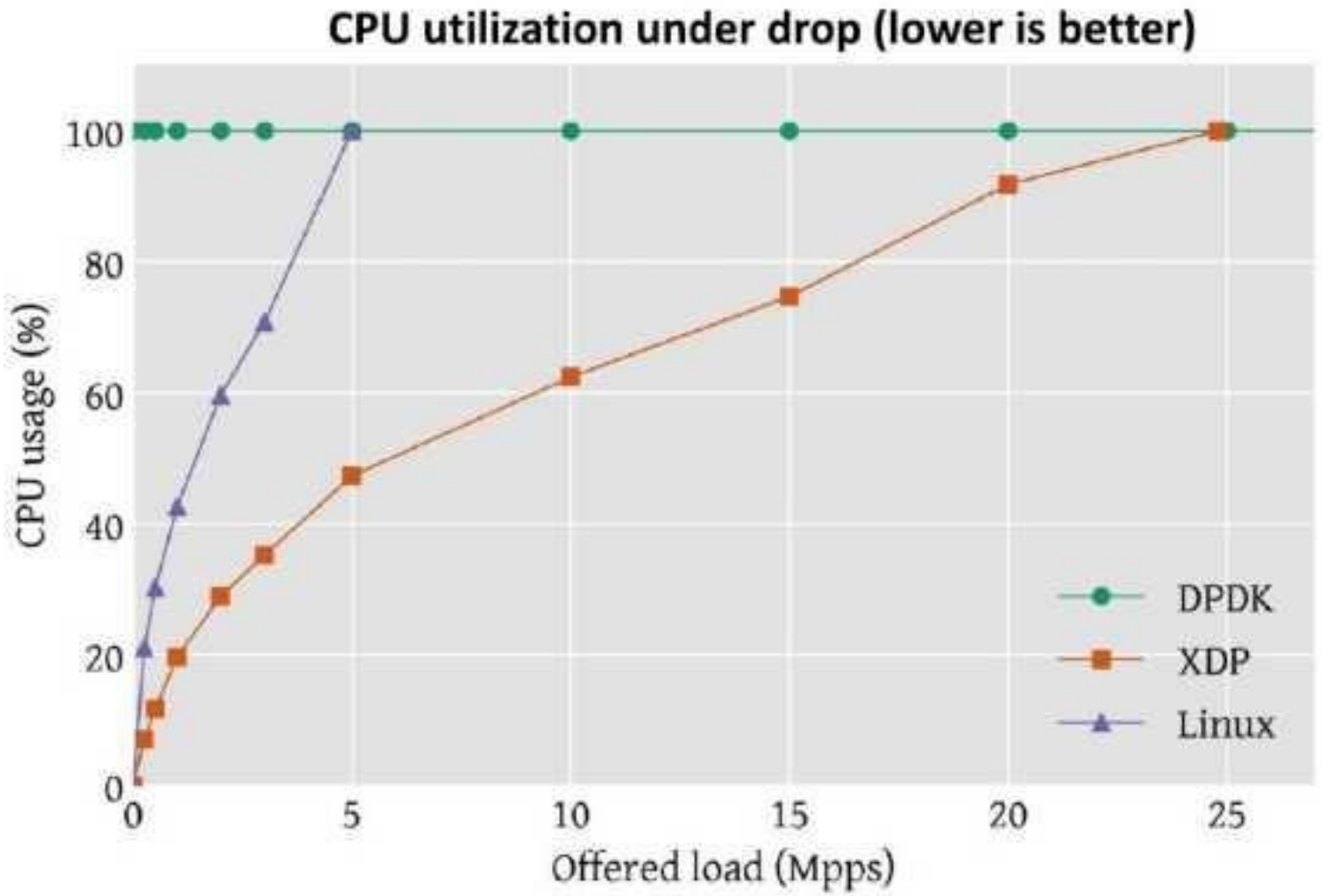
DPDK 与 XDP 转发吞吐量对比



2. eBPF 的场景有哪些？

场景3 - XDP

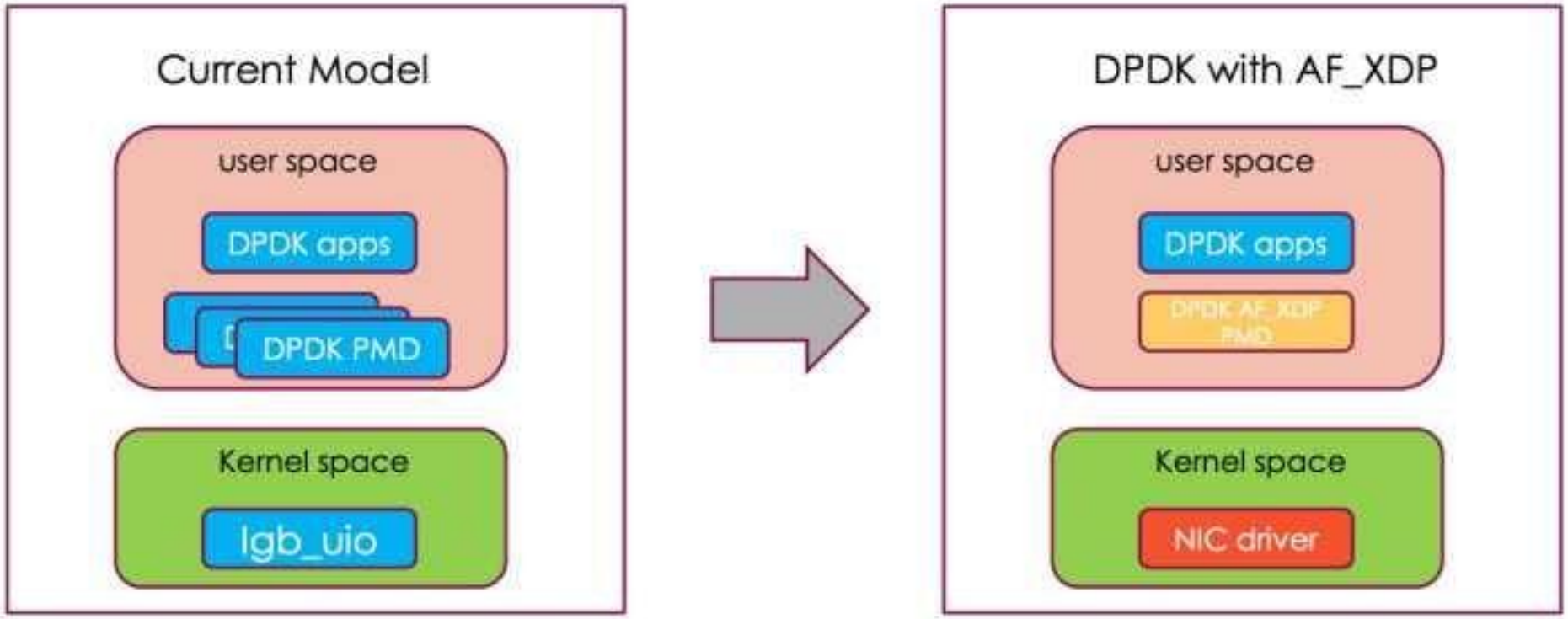
DPDK 与 XDP 转发 CPU 使用对比



2. eBPF 的场景有哪些？

场景3 - XDP

DPDK 与 XDP 融合方案



2. eBPF 的场景有哪些？

2.3 安全

Seccomp (Secure Computing)

Seccomp 是 Linux 内核中实现的安全层，用于开发人员过滤特定的系统调用，基于 cBPF。Seccomp 过滤基于 BPF 过滤器，采用 SECCOMP_MODE_FILTER 模式，而系统调用过滤的方式与数据包的过滤方式相同。可以基于系统上下文（用户、权限、能力）等对系统调用进行过滤。

KRSI (Kernel Runtime Security Instrumentation)

