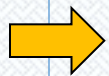


eBPF 与 Go , 超能力组合

-- 狄卫华



目录



1. eBPF 是什么？

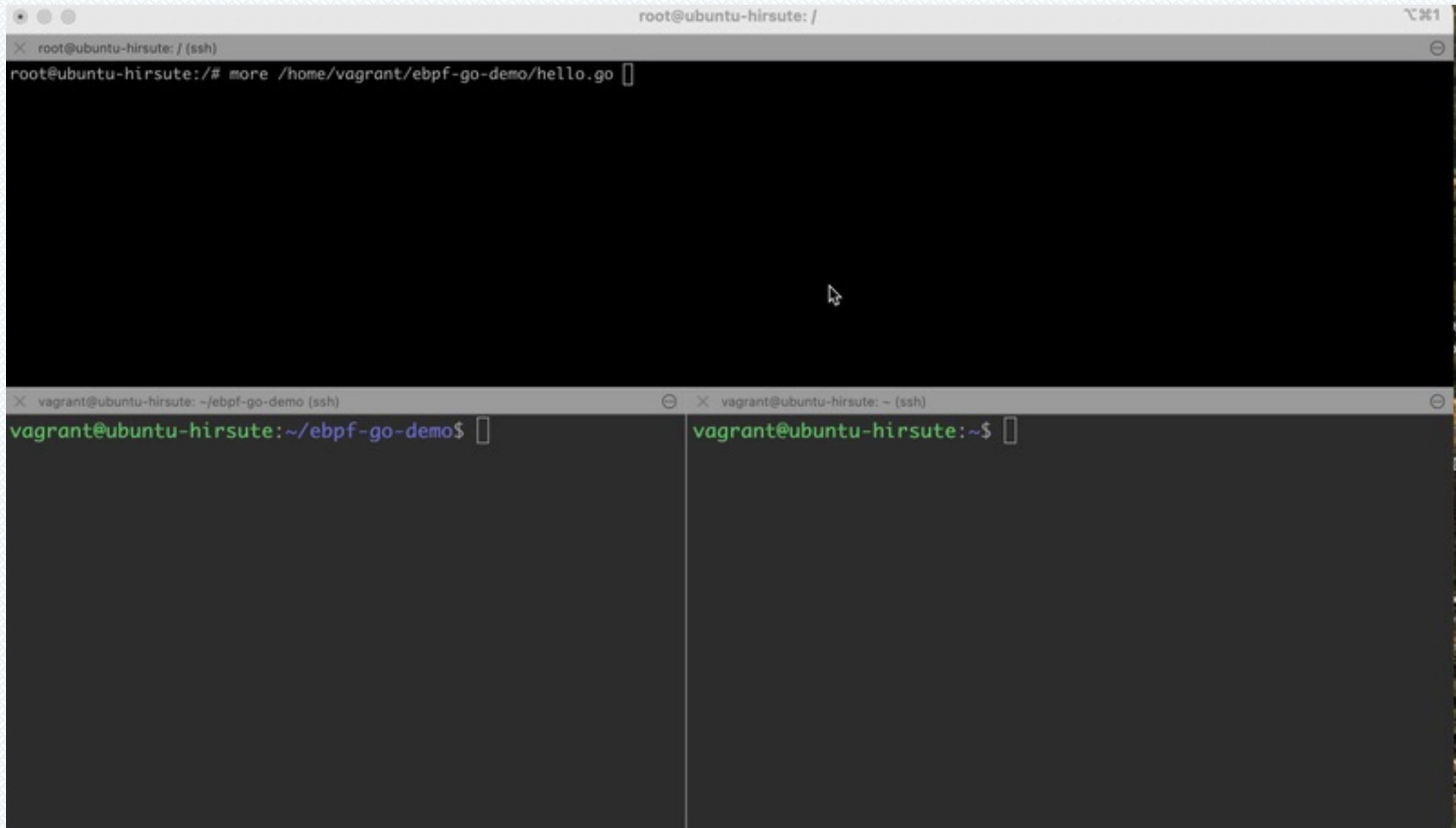
文本

2. eBPF 的应用场景有哪些？

3. eBPF 工作原理

4. eBPF 与 Go

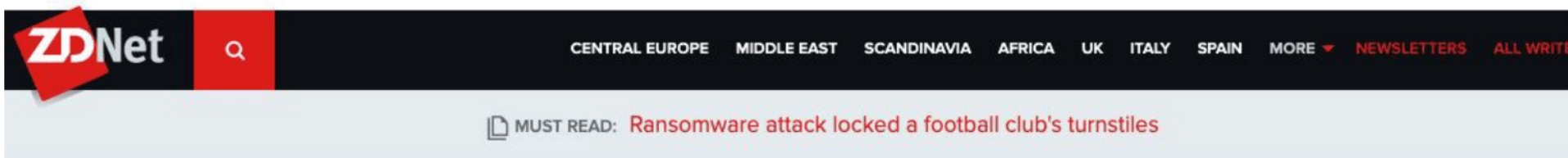
热身阶段 - 小的 demo



The screenshot displays a terminal window with three panes. The top pane shows a root user at the prompt, viewing the contents of a file named `hello.go` using the `more` command. The bottom-left pane shows a vagrant user at the prompt, having navigated to the directory `~/ebpf-go-demo`. The bottom-right pane shows the same vagrant user at the prompt, ready to execute a command. The terminal window has a title bar indicating the current session is an SSH connection to `root@ubuntu-hirsute: /`.

```
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 !!!



Netflix: BPF is a new type of software we use to run Linux apps securely in the kernel

A Netflix performance architect says BPF promises a fundamental change to a 50-year-old kernel model.

"BPF is the biggest operating systems change I've seen in my career, and it's thrilling to be a part of it," wrote Gregg.

1. eBPF 是什么？

1.1 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。



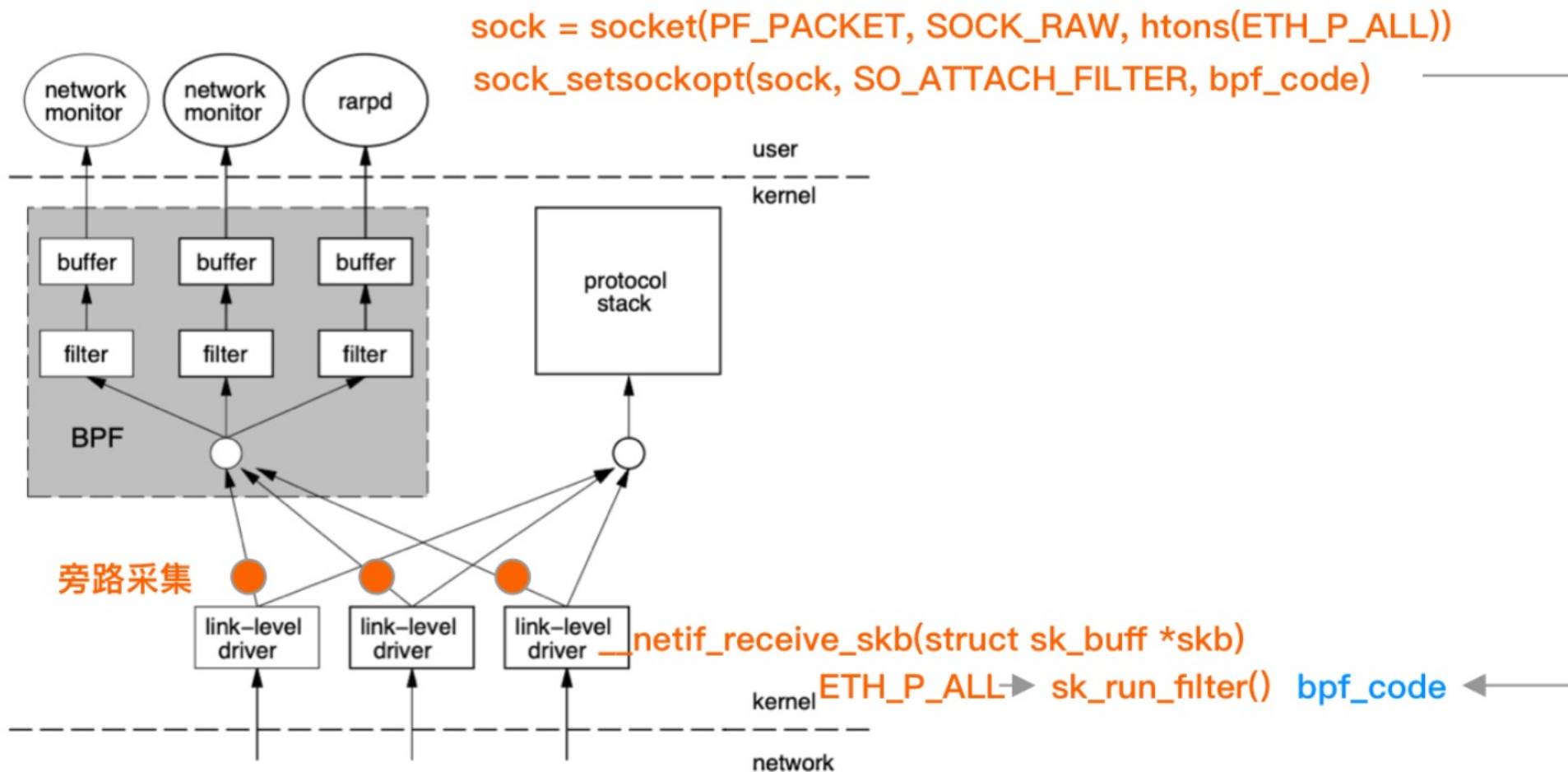
1. eBPF 是什么？

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'
```

```
(000) ldh    [12]
```

```
(001) jeq    #0x800      jt 2 jf 12
```

```
(002) ldb    [23]
```

```
(003) 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，应用场景不再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？

1.2 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？

1.3 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（114 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？

1.4 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
-

1. eBPF 是什么？

1.4 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？

1.5 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？