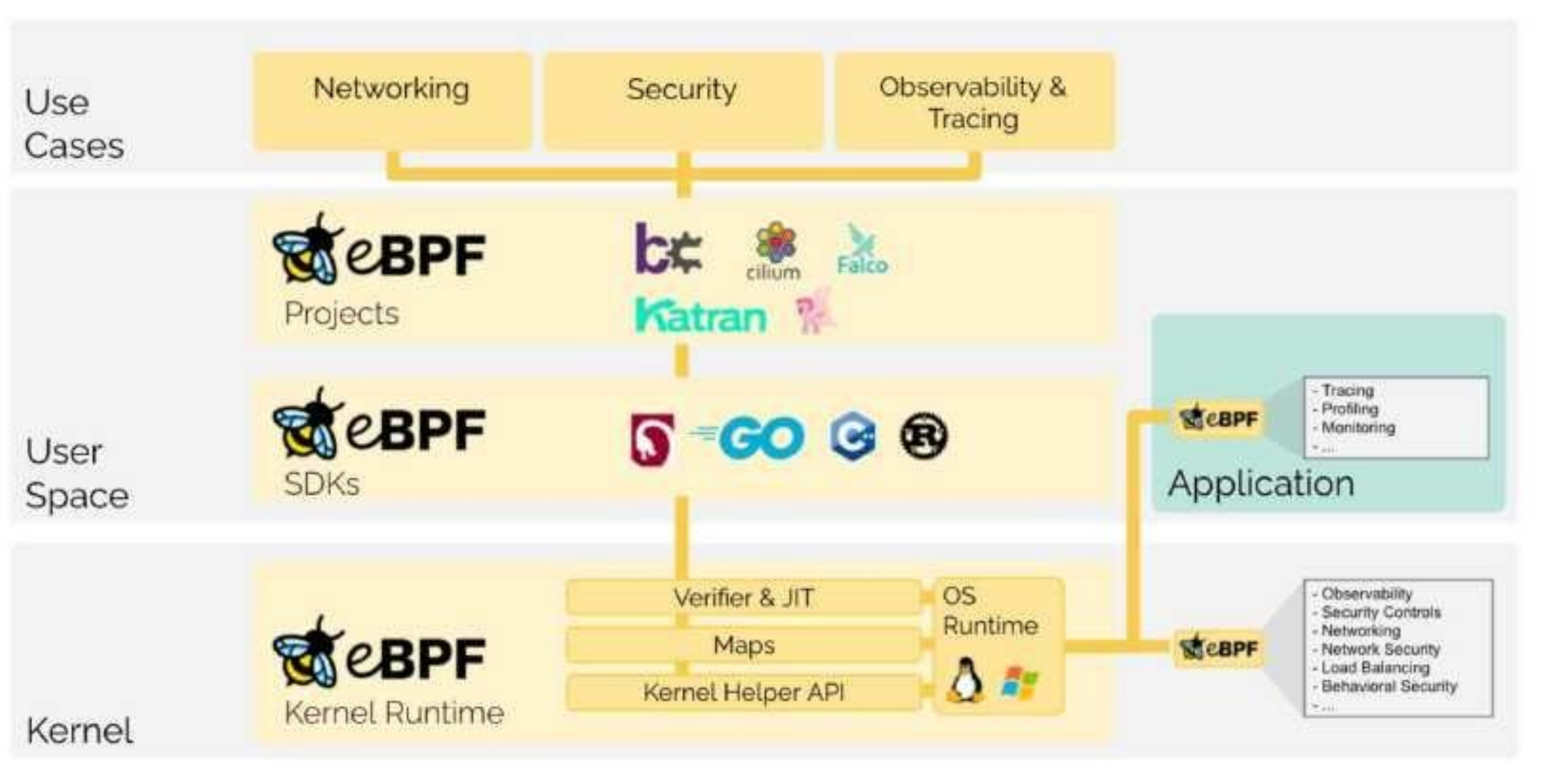
内核运行时安全工具（KRSI）旨在提供一个可扩展的 LSM，允许特权用户将 eBPF 程序附加到安全钩上，以动态地实现 MAC 和审计策略。 Google 主力支持。

Tracee/Falco 等基于 eBPF 的安全产品 .....

目 录

3. eBPF 是怎么工作的？

3.1 eBPF 全景图



3. eBPF 是怎么工作的？

2. eBPF 工作原理

eBPF 使用与 cBPF 相同，都是基于事件驱动架构。

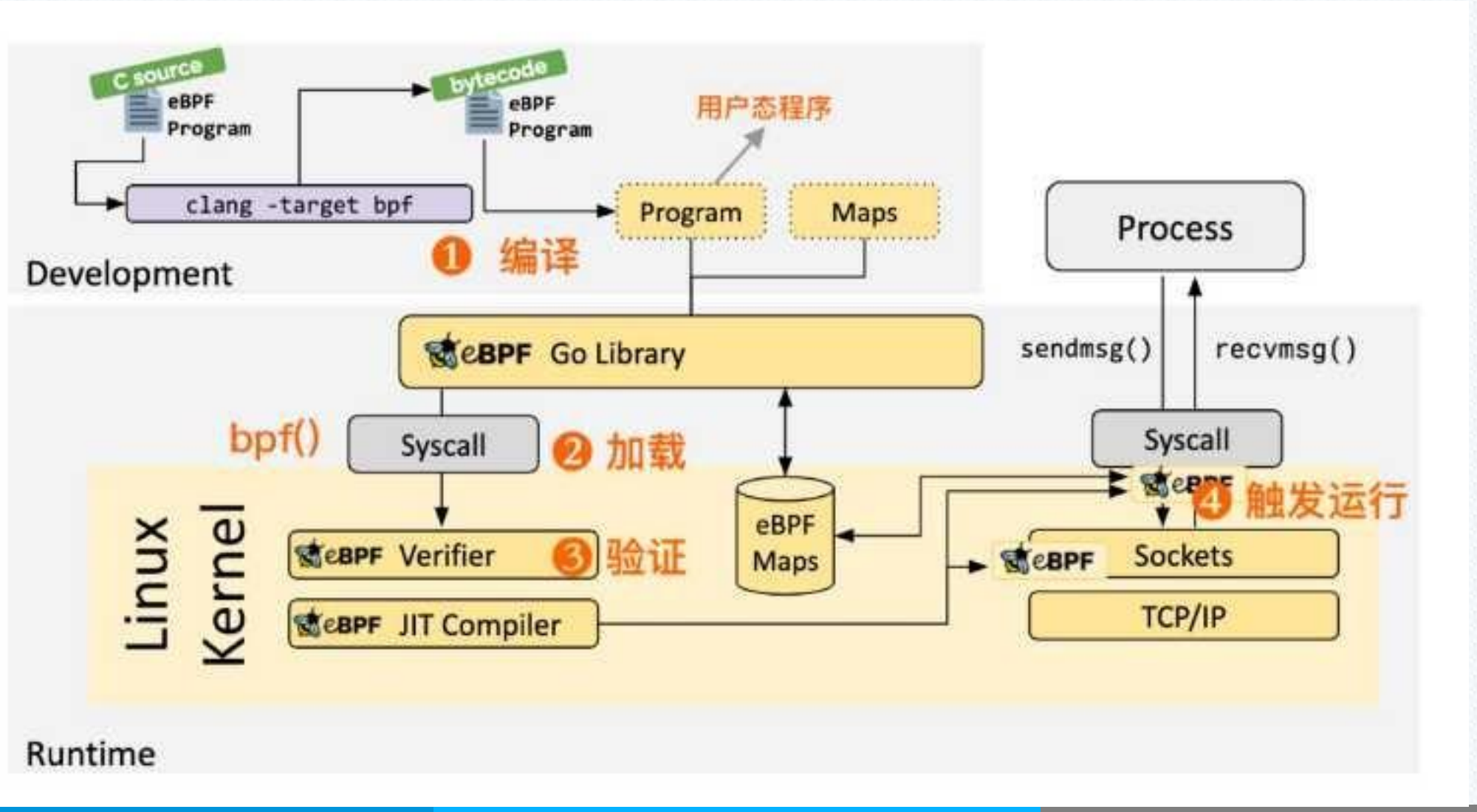
eBPF 触发的钩子点可以为：

- 用户空间代码的调用
- 内核例程在特定内存地址的执行
- 网络数据包的到达或者数据包流转路径上
- 系统调用
- ……

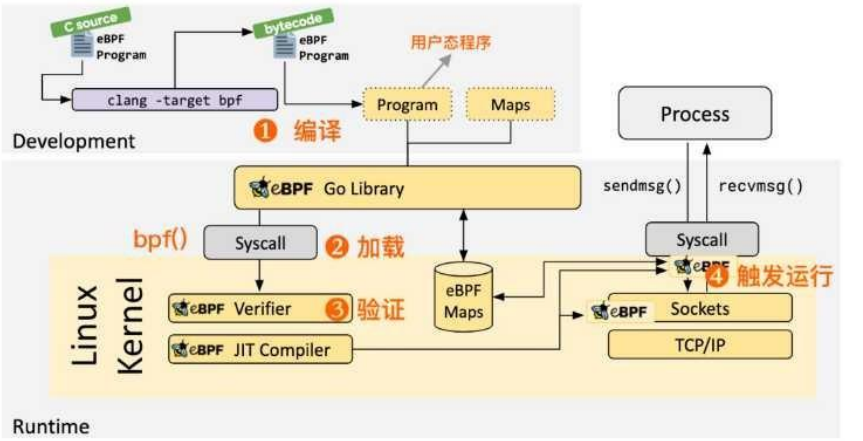
在触发点条件满足时，则会触发 eBPF 程序的运行。

3. eBPF 是怎么工作的？

3.2 eBPF 工作原理



3. eBPF 是怎么工作的？



3.2 eBPF 工作原理

1. 编译：我们可以使用 LLVM 或者 GCC 工具将编写的 eBPF 代码程序编译成 eBPF 字节码；

2. 加载：然后使用加载程序 Loader 通过 bpf() 系统调用将字节码加载至内核；

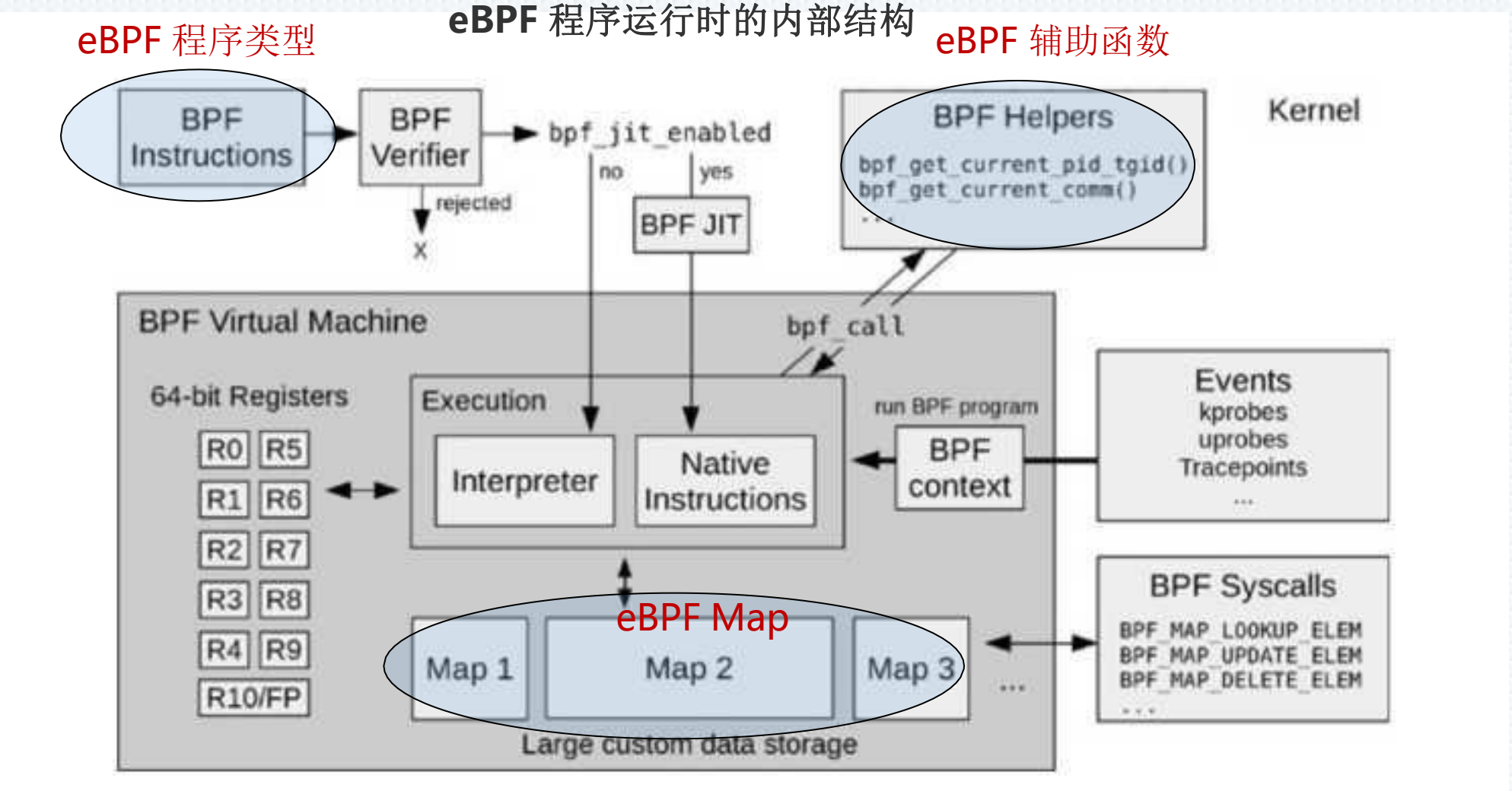
3.验证：内核使用验证器（Verfier）组件保证执行字节码的安全性，以避免对内核造成灾难，在确认字节码安全后将其加载对应的内核模块执行；验证过程：

- 首先进行**深度优先搜索**，禁止循环；其他 CFG 验证。
- 以上一步生成的指令作为输入，**遍历所有可能的执行路径**。具体说 就是模拟每条指令的执行，**观察寄存器和栈的状态变化**。

4. 触发运行：在内核对应事件触发时，运行的 eBPF 字节码程序，可通过 map 或者 perf_event 与用户态程序进行数据交互；

3. eBPF 是怎么工作的？

3.2 eBPF 工作原理



3. eBPF 是怎么工作的？

3.3 eBPF 辅助函数

由于 eBPF 程序不能随意调用内核函数，因此内核专门提供 eBPF 程序可以调用的辅助函数，5.15 最新内核有 175 个函数。可以在 Linux 系统中通过 "man bpf-helpers 5.13" 查看，文档按照添加的时间排序，根据惯例，辅助函数不能超过 5 个参数。

辅助函数与程序类型有强对应关系。辅助函数为不同的 Linux 内核版本引入，辅助函数与内核版本对应关系参见这里。

linux/include/uapi/linux/bpf.h

```
5335  #define __BPF_FUNC_MAPPER(FN)          \
5336          FN(unspec),                    \
5337          FN(map_lookup_elem),           \
5338          FN(map_update_elem),           \
5339          FN(map_delete_elem),           \
5340          FN(probe_read),                \
5341          FN(ktime_get_ns),              \
5342          FN(trace_printk),              \
5343          FN(get_prandom_u32),           \
5344          FN(get_smp_processor_id),      \
5345          FN(skb_store_bytes),           \
5346          FN(l3_csum_replace),           \
5347          FN(l4_csum_replace),           \
5348          FN(tail_call),                 \
5349          FN(clone_redirect),            \
5350          FN(get_current_pid_tgid),      \
5351          FN(get_current_uid_gid),       \
5352          FN(get_current_comm),          \
5353          FN(get_cgroup_classid),        \
5354          FN(skb_vlan_push),             \
5355          FN(skb_vlan_pop),              \
5356          FN(skb_get_tunnel_key),        \
5357          FN(skb_set_tunnel_key),        \
5358          FN(perf_event_read),           \
5359          FN(redirect),                  \
5360          FN(get_route_realm),           \
```

3. eBPF 是怎么工作的？

3.3 eBPF 辅助函数

辅助函数名称	描述
bpf_trace_printk()	调试信息，向 tracefs 打印数据；/sys/kernel/debug/tracing/trace_pipe
bpf_probe_read_kernel()	从内核地址从读取数据
bpf_get_current_comm()	获取调用的进程名称
bpf_get_current_task()	返回当前 task 结构体，包含了运行进程的细节信息
bpf_get_stackid()	获取调用的栈信息
bpf_perf_event_output	将数据写入 perf_event 环形缓冲区，对于每个事件输出
bpf_map_lookup_elem	从 map 中查询数据
...	

3. eBPF 是怎么工作的？

3. eBPF 辅助函数 （部分样例）

bpf_trace_printk

long bpf_trace_printk(const char *fmt, u32 fmt_size, ...)