1.5 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？

1.5 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？

1.5 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？

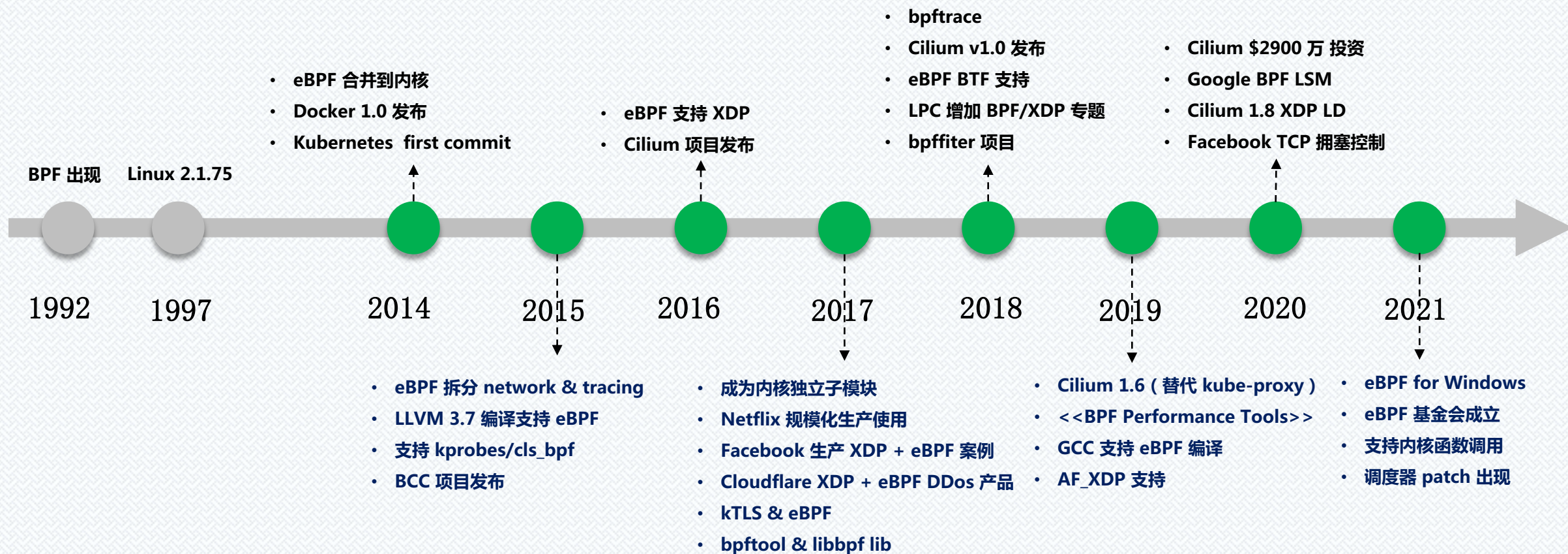
1.6 eBPF 限制

- 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？

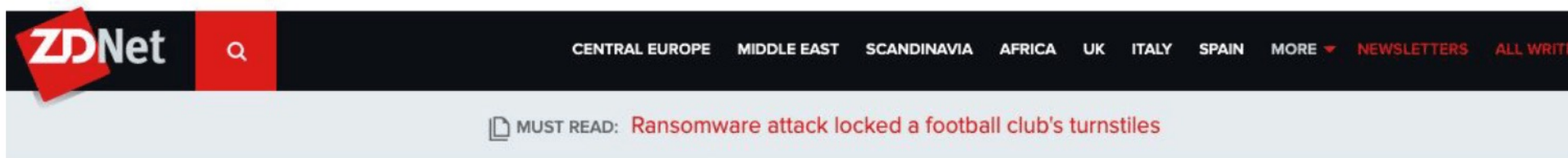
1.7 eBPF 年鉴



1. eBPF 是什么？

1.8 eBPF 总结

eBPF：过去 50 年操作系统最大的变革 !!!



Netflix: BPF is a new type of software we use to run Linux apps securely in the kernel

A Netflix performance architect says BPF promises a fundamental change to a 50-year-old kernel model.

"BPF is the biggest operating systems change I've seen in my career, and it's thrilling to be a part of it," wrote Gregg.

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:



TheRustyTwit
@rusty_twit

Replying to @LaF0rge

Well, iptables perf used to be "mostly good enough".
Replacing it has taken so long because it requires a
radically different approach; nice to see it finally
happening!

12:46 AM · Apr 18, 2018 · [Twitter for Android](#)

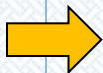
毫无疑问，这是内核里发展最快的子系统！



目录



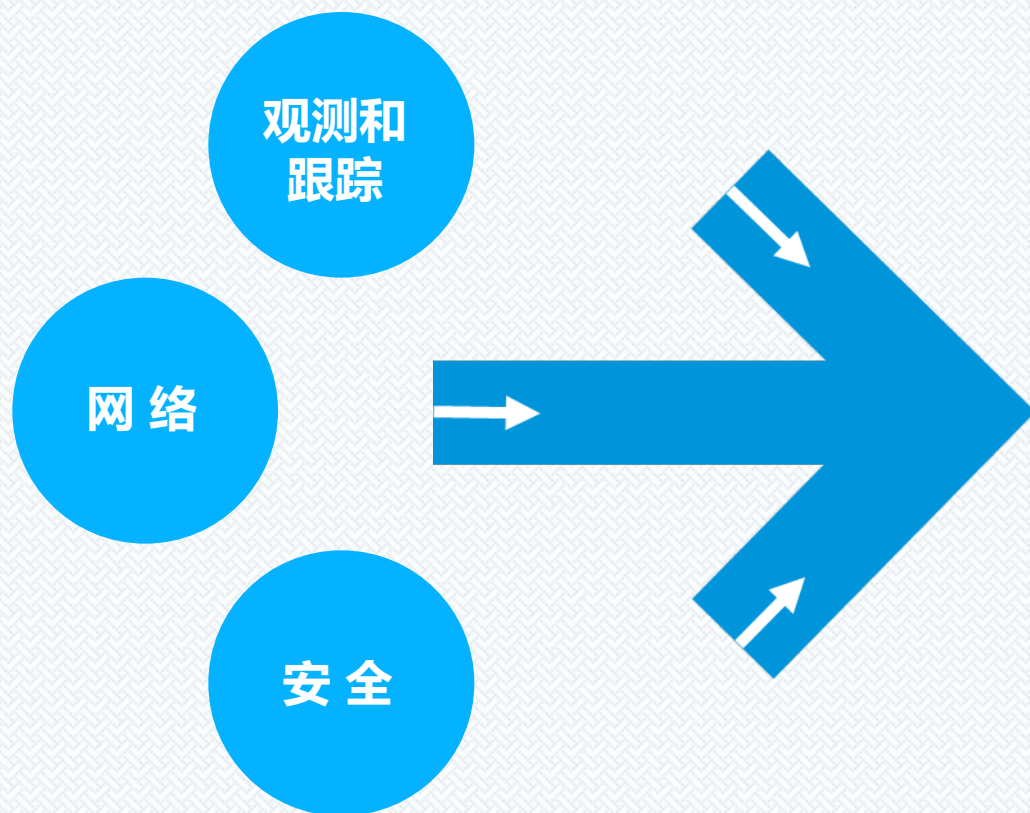
●	1. eBPF 是什么？
●	2. eBPF 的应用场景有哪些？
●	3. eBPF 工作原理
●	4. eBPF 与 Go



文本

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

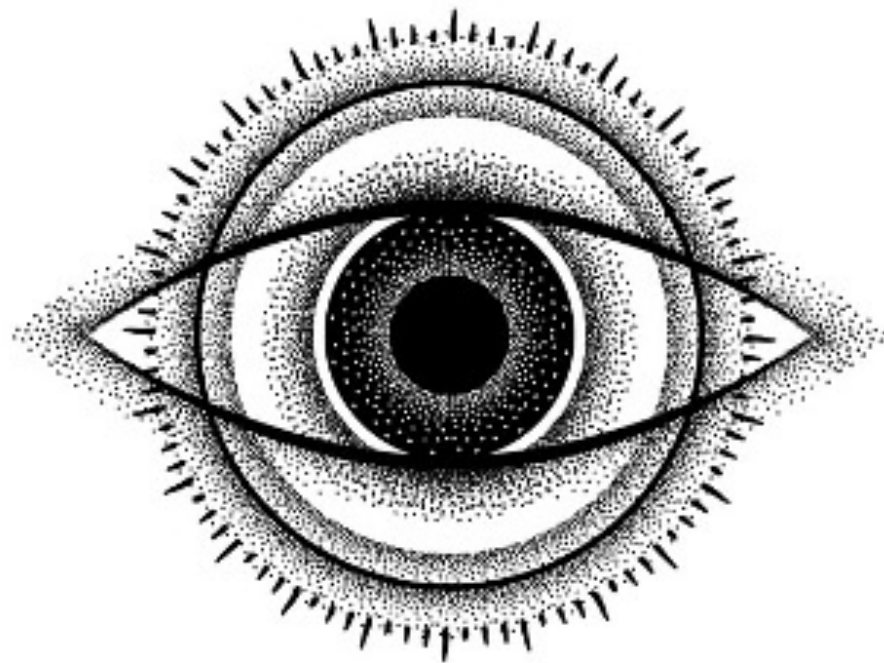
可编程性和效率的结合使得 eBPF 自然而然地满足了网络解决方案的所有数据包处理要求。eBPF 的可编程性使其能够在不离开 Linux 内核的包处理上下文的情况下，添加额外的协议解析器，并轻松编程任何转发逻辑以满足不断变化的需求。JIT 编译器提供的效率使其执行性能接近于本地编译的内核代码。

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以确保系统的安全。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

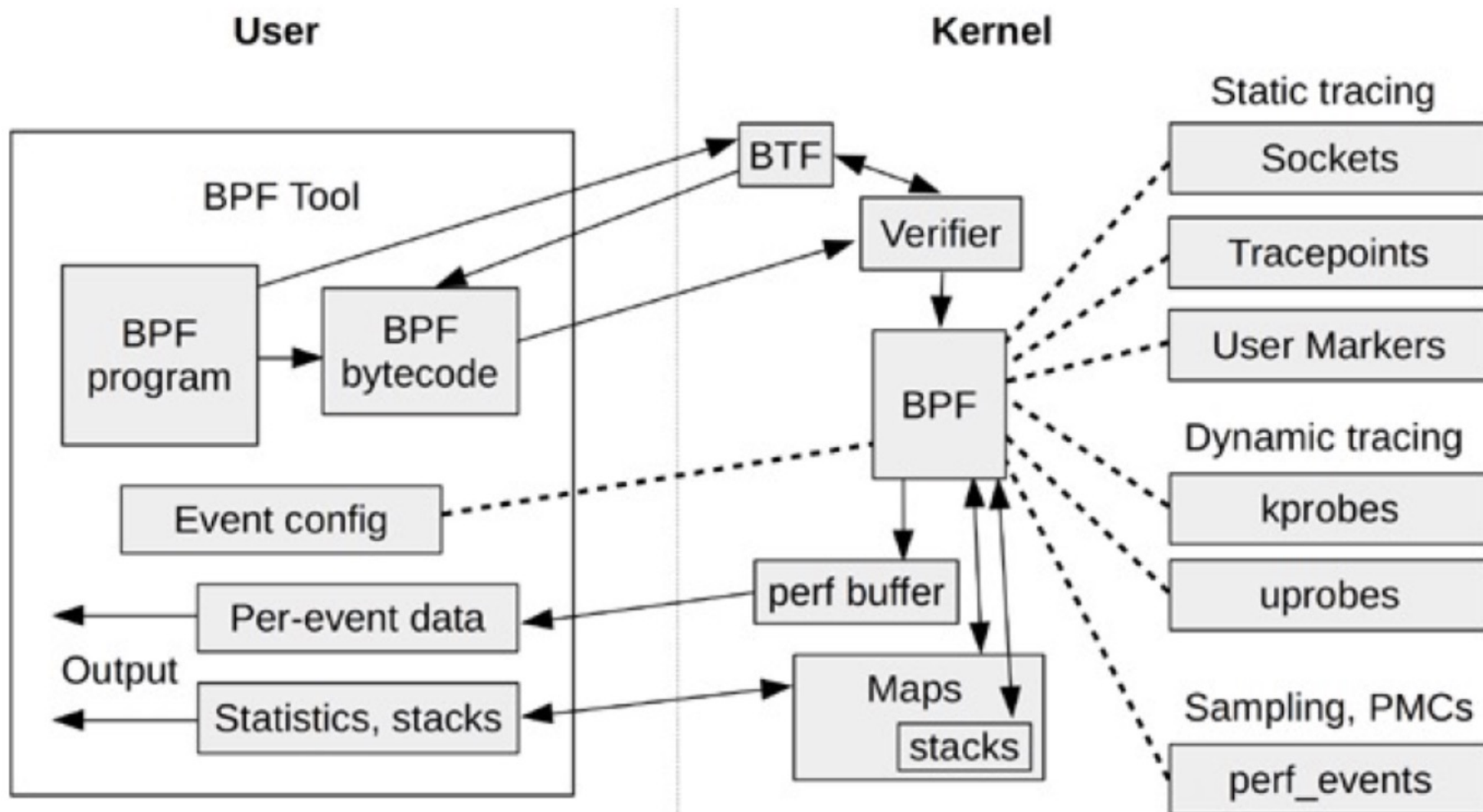


上帝之眼

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？

2.2 观测和跟踪

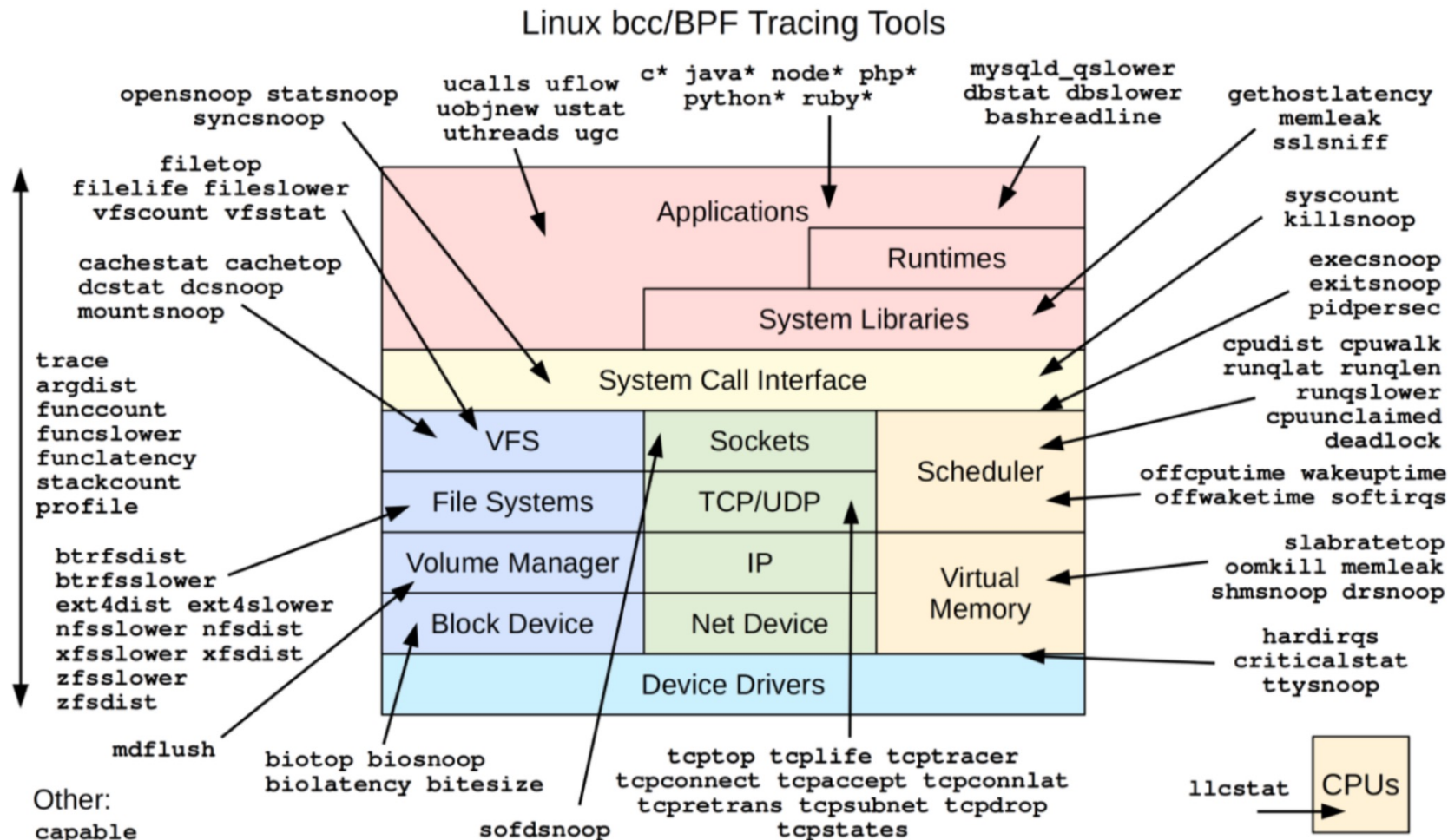
跟踪的事件对象支持：

- **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



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。