该函数的速度较慢，应该只用于调试使用，避免在生产环境使用，在生产环境中应该使用 perf 事件

函数限制如下：

- 最大只支持 3 个参数，而且只运行一个 %s 的参数；
- 程序共享输出共享 /sys/kernel/debug/tracing/trace_pipe 文件，可能导致文件输出错乱；
- 该实现方式在数据量大的时候，性能也存在一定的问题；

bpf_probe_write_user

尝试以一种安全的方式向用户态空间写数据，可以修改用户空间函数的参数，比如读文件返回的内容等；

3. eBPF 是怎么工作的？

3.4 eBPF 程序类型

BPF 程序的类型定义了 BPF 程序可以挂载的**事件类型**以及事件的参数，也限定了可以访问**辅助函数的集合**，程序类型与协助函数的对应关系参见这里。截止到目前 Linux 内核 5.15 版本有 31 个，归结为 31 种使用场景。程序类型与事件参数的部分样例如下：

linux/include/uapi/linux/bpf.h

```
921  */
922  enum bpf_prog_type {
923          BPF_PROG_TYPE_UNSPEC,
924          BPF_PROG_TYPE_SOCKET_FILTER,
925          BPF_PROG_TYPE_KPROBE,
926          BPF_PROG_TYPE_SCHED_CLS,
927          BPF_PROG_TYPE_SCHED_ACT,
928          BPF_PROG_TYPE_TRACEPOINT,
929          BPF_PROG_TYPE_XDP,
930          BPF_PROG_TYPE_PERF_EVENT,
931          BPF_PROG_TYPE_CGROUP_SKB,
932          BPF_PROG_TYPE_CGROUP_SOCK,
933          BPF_PROG_TYPE_LWT_IN,
934          BPF_PROG_TYPE_LWT_OUT,
935          BPF_PROG_TYPE_LWT_XMIT,
936          BPF_PROG_TYPE_SOCK_OPS,
937          BPF_PROG_TYPE_SK_SKB,
938          BPF_PROG_TYPE_CGROUP_DEVICE,
939          BPF_PROG_TYPE_SK_MSG,
940          BPF_PROG_TYPE_RAW_TRACEPOINT,
```

程序类型	事件参数
BPF_PROG_TYPE_SOCKET_FILTER	bpf_prog(**struct __sk_buff** *skb)
BPF_PROG_TYPE_KPROBE	bpf_prog(**struct pt_regs** *ctx)
BPF_PROG_TYPE_TRACEPOINT	取决于跟踪 tracepoint 定义类型
BPF_PROG_TYPE_XDP	bpf_prog(**struct xdp_md** *xdp)
...	....

3. eBPF 是怎么工作的？

3.4 eBPF 程序类型

程序类型	辅助函数
BPF_PROG_TYPE_SOCKET_FILTER	BPF_FUNC_skb_load_bytes() BPF_FUNC_skb_load_bytes_relative() BPF_FUNC_get_socket_cookie() BPF_FUNC_get_socket_uid() BPF_FUNC_perf_event_output() Base functions
BPF_PROG_TYPE_KPROBE	BPF_FUNC_perf_event_output() BPF_FUNC_get_stackid() BPF_FUNC_get_stack() BPF_FUNC_perf_event_read_value() BPF_FUNC_override_return() Tracing functions
...	...

$ git grep -W 'func_proto(enum bpf_func_id func_id' kernel/ net/ drivers/

完整参见这里。

3. eBPF 是怎么工作的？

3.5 eBPF map 类型

BPF 程序最神奇的功能之一就是：

内核运行的代码与加载其的用户空间程序可以通过 map 机制实现 **双向实时通信**。

eBPF Map 以 key/value 对保存在内核中，可以被任何 eBPF 程序访问；用户空间程序通过导出的文件描述符进行访问。Map 定义支持不同类型的数据类型结构，5.15 版本中有 29 种不同类型结构。

linux/include/uapi/linux/bpf.h

```
878  };
879
880  enum bpf_map_type {
881          BPF_MAP_TYPE_UNSPEC,
882          BPF_MAP_TYPE_HASH,
883          BPF_MAP_TYPE_ARRAY,
884          BPF_MAP_TYPE_PROG_ARRAY,
885          BPF_MAP_TYPE_PERF_EVENT_ARRAY,
886          BPF_MAP_TYPE_PERCPU_HASH,
887          BPF_MAP_TYPE_PERCPU_ARRAY,
888          BPF_MAP_TYPE_STACK_TRACE,
889          BPF_MAP_TYPE_CGROUP_ARRAY,
890          BPF_MAP_TYPE_LRU_HASH,
891          BPF_MAP_TYPE_LRU_PERCPU_HASH,
892          BPF_MAP_TYPE_LPM_TRIE,
893          BPF_MAP_TYPE_ARRAY_OF_MAPS,
894          BPF_MAP_TYPE_HASH_OF_MAPS,
895          BPF_MAP_TYPE_DEVMAP,
896          BPF_MAP_TYPE_SOCKMAP,
897          BPF_MAP_TYPE_CPUMAP,
898          BPF_MAP_TYPE_XSKMAP,
899          BPF_MAP_TYPE_SOCKHASH,
900          BPF_MAP_TYPE_CGROUP_STORAGE,
```

3. eBPF 是怎么工作的？

3.5 eBPF map 类型

map 类型支持 Hash、Array、lpm、Socket 等多种类型;

需要注意的是在 Linux 内核中操作 map 是一个原子操作，但是通过用户空间的操作是通过文件描述符，通过 bpf() 系统调用可以理解是一个异步的过程;

对于 map 操作的并发控制，可以通过 bpf_spin_lock 进行锁定;

目 录

4. eBPF 软件开发生态

1. eBPF 程序组成

eBPF 程序的高层次组件：

- **后端**：这是在内核中加载和运行的 eBPF 字节码。它将数据写入内核 map 和环形缓冲区的**数据结构**中；
- **加载器**：它将字节码**后端**加载到内核中。通常情况下，当加载器进程终止时，字节码会被内核自动卸载；
- **前端**：从**数据结构**中读取数据（由**后端**写入）并将其显示给用户；
- **数据结构**：这些是**后端**和**前端**之间的通信手段。它们是由内核管理的 map 和环形缓冲区，可以通过文件描述符访问，并需要在**后端**被加载之前创建，数据会持续存在，直到没有更多的**后端**或**前端**进行读写操作；

4. eBPF 软件开发生态

4.1 eBPF 程序组成

```c
static int test_sock(void)
{
    int sock = -1, map_fd, prog_fd, i, key;
    long long value = 0, tcp_cnt, udp_cnt, icmp_cnt;

    map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key), sizeof(value), 256, 0);

    struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */),
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        // ...
    };

    size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
    prog_fd = bpf_load_program(BPF_PROG_TYPE_SOCKET_FILTER, prog, insns_cnt,
                               "GPL", 0, bpf_log_buf, BPF_LOG_BUF_SIZE);

    sock = open_raw_sock("lo");
    setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

    for (i = 0; i < 10; i++) {
        key = IPPROTO_TCP;
        assert(bpf_map_lookup_elem(map_fd, &key, &tcp_cnt) == 0);

        // ...
        sleep(1);
    }
    // ...
}
```

数据结构

后端 BPF 程序

加载器

前端

4. eBPF 软件开发生态

4.2 BPFTrace

bpftrace 是一种用于 eBPF 的高级跟踪语言，使用 LLVM 作为后端，将脚本编译为 BPF 字节码，语言的灵感来自于awk 和 C，以及诸如 DTrace 和 SystemTap 这样的跟踪器。

统计 sys_enter 系统调用进入的进程 pid 和 comm 命令行，并进行聚合统计：

```
$ bpftrace -e 'tracepoint:raw_syscalls:sys_enter {@[pid, comm] = count();}'
```

4.3 BCC

为了简化 BPF 程序开发，社区创建了 BCC 项目：其为编写、加载和运行 eBPF 程序提供了一个易于使用的框架，除了 “限制性 C” 之外，还可以通过编写简单的 Python 或 Lua 脚本来实现。

BCC 提供了大量跟踪和观测工具，涉及内核 CPU、Mem、调度和网络等子系统。

BPF 后端程序我们也可以看到是使用了面向对象 C 语言的方法，这是 BCC 提供的一层语法糖，在运行前会进行一次统一解析，再编译成 BPF 字节码。

以下样例是使用 BCC 使用 Python 前端绑定编写的统计 syscall 的样例。

4. eBPF 软件开发生态

4.3 BCC

4. eBPF 软件开发生态

4.4 C 语言 - 原生

底层基于 libbpf 库实现用户态程序的编写和 BPF 程序的加载

```c
int main(int ac, char **argv)
{
    // ...
    snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);

    bpf_prog_load(filename, BPF_PROG_TYPE_SOCKET_FILTER, &obj, &prog_fd);

    map_fd = bpf_object__find_map_fd_by_name(obj, "my_map");

    sock = open_raw_sock("lo");
    setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

    f = popen("ping -4 -c5 localhost", "r");

    for (i = 0; i < 5; i++) {
        long long tcp_cnt, udp_cnt, icmp_cnt;

        int key = IPPROTO_TCP;
        assert(bpf_map_lookup_elem(map_fd, &key, &tcp_cnt) == 0);

        // ...

        sleep(1);
    }

    return 0;
}
```

加载器

前端

sockex1_user.c

4. eBPF 软件开发生态

4.4 C 语言 - 原生

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

// ...