```
ceuse#
```

```
ceuse# █
```

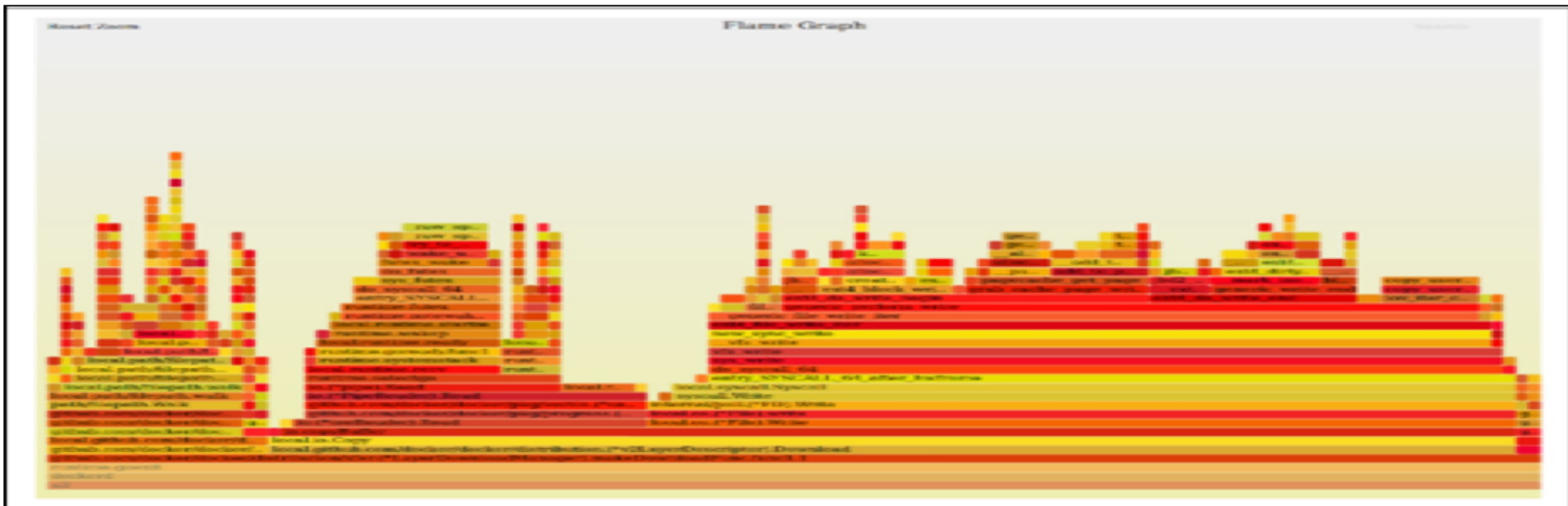
2. eBPF 的场景有哪些？

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

Get the return value

Using the binary of the main process of the container

Of the function defined as main.counterValue

And print the return value

```
1 kubectl trace run -e 'uretprobe:/proc/$container_pid/exe:"main.counterValue" { printf("%d %d\n",  
pid, retval) }' pod/caturday-566d99889-2wr5s -a -n caturday  
2 trace 338ee120-079b-11e9-a967-8c164500a77e created  
3 Attaching 1 probe...  
4 3160 1  
5 3160 2  
6 3160 3  
7 3160 4  
8 3160 5  
9 3160 6  
10 3160 7  
11 3160 8  
12 3160 9  
13 3160 10  
14 3160 11  
15 3160 12  
16 3160 13  
17 3160 14  
18 3160 15  
19 3160 16  
20 3160 17  
21 3160 18  
22 3160 19  
23 3160 20  
24 3160 21  
25 3160 22  
26 3160 23
```