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, long);
    __uint(max_entries, 256);
} my_map SEC(".maps");

SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
    int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
    long *value;

    if (skb->pkt_type != PACKET_OUTGOING)
        return 0;

    value = bpf_map_lookup_elem(&my_map, &index);
    if (value)
        __sync_fetch_and_add(value, skb->len);

    return 0;
}
char _license[] SEC("license") = "GPL";
```

数据

后端 BPF 程序

sockex1_kern.c

4.5 C 语言 - libbpf-bootstrap

libbpf-bootstrap 是一个脚手架工具，它为初级用户设置了尽可能多的东西，可直接进入编写 BPF 程序。其集成了过去几年 BPF 社区开发的最佳实践，并提供了一个现代和方便的工作流程，可以说是迄今为止最好的 BPF 用户体验。 libbpf-bootstrap 依赖于 libbpf 并使用一个简单的 Makefile。

4. eBPF 软件开发生态

4.5 C 语言 - libbpf-bootstrap

```c
int main(int argc, char **argv)
{
    struct minimal_bpf *skel;
    int err;

    /* Open BPF application */
    skel = minimal_bpf__open();

    /* ensure BPF program only handles write() syscalls from our process */
    skel->bss->my_pid = getpid();

    /* Load & verify BPF programs */
    minimal_bpf__load(skel);


    /* Attach tracepoint handler */
    minimal_bpf__attach(skel);


    for (;;) {
        /* trigger our BPF program */
        fprintf(stderr, ".");
        sleep(1);
    }

cleanup:
    minimal_bpf__destroy(skel);
    return -err;
}
```

1
2
3
4

4. eBPF 软件开发生态

4.5 C 语言 - libbpf-bootstrap

```
SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
        int pid = bpf_get_current_pid_tgid() >> 32;

        if (pid != my_pid)
                return 0;

        bpf_printk("BPF triggered from PID %d.\n", pid);

        return 0;
}
```

6. 用户空间程序 Go 语言编写

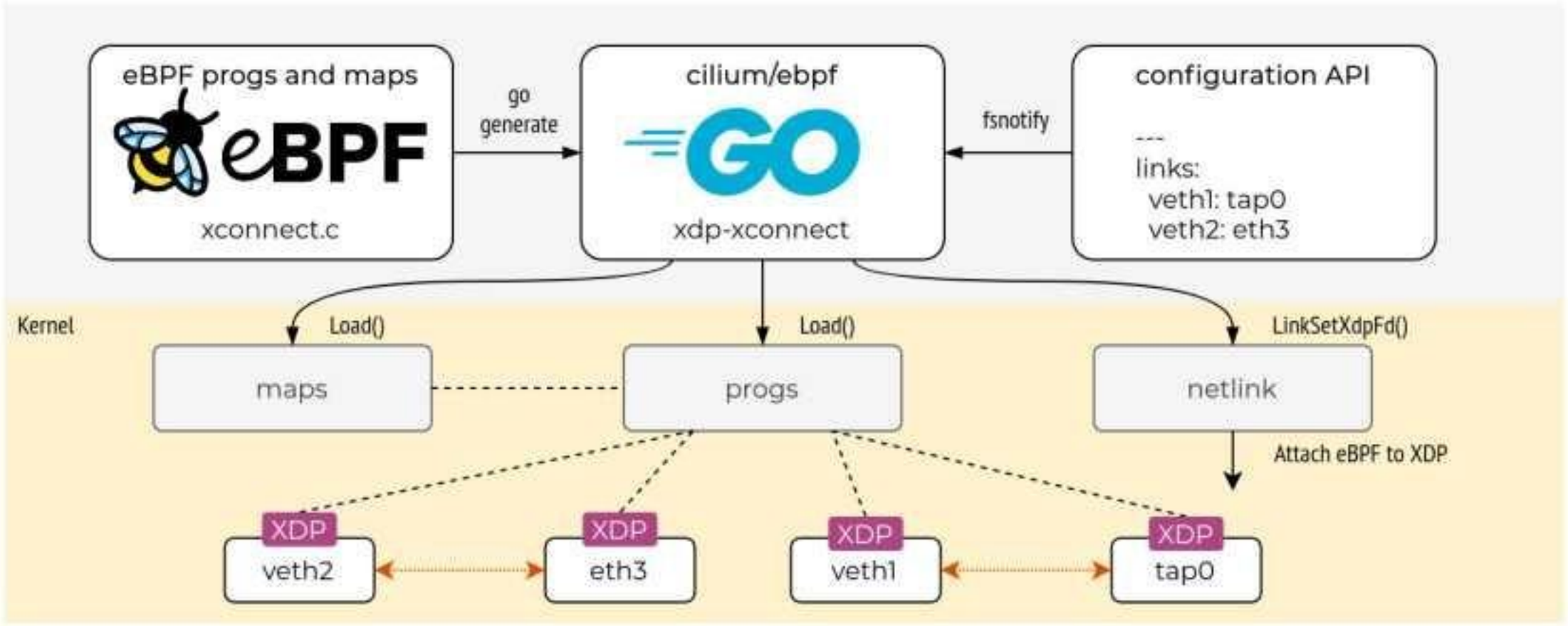
Go 语言编写的程序一般采用静态编译，分发和部署相对方便，如果将 BPF 编译后的字节码直接内嵌到 Go 程序中，分发则更加方便。

当前在加载和与 BPF 程序交互方面，Go 语言 SDK 主要分为两种方式：

- 采用 cgo 的模式，通过 libbpf/libcc 等 C 语言的库实现，如 iovisor/gobpf ；
- 采用 Go 原生实现 BPF 程序加载和交互，如 cilium/ebpf;

4. eBPF 软件开发生态

4.6 用户空间程序 Go 语言编写



4. eBPF 软件开发生态

4.6 用户空间程序 Go 语言编写

仓库名	描述	贡献者/核心贡献者	创建时间	更新时间	stars	备注
cilium/ebpf	eBPF Library for Go	43/7	2019-09-05	2021-08-10	1523	纯 Go 语言实现
dropbox/goebpf	Library to work with eBPF programs from Go	8/1	2019-01-24	2021-08-08	723	底层依赖libbpf（ cgo ）
iovisor/gobpf	Go bindings for creating BPF programs.	49/3	2016-11-18	2021-08-10	1409	底层依赖 libbcc/libbpf(cgo)
aquasecurity/libbpfgo	eBPF library for Go, wrapping libbpf	8/2	2020-11-8	2012-08-10	150	底层依赖 libpf（cgo)

4. eBPF 软件开发生态

6. 程序分发模式

- **源代码分发**