```
1 func counterValue(counter prometheus.Counter) int {  
2     dm := &dto.Metric{}  
3     counter.Write(dm)  
4     return int(dm.Counter.GetValue())  
5 }
```

Yes, It's Caturday!

HINT: It's almost ALWAYS caturday.



Hostname caturday-566d99889-2wr5s
Count renders 23
Remote 192.168.1.63:54718

Linux kernel map

functionalities layers

user space interfaces

system calls and system files

human interface

HI char devices

cdev_add

input_fops console_fops video_fops snd_fops

sys_capset fb_fops sys_syslog

system

interfaces core

kernel/ System Call Interface system files

linux/syscalls.h linux/uaccess.h copy_from_user

/proc/sysfs/dev sysfs_ops

sysfs_ops

register_chrdev cdev_add cdev_map sys_epoll_create sys_reboot

sys_init_module

Device Model

drivers/base/ driver_init linux/kobject.h kobject bus_register

bus_type device_create class device_type device_driver driver_register probe kvm

load_module module module_param kernel_param

kobject_uevent_init kobject_uevent

processing

processes

kernel/ fs/exec.c

kernel/signal.c sys_fork sys_kill sys_signal do_sigaction

sys_futex

sys_nanosleep

threads

schedule_work wait_for_completion

INIT_WORK queue_work work_struct kthread_create kernel_thread current thread_info do_fork

synchronization

mutex lock_kernel kernel_mutex wait_for_completion timer_list run_timer_softirq

down_interruptible up semaphore spin_lock_irqsave spin_unlock_irqrestore msleep

memory

memory access

mm/ sys_brk sys_mmap sys_shm_ops sys_shmat

si_meminfo

/dev/mem mem_fops mmap_ops

vmalloc_init vmalloc_vfree vm_struct virt_to_page

virtual memory

vmalloc_init vmalloc_vfree vm_struct virt_to_page

storage

files & directories access

fs/ sys_open sys_write sys_read sys_poll sys_select sys_flock sys_splice sys_sendfile

fs/ sys_statfs sys_statfs sys_statfs sys_statfs

fs/ sys_statfs sys_statfs sys_statfs sys_statfs

Virtual File System

file_operations file_operations file_operations file_operations

file_operations file_operations file_operations file_operations

networking

sockets access

net/ sys_socketcall sys_connect sys_accept sys_bind sys_listen sys_sendmsg sys_recvmsg sys_setsockopt

/proc/net/ tcp_seq_show tcp_seq_show tcp_seq_show

socket_ioctl

protocol families

inet_init sock_create inet_family_ops inet_create inet_dgram_ops inet_stream_ops

networking storage

nfs_file_operations smb_fs_type cifs_file_ops lcacl_tcp_transport

socket splice

sock_sendpage tcp_sendpage tcp_sendpage

logical

functions implementations

ois also

device control

abstract devices and HID class drivers

drivers/ input/ drivers/media/ sound/ console kbd fb_fops dm_fops mousedev

hardware interfaces