用户空间程序和 BPF 程序在目标主机运行前编译和运行；BPFftrace 和 BCC 由于采用的是更高层次语言编写，是基于源码分发，因此需要 LLVM/Clang 编译器一起部署，而且在部分生产环境中部署时，由于需要编译可能导致系统有抖动影响生产服务；

- **二进制分发**

二进制文件包括用户空间程序和编译后的 BPF 字节码；

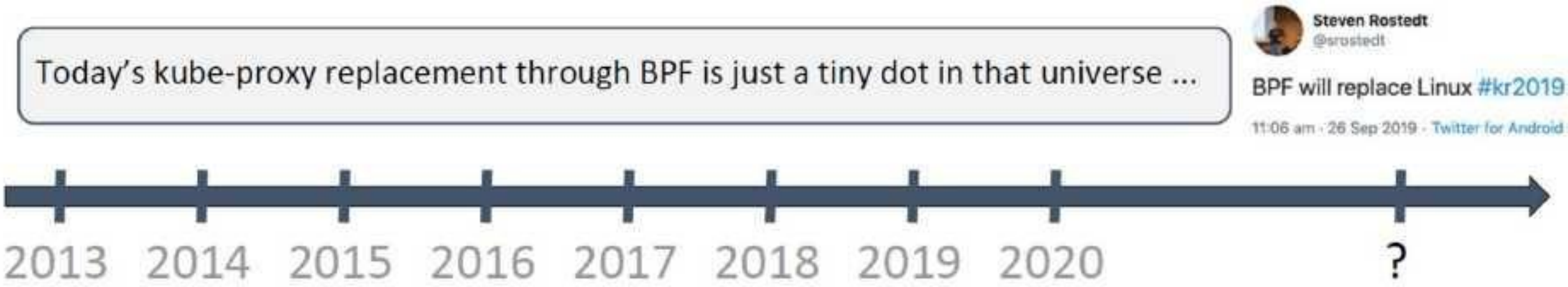
  - 编译机与目标机器环境完全相同；
  - 编译机与目标机器环境内核版本不同；

目 录

5. eBPF 的未来

我们的目标是星辰大海，与此相比，**kube-proxy** 替换只是最微不足道的开端。

BPF will replace Linux！！！



5. eBPF 的未来

CO-RE（Compile-Once Run-Everywhere）

BTF 和 CO-RE 在运行时消除了 LLVM/Clang 和内核头文件依赖关系，不仅使 BPF 在嵌入式 Linux 环境中更加实用，而且在任何地方都可以使用。其中：

- **BTF**：BPF 类型格式，它提供结构信息以避免 Clang 和内核头文件依赖。
- **CO-RE**：它使已编译的 BPF 字节码可重定位，从而避免了 LLVM 重新编译的需要。LLVM 和 Clang 编译后的结果是一个轻量级的 ELF 二进制文件，其中包含预编译的 BPF 字节码，并且可以在任何地方运行。BCC 项目中的大部分工具已经完成基于 CO-RE 的重写。

使用 CO-RE 的能力需要内核使用以下的编译选项： CONFIG_DEBUG_INFO_BTF = y

Ubuntu 20.10 已经将此配置选项设置为默认选项，所有其他发行版都应遵循。发行维护者的注意事项：它需要 pahole >= 1.16。

BTF 和 CO-RE 这两项新技术为 BPF 成为价值十亿美元的产业铺平了道路。

5. eBPF 的未来

1. 软件定义内核

内核面临的挑战：

- **复杂度性不断增长，性能和可扩展性新需求**
- **永远保持后向兼容**

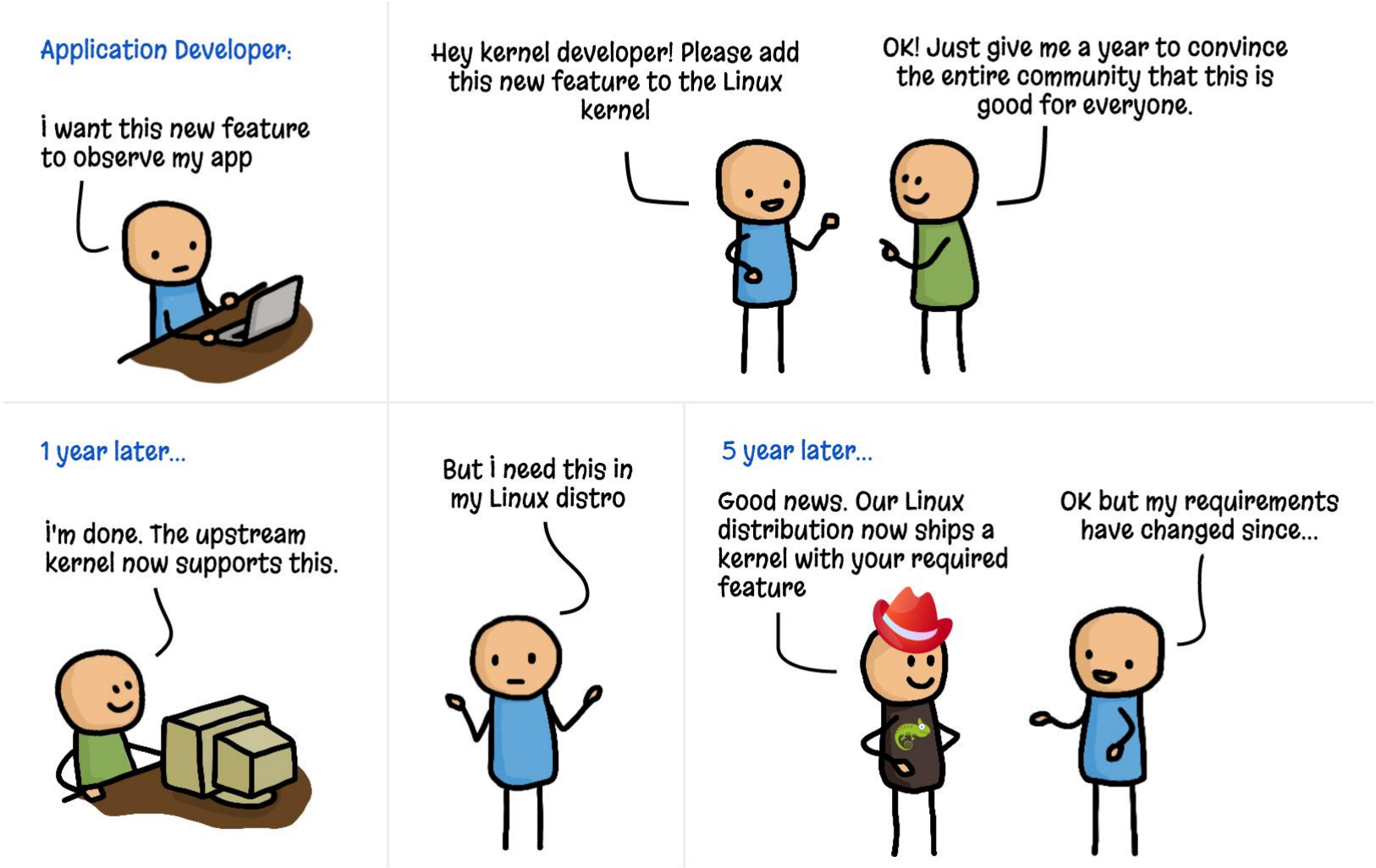