HI peripherals device drivers

uvc_driver ac97_driver vga_con atkbd_driver i8042_driver psmouse

device access and bus drivers

request_region pci_driver request_mem_region pci_request_driver pci_request_regions usb_driver usb_submit_urb usb_hcd usb_submit_urb ioremap writeback ehci_irq ehci_urb_enqueue usb_hcd_irq pci_read pci_write

CPU specific

arch/x86/ setup_arch x86_init trap_init native_irq

/proc/interrupts system_call show_regs pt_regs atomic_t cli sti

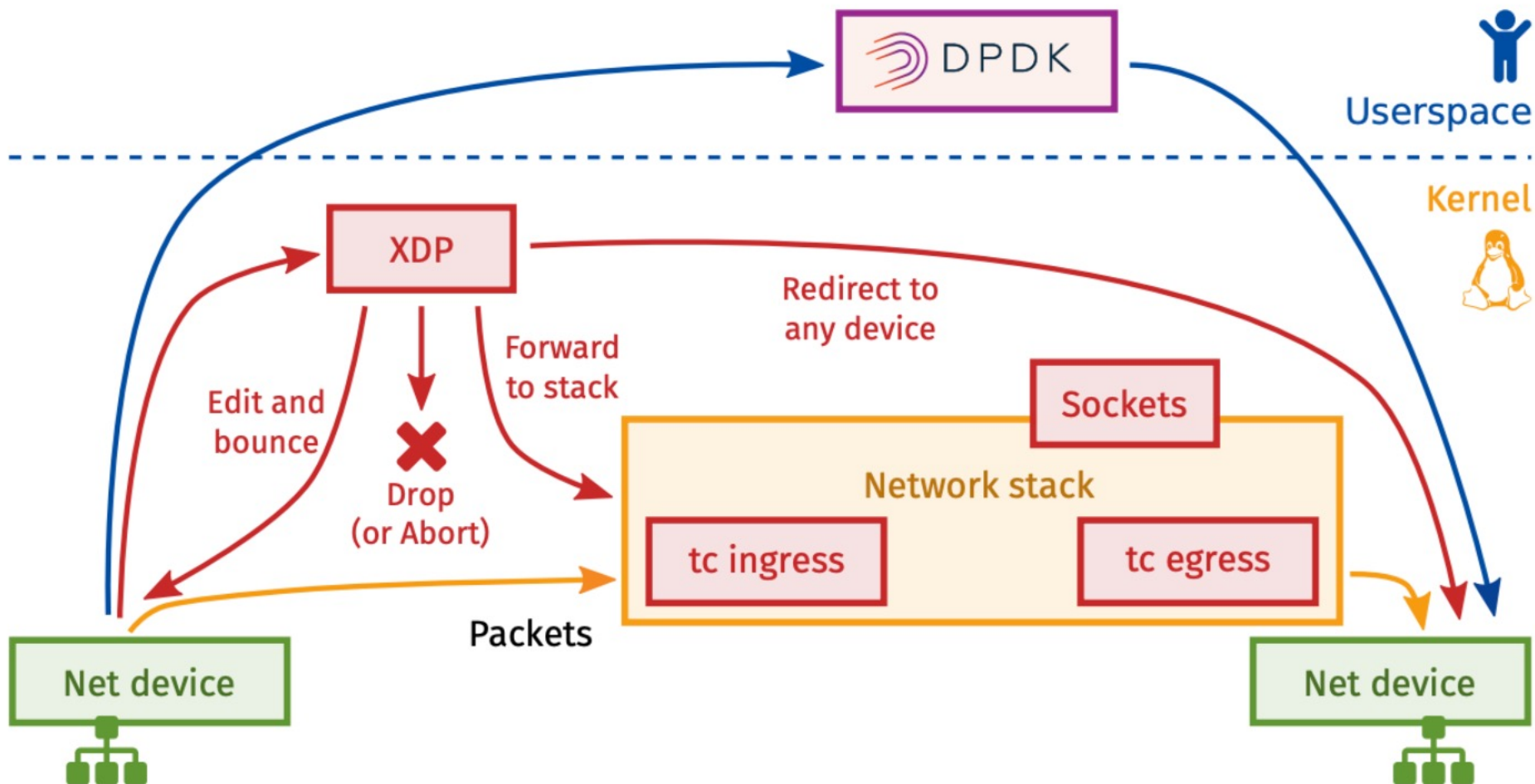
memory

physical memory operations

arch/x86/ mm/ get_page_from_freelist zone1st zone2nd zone3rd zone4th zone5th zone6th zone7th zone8th zone9th zone10th zone11th zone12th zone13th zone14th zone15th zone16th zone17th zone18th zone19th zone20th zone21th zone22th zone23th zone24th zone25th zone26th zone27th zone28th zone29th zone30th zone31th zone32th zone33th zone34th zone35th zone36th zone37th zone38th zone39th zone40th zone41th zone42th zone43th zone44th zone45th zone46th zone47th zone48th zone49th zone50th zone51th zone52th zone53th zone54th zone55th zone56th zone57th zone58th zone59th zone60th zone61th zone62th zone63th zone64th zone65th zone66th zone67th zone68th zone69th zone70th zone71th zone72th zone73th zone74th zone75th zone76th zone77th zone78th zone79th zone80th zone81th zone82th zone83th zone84th zone85th zone86th zone87th zone88th zone89th zone90th zone91th zone92th zone93th zone94th zone95th zone96th zone97th zone98th zone99th zone100th zone101th zone102th zone103th zone104th zone105th zone106th zone107th zone108th zone109th zone110th zone111th zone112th zone113th zone114th zone115th zone116th zone117th zone118th zone119th zone120th zone121th zone122th zone123th zone124th zone125th zone126th zone127th zone128th zone129th zone130th zone131th zone132th zone133th zone134th zone135th zone136th zone137th zone138th zone139th zone140th zone141th zone142th zone143th zone144th zone145th zone146th zone147th zone148th zone149th zone150th zone151th zone152th zone153th zone154th zone155th zone156th zone157th zone158th zone159th zone160th zone161th zone162th zone163th zone164th zone165th zone166th zone167th zone168th zone169th zone170th zone171th zone172th zone173th zone174th zone175th zone176th zone177th zone178th zone179th zone180th zone181th zone182th zone183th zone184th zone185th zone186th zone187th zone188th zone189th zone190th zone191th zone192th zone193th zone194th zone195th zone196th zone197th zone198th zone199th zone200th zone201th zone202th zone203th zone204th zone205th zone206th zone207th zone208th zone209th zone210th zone211th zone212th zone213th zone214th zone215th zone216th zone217th zone218th zone219th zone220th zone221th zone222th zone223th zone224th zone225th zone226th zone227th zone228th zone229th zone230th zone231th zone232th zone233th zone234th zone235th zone236th zone237th zone238th zone239th zone240th zone241th zone242th zone243th zone244th zone245th zone246th zone247th zone248th zone249th zone250th zone251th zone252th zone253th zone254th zone255th zone256th zone257th zone258th zone259th zone260th zone261th zone262th zone263th zone264th zone265th zone266th zone267th zone268th zone269th zone270th zone271th zone272th zone273th zone274th zone275th zone276th zone277th zone278th zone279th zone280th zone281th zone282th zone283th zone284th zone285th zone286th zone287th zone288th zone289th zone290th zone291th zone292th zone293th zone294th zone295th zone296th zone297th zone298th zone299th zone300th zone301th zone302th zone303th zone304th zone305th zone306th zone307th zone308th zone309th zone310th zone311th zone312th zone313th zone314th zone315th zone316th zone317th zone318th zone319th zone320th zone321th zone322th zone323th zone324th zone325th zone326th zone327th zone328th zone329th zone330th zone331th zone332th zone333th zone334th zone335th zone336th zone337th zone338th zone339th zone340th zone341th zone342th zone343th zone344th zone345th zone346th zone347th zone348th zone349th zone350th zone351th zone352th zone353th zone354th zone355th zone356th zone357th zone358th zone359th zone360th zone361th zone362th zone363th zone364th zone365th zone366th zone367th zone368th zone369th zone370th zone371th zone372th zone373th zone374th zone375th zone376th zone377th zone378th zone379th zone380th zone381th zone382th zone383th zone384th zone385th zone386th zone387th zone388th zone389th zone390th zone391th zone392th zone393th zone394th zone395th zone396th zone397th zone398th zone399th zone400th zone401th zone402th zone403th zone404th zone405th zone406th zone407th zone408th zone409th zone410th zone411th zone412th zone413th zone414th zone415th zone416th zone417th zone418th zone419th zone420th zone421th zone422th zone423th zone424th zone425th zone426th zone427th zone428th zone429th zone430th zone431th zone432th zone433th zone434th zone435th zone436th zone437th zone438th zone439th zone440th zone441th zone442th zone443th zone444th zone445th zone446th zone447th zone448th zone449th zone450th zone451th zone452th zone453th zone454th zone455th zone456th zone457th zone458th zone459th zone460th zone461th zone462th zone463th zone464th zone465th zone466th zone467th zone468th zone469th zone470th zone471th zone472th zone473th zone474th zone475th zone476th zone477th zone478th zone479th zone480th zone481th zone482th zone483th zone484th zone485th zone486th zone487th zone488th