  Never break user space。使原本就复杂的内核变得更加复杂，对于网络来说，这意味着**快速收发包路径（fast path）将受到影响**。

- **特征渐进式演进**

内核不允许一次性引入非常大的改动，需要将它们拆分成数量众多的小 patch，每次合并的 patch 保证系统后向兼容，并且对系统的影响非常小。

5. eBPF 的未来

5.1 软件定义内核



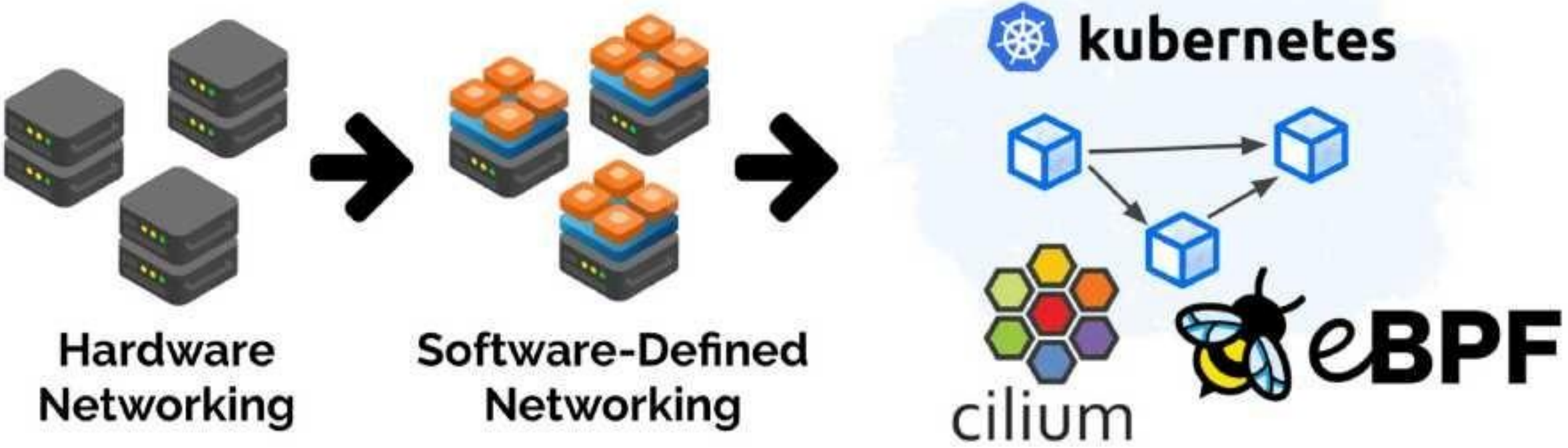
5. eBPF 的未来

5.1 软件定义内核

eBPF 内核完全可编程，安全地可编程。eBPF 是内核功能快速迭代和功能更加多样化的基石。

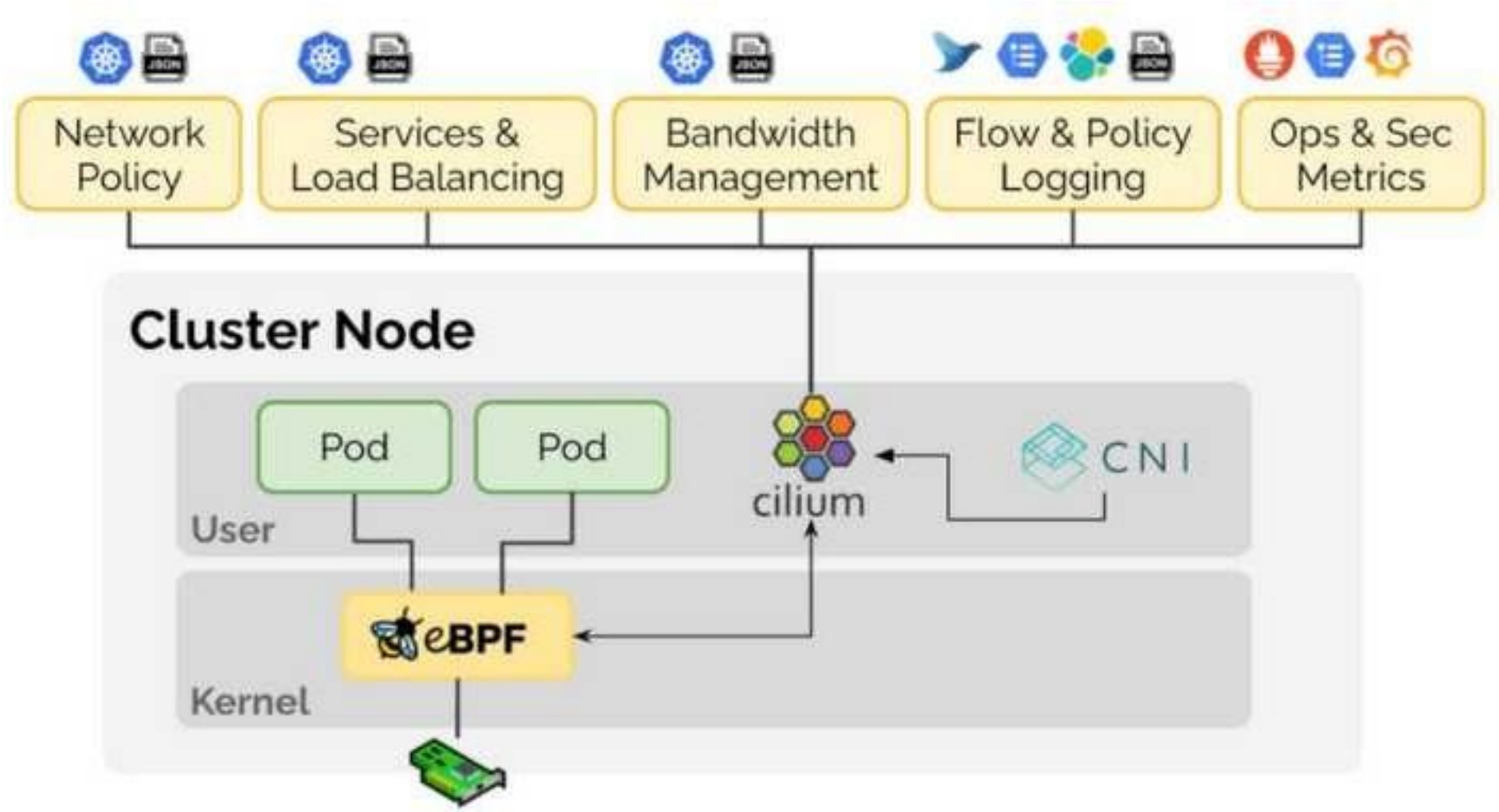
5. eBPF 的未来

5.2 eBPF 网络和安全的未来



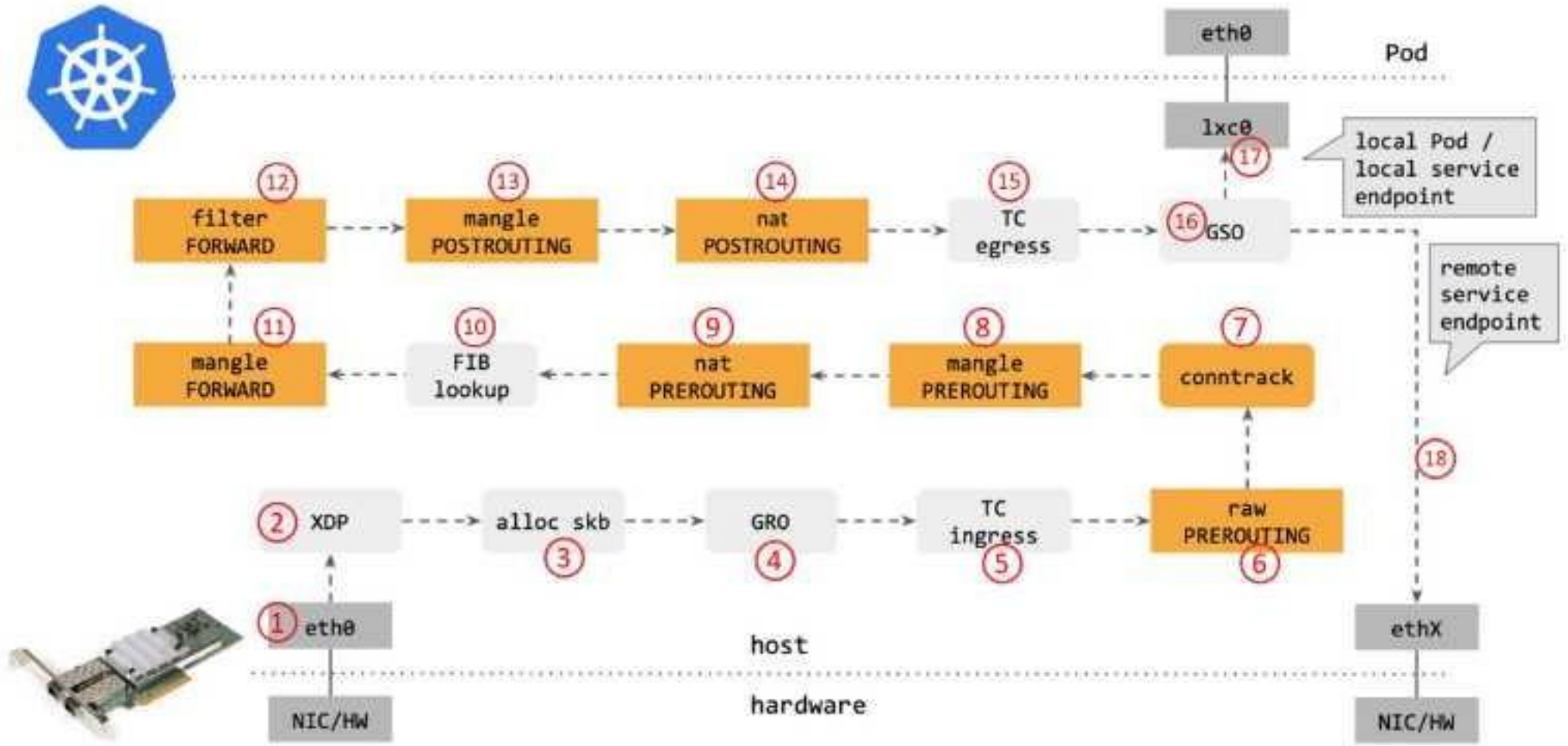
5. eBPF 的未来

5.2 eBPF 网络和安全的未来



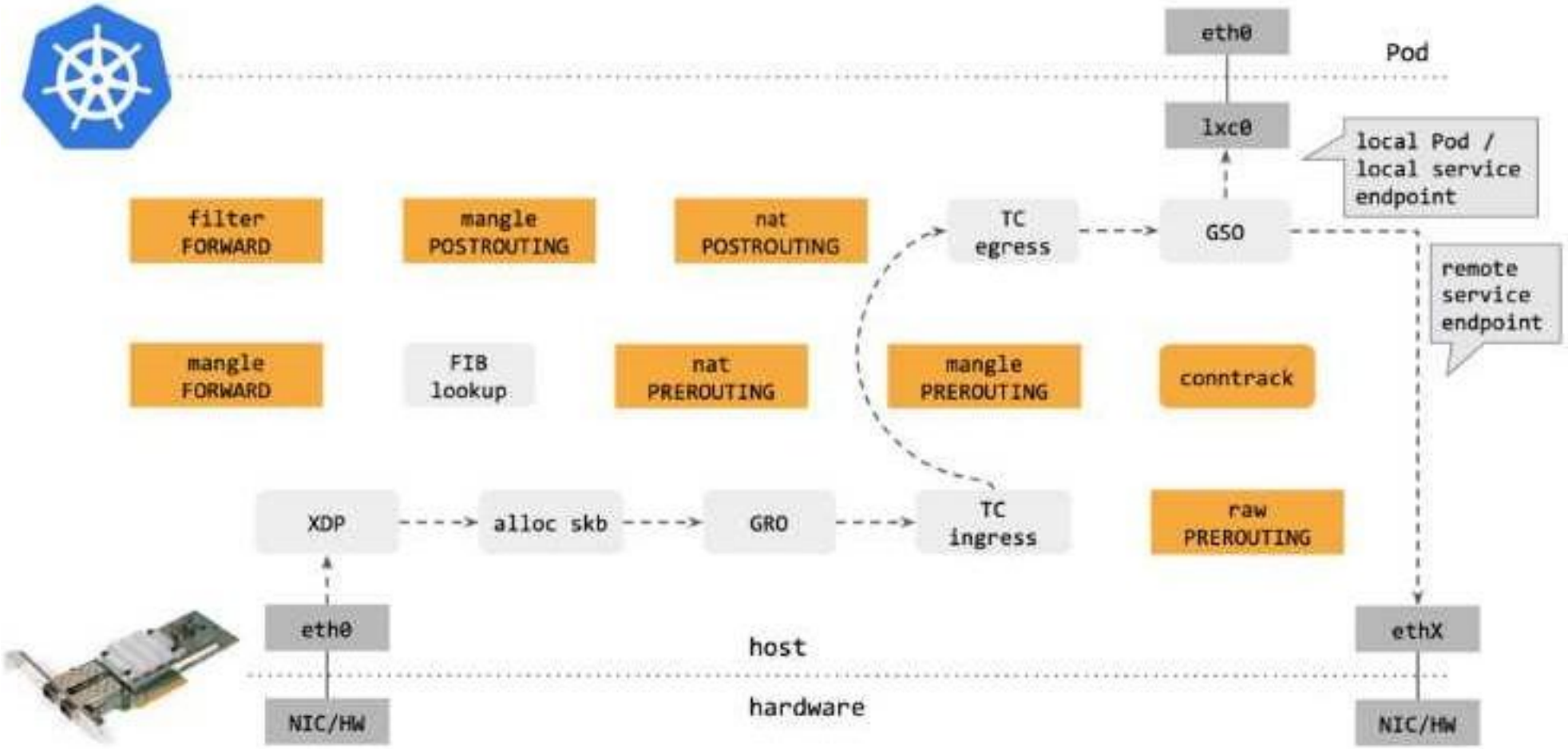
5. eBPF 的未来

5.2 eBPF 网络和安全的未来



5. eBPF 的未来

5.2 eBPF 网络和安全的未来



5. eBPF 的未来

5.3 eBPF 让观测更加智能和标准

无侵入式的服务依赖拓扑

5. eBPF 的未来

5.3 eBPF 让观测更加智能和标准

HTTP 监控

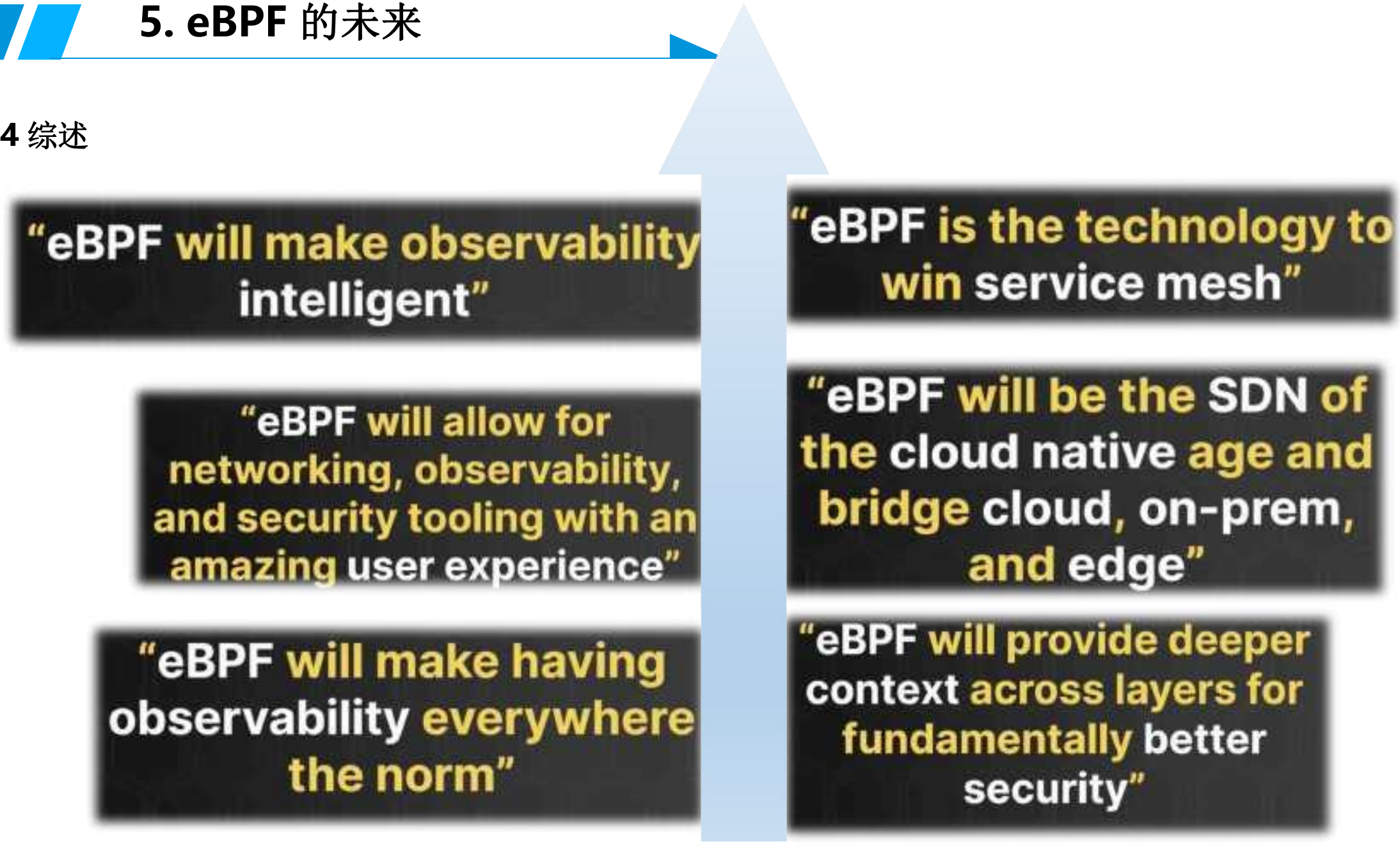
5. eBPF 的未来

5.3 eBPF 让观测更加智能和标准

DNS 监控

5. eBPF 的未来

5.4 综述



相关资料

网址：

官方网址：　　https://www.ebpf.io/

深入浅出 BPF：　https://www.ebpf.top/

图书：

快速入门，全面

聚焦观测和跟踪，性能分析各个维度

Q & A

深入浅出BFP 公众号

感谢聆听！！！