zone489th zone490th zone491th zone492th zone493th zone494th zone495th zone496th zone497th zone498th zone499th zone500th zone501th zone502th zone503th zone504th zone505th zone506th zone507th zone508th zone509th zone510th zone511th zone512th zone513th zone514th zone515th zone516th zone517th zone518th zone519th zone520th zone521th zone522th zone523th zone524th zone525th zone526th zone527th zone528th zone529th zone530th zone531th zone532th zone533th zone534th zone535th zone536th zone537th zone538th zone539th zone540th zone541th zone542th zone543th zone544th zone545th zone546th zone547th zone548th zone549th zone550th zone551th zone552th zone553th zone554th zone555th zone556th zone557th zone558th zone559th zone560th zone561th zone562

2. eBPF 的场景有哪些？

2.3 网络





2. eBPF 的场景有哪些？

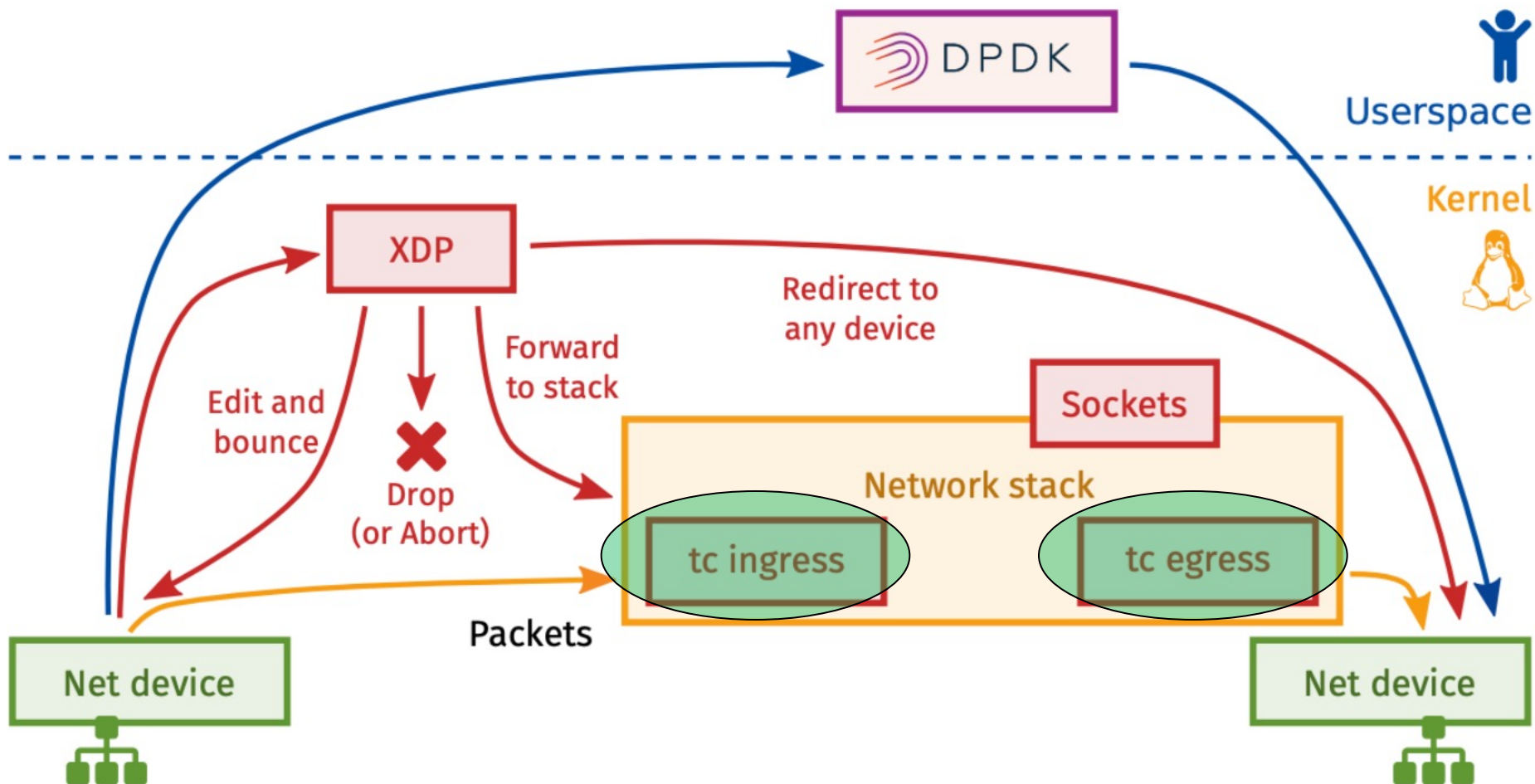


场景1：网络数据包过滤

BPF_PROG_TYPE_SOCKET_FILTER 类型，与 cBPF 早期的 tcpdump 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

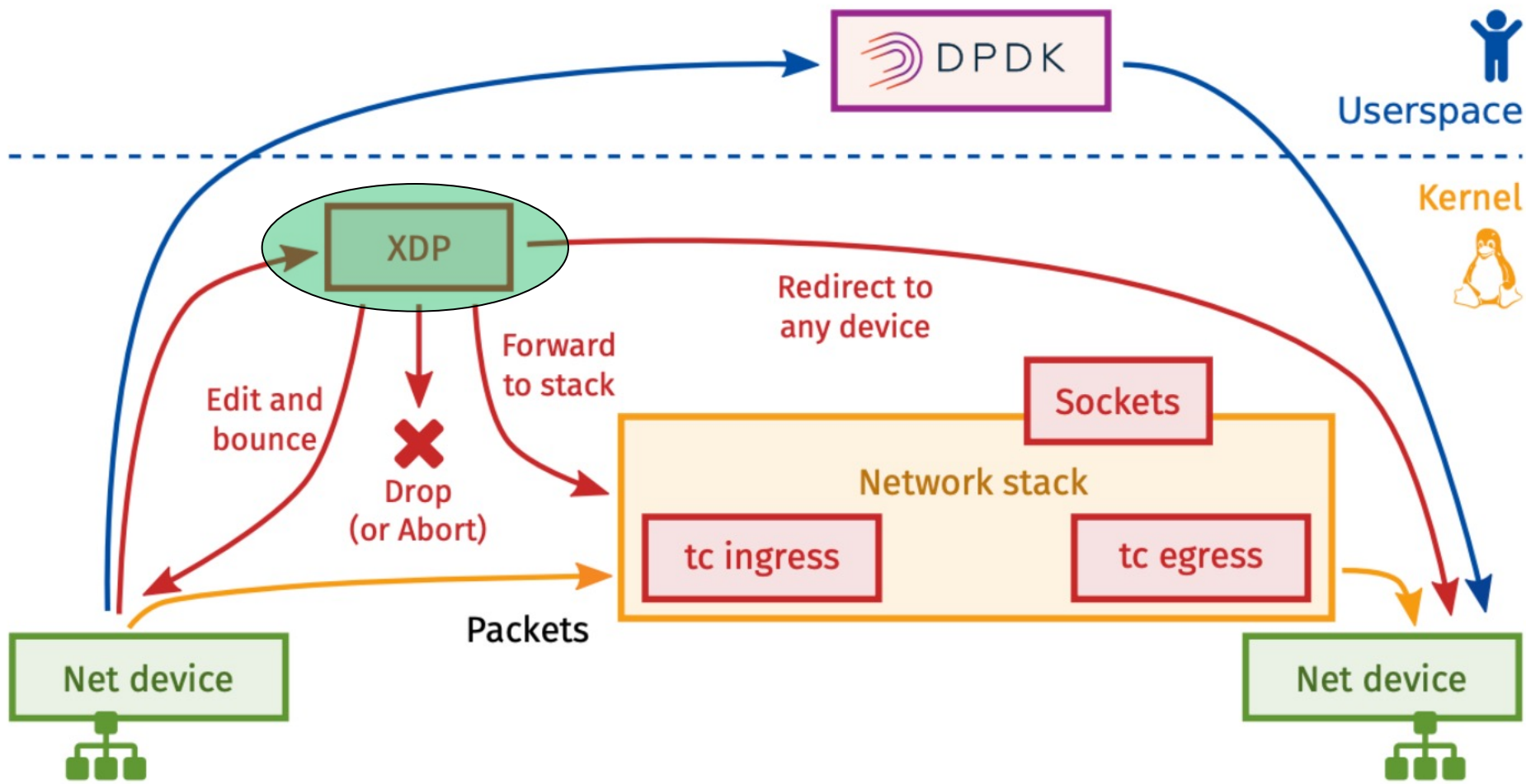
流量控制是一种强大的机制，在所有分类器中，可以编程网络数据路径 cls-bpf 分类器。cls-bpf 可以将 BPF 程序直接挂钩到入口和出口层，从而实现对于数据包流入和流出控制。

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；
- TC_ACT_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？

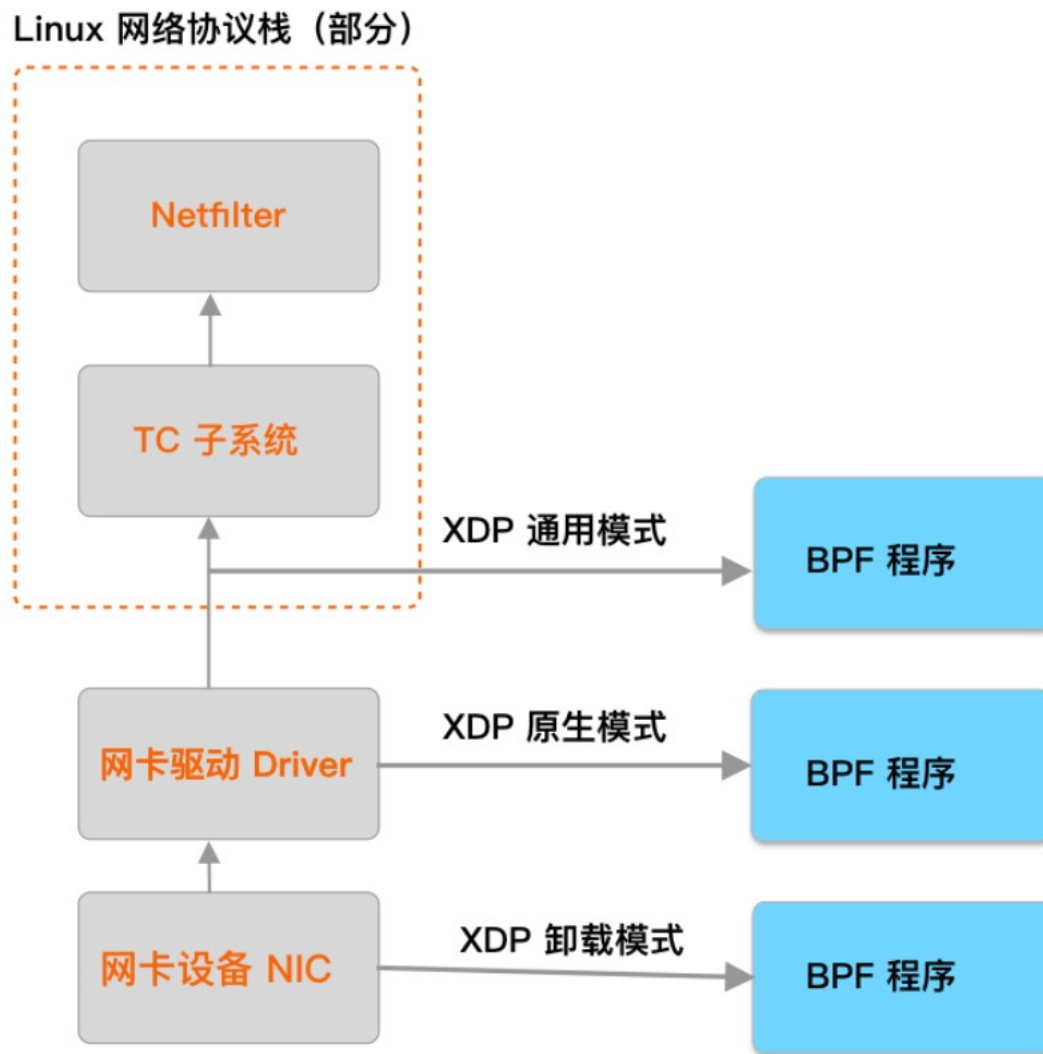
场景3 - XDP

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：

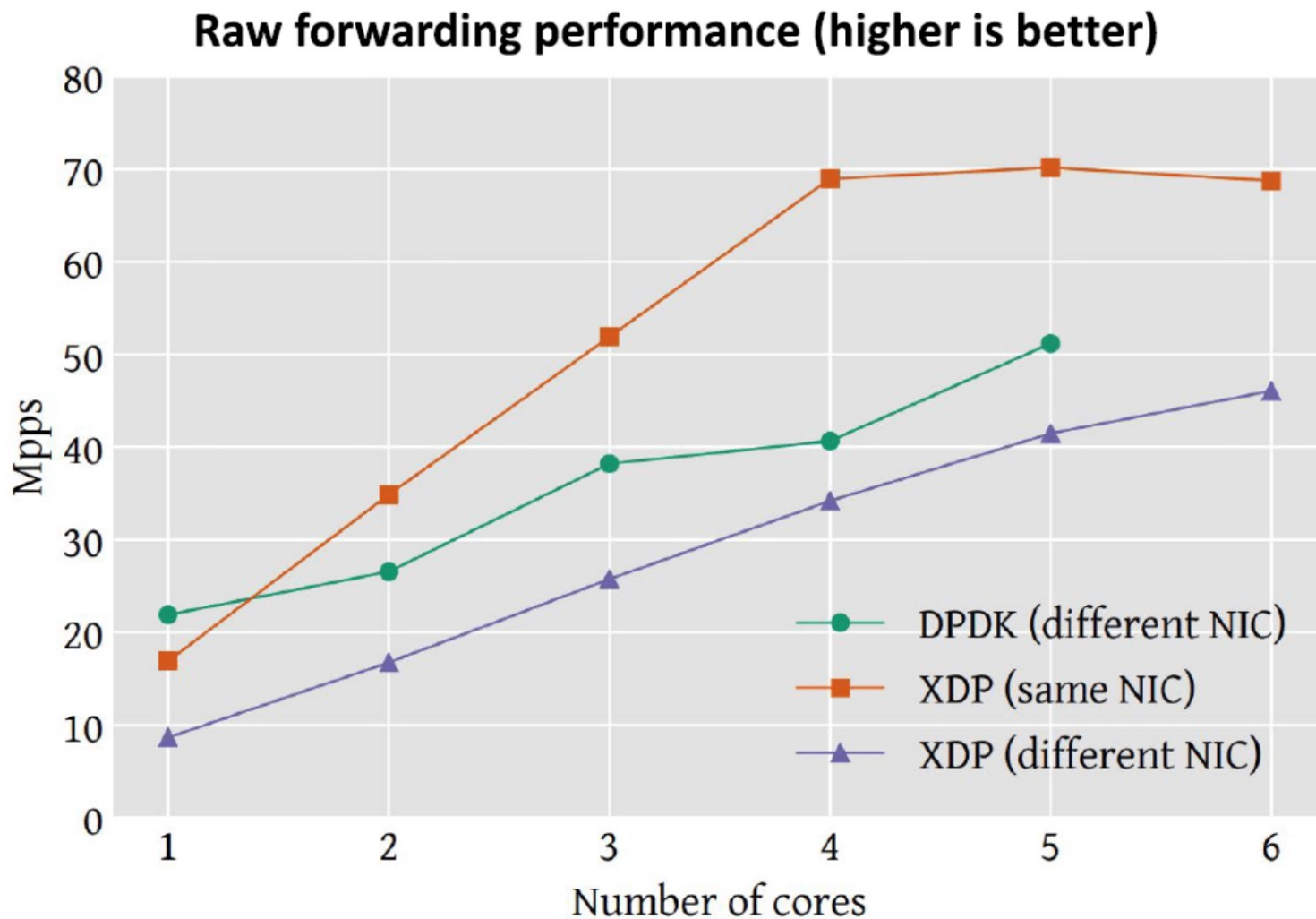
- **XDP_DROP** 丢弃数据包，这通常发生在驱动程序最早的 RX 阶段，对于降低 Dos 场景而言，尽早丢弃包是关键，这可以尽量少的占用 CPU 的处理时间；
- **XDP_TX**：转发数据包，可能发生在数据包被修改前或修改后，将处理后的数据包发回给相同的网卡；
- **XDP_PASS**：将数据包传递到普通网络协议栈处理，与没有 XDP BPF 程序运行的效果一致；
- **XDP_REDIRECT**：与 XDP_TX 类似，只是转发的目的地可以为其他 CPU 处理队列、不同的网卡或转发到特定的用户空间 socket (AF_XDP)，具体依赖于 Redirect Map 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

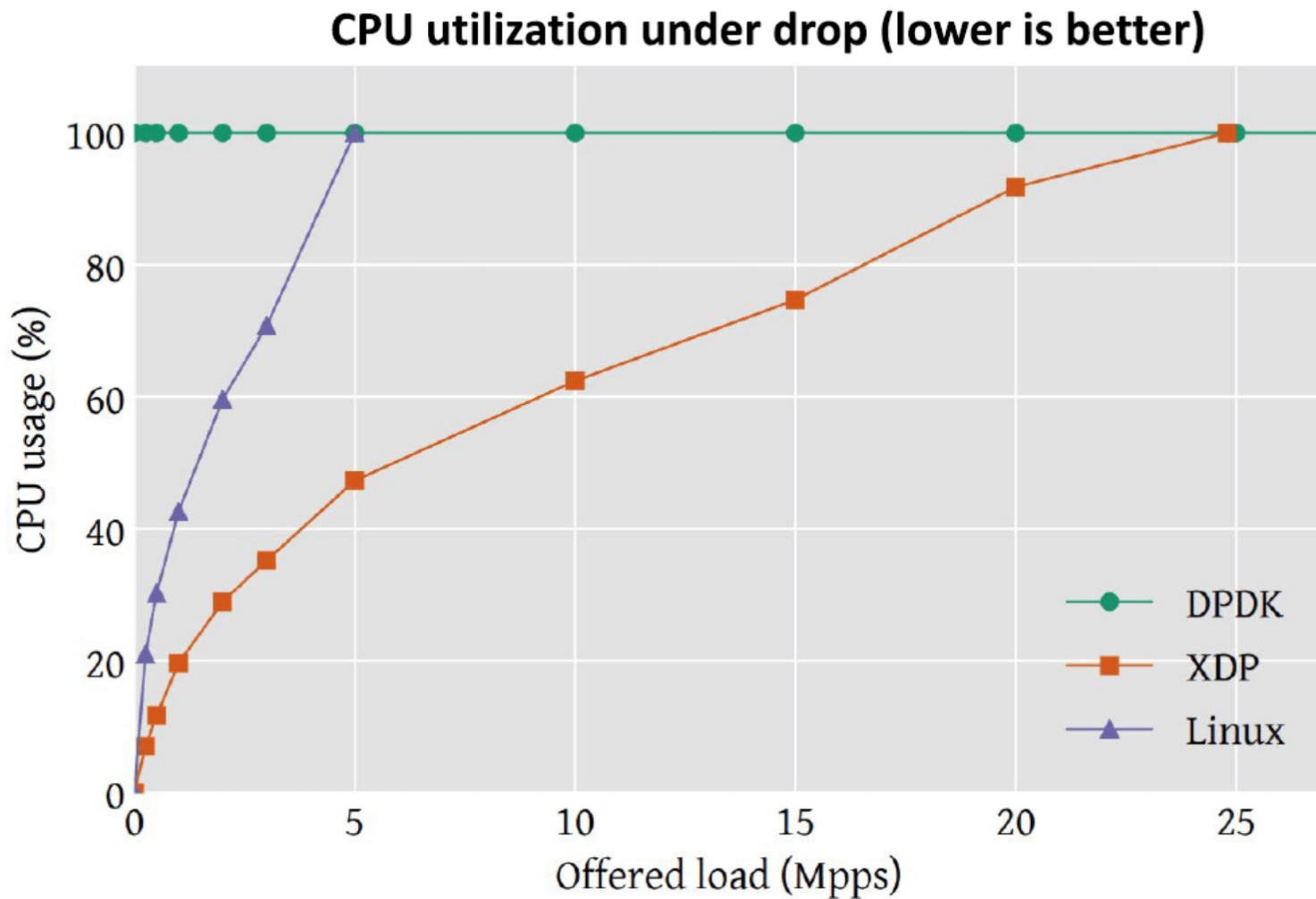
DPDK 与 XDP 转发吞吐量对比



2. eBPF 的场景有哪些？

场景3 - XDP

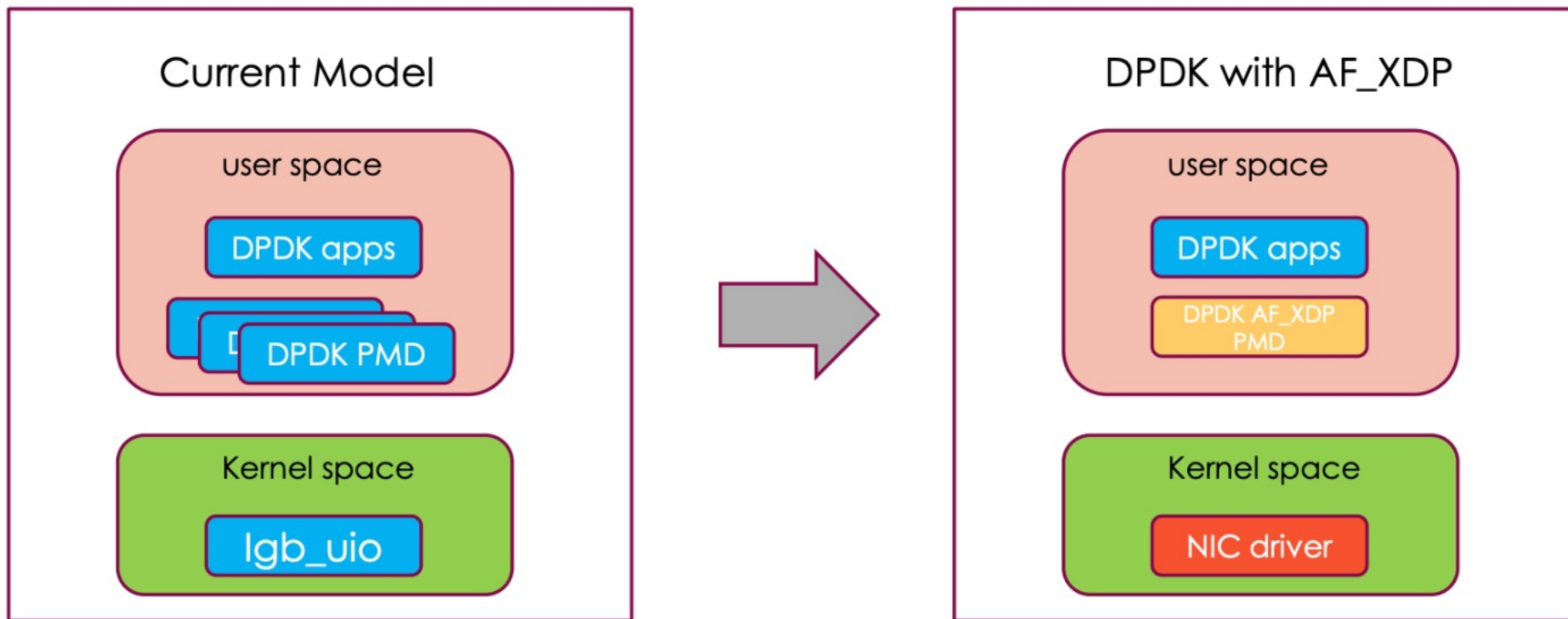
DPDK 与 XDP 转发 CPU 使用对比



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



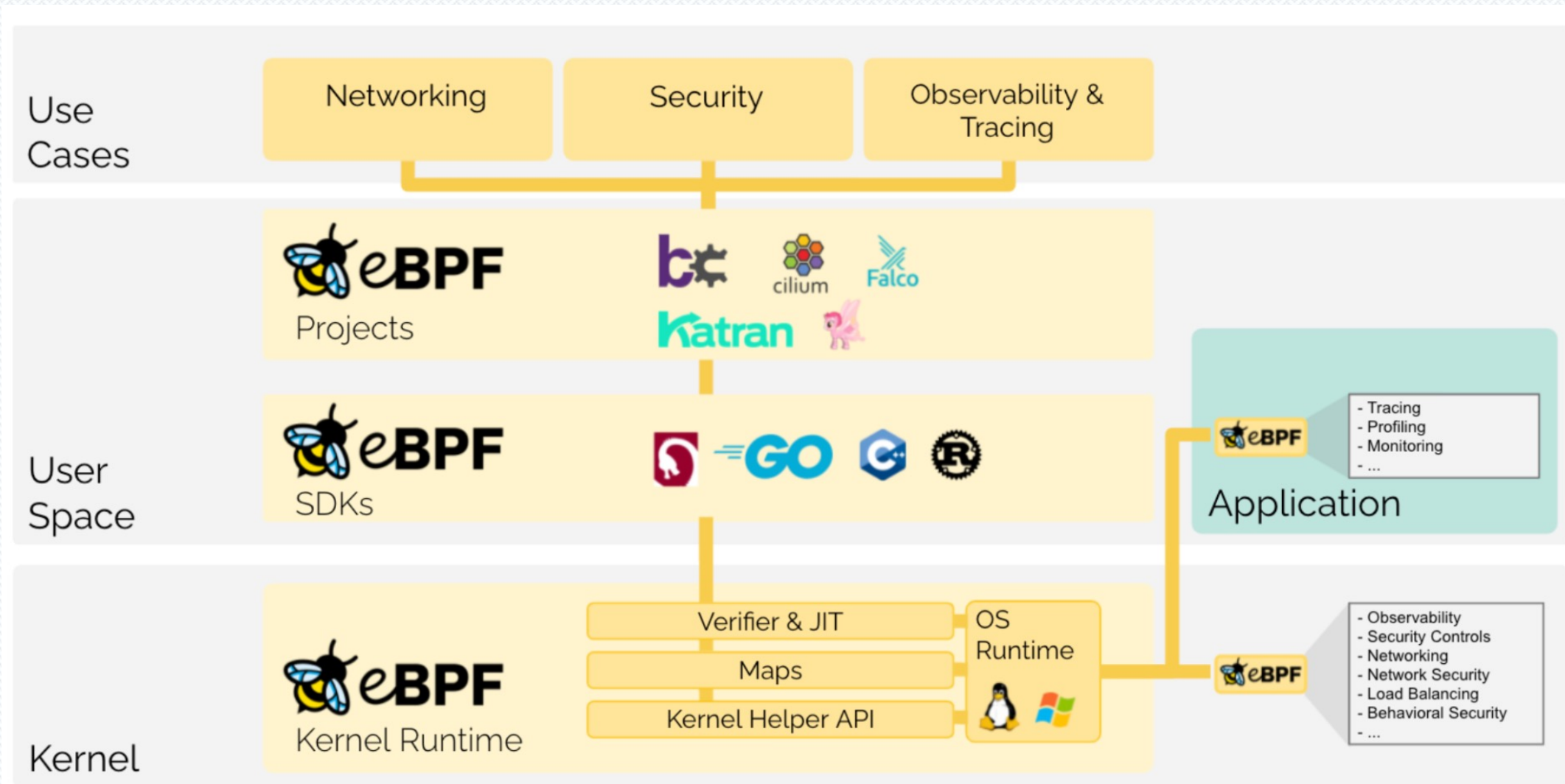
- 1. eBPF 是什么 ?
- 2. eBPF 的应用场景有哪些 ?
- ➔ 3. eBPF 工作原理
- 4. eBPF 与 Go

文本

3. eBPF 工作原理

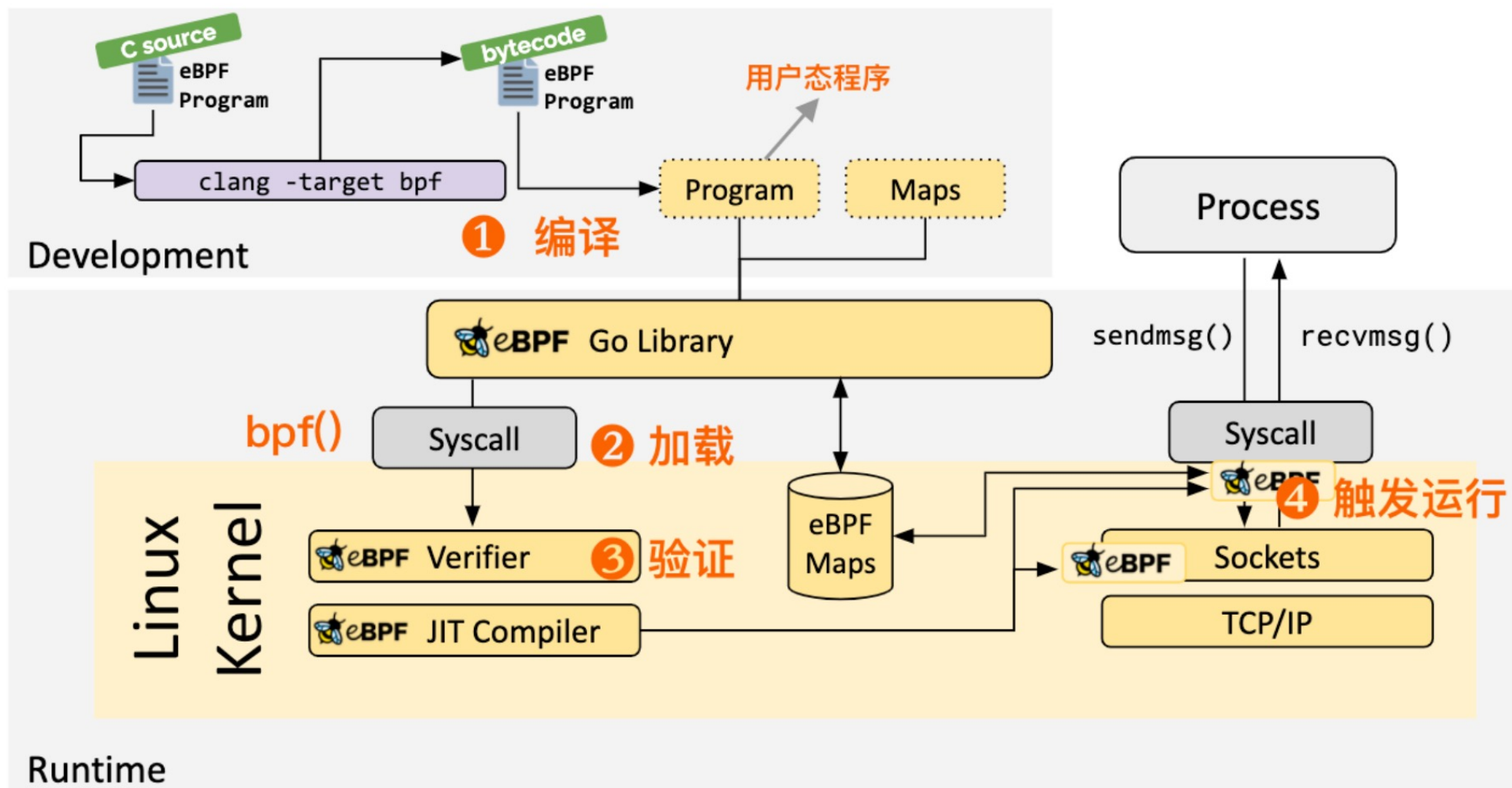
理

3.1 eBPF 全景图



3. eBPF 工作原理

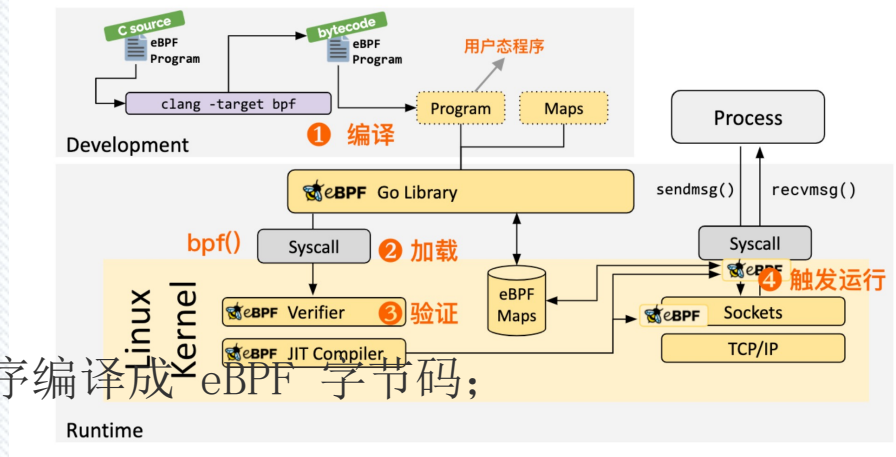
3.2 eBPF 工作原理



3. eBPF 工作原理

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（Verifier） 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 工作原理

3.2 eBPF 程序工作原理

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。



目录

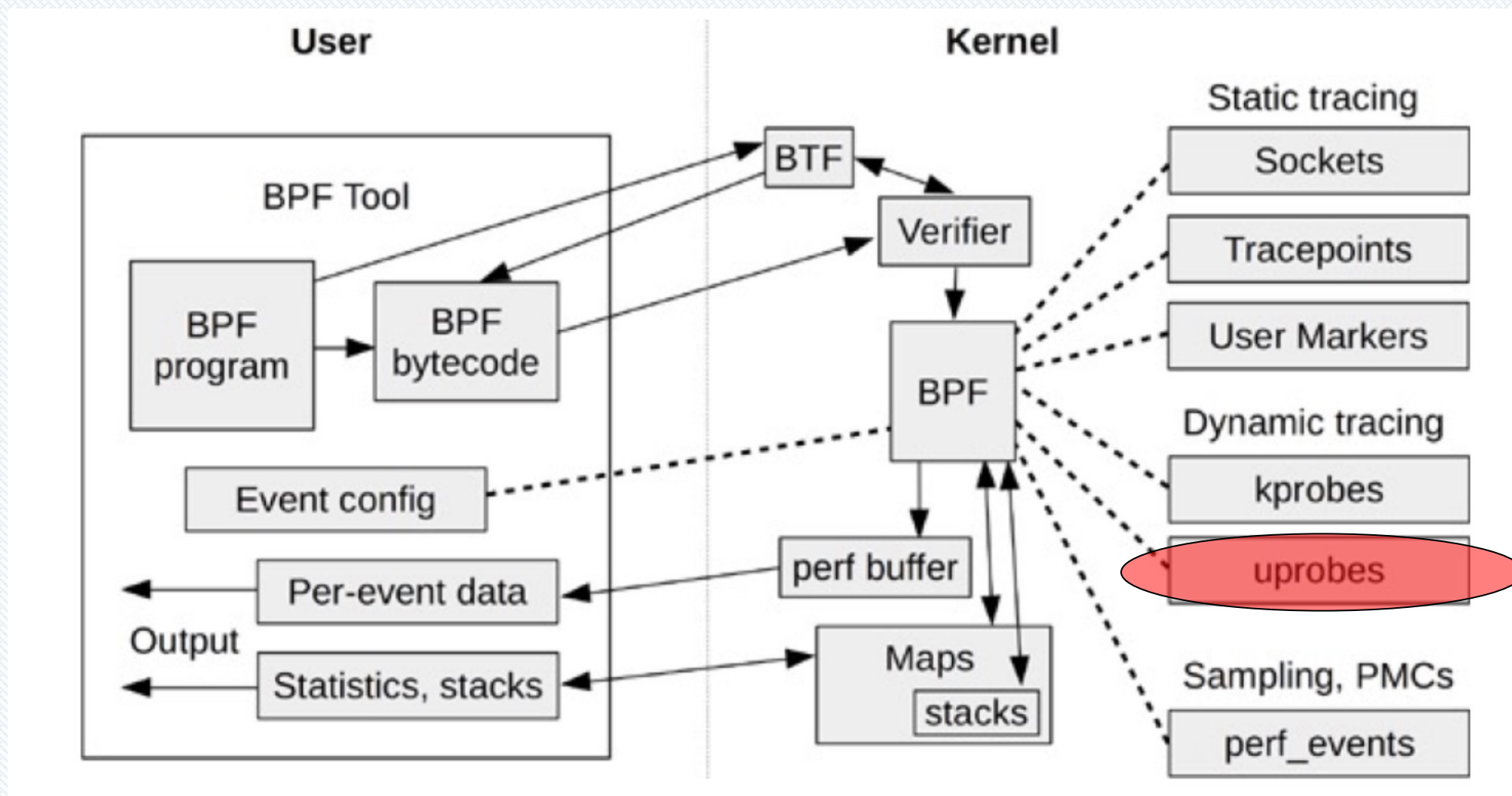


- 1. eBPF 是什么 ?
- 2. eBPF 的应用场景有哪些 ?
- 3. eBPF 工作原理
- 4. eBPF 与 Go

文本

4. eBPF 与 Go

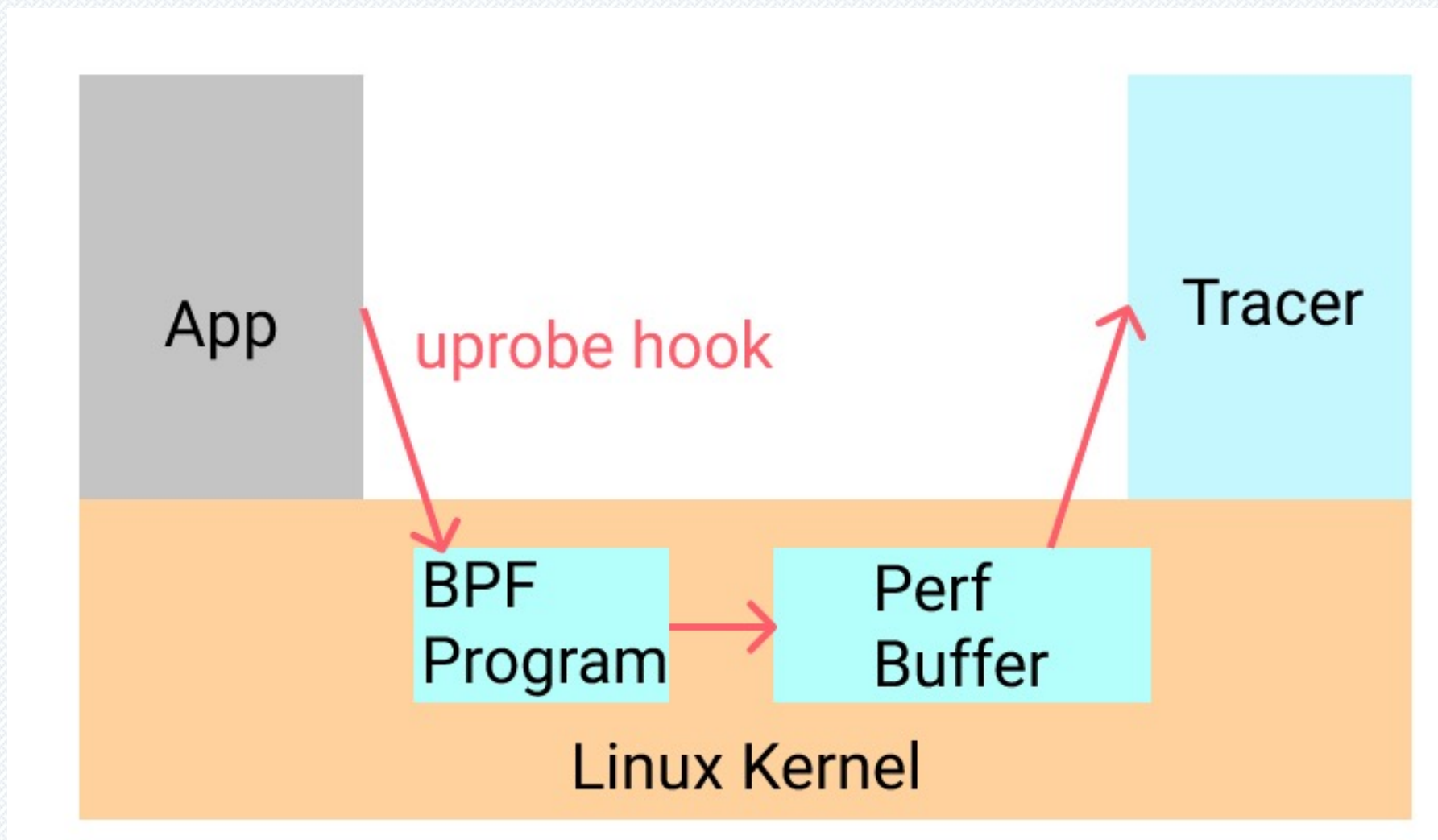
4.1 使用 eBPF 跟踪 Go 程序





4. eBPF 与 Go

4.1 使用 eBPF 跟踪 Go 程序





4. eBPF 与 Go



4.1 使用 eBPF 跟踪 Go 程序

```
$ sudo bpftrace -l 'uprobe:./tracee:main*'
uprobe:./tracee:main.ebpfDemo
uprobe:./tracee:main.main
```

```
$ sudo bpftrace -e 'uprobe:tracee/tracee:main.ebpfDemo
{printf("arg: %d\n", sarg0);}'
Attaching 1 probe...
arg: 100
```



4. eBPF 与 Go



4.1 使用 eBPF 跟踪 Go 程序

```
sudo bpftrace -e
'uprobe:/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee/tracee:main.ebpfDemo
{
    printf("arg: %s\n", ustack(perf));
}'
```

Attaching 1 probe...

arg:

```
49a620 main.ebpfDemo+0 (/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee/tracee)
434909 runtime.main+521 (/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee/tracee)
462fc1 runtime.goexit+1 (/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee/tracee)
```

 $\wedge C$

@ns[tracee]:

[64K, 128K)

[128K, 256K)

[256K, 512K)



4. eBPF 与 Go



4.1 使用 eBPF 跟踪 Go 程序

```
uprobe:tracee/tracee:main.ebpfDemo
{
    @start[pid] = nsecs;
}
```

```
uretprobe:tracee/tracee:main.ebpfDemo
/@start[pid]/
{
    @ns[comm] = hist(nsecs - @start[pid]);
    delete(@start[pid]);
}
```



4. eBPF 与 Go



4.1 使用 eBPF 跟踪 Go 程序

- Ubuntu 20.04 LTS/amd64
- bcc v0.23.0（源码安装）
- iovisor/gobpf v0.2.0
- Go1.15 linux/amd64（最新 1.17 版本调用方式可能有变化）

4.2 使用 eBPF 修改 Go 程序参数

局限性

- eBPF 在分析 Go 语言 ABI 的时候局限于简单类型，更加复杂的类型（用户自定义结构体/指针/引用/通道接口等），比较适用的场景是函数的调用次数，函数延迟，函数返回值等；
- 基于 uprobe 需要被跟踪的程序带有符号表（not stripped）
- eBPF 需要特权用户，一般情况下会限制适用范围；
- Go 语言可能会引起栈扩容，导致跟踪失效；



4. eBPF 与 Go

4.2 使用 eBPF 修改 Go 程序参数 – 演示

The image shows two terminal windows. The top window has a title bar that reads 'ubuntu@ubuntu: ~/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee (multipass)'. Inside, the command 'pwd' is entered, and the output is '/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee'. The bottom window has a title bar that reads 'ubuntu@ubuntu: ~/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracer (multipass)'. Inside, the prompt '[ubuntu@ubuntu]\$' is shown with a cursor, indicating the program is ready to be executed or modified.

```
ubuntu@ubuntu: ~/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee (multipass)
[ubuntu@ubuntu]$ pwd
/home/ubuntu/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracee
[ubuntu@ubuntu]$ █

ubuntu@ubuntu: ~/go/src/github.com/DavadDi/bpf_demo/bpf_go_tracer/tracer (multipass)
[ubuntu@ubuntu]$ █
```



4. eBPF 与 Go

4.2 使用 eBPF 修改 Go 程序参数 – 原理

```
[ubuntu@ubuntu]$ objdump --syms tracee | grep main.ebpfDemo
```

I

X ubuntu@ubuntu: ~/go/src/github.com/DavAdDi/bpf_demo/bpf_go_tracer/tracer (multipass)

```
[ubuntu@ubuntu]$ sudo ./tracer --binary ../tracee/tracee --func main.ebpfDemo
```

```
objdump -d tracee | grep main.ebpfDemo -B 100
```

4. eBPF 与 Go

4.2 使用 eBPF 修改 Go 程序参数 – 原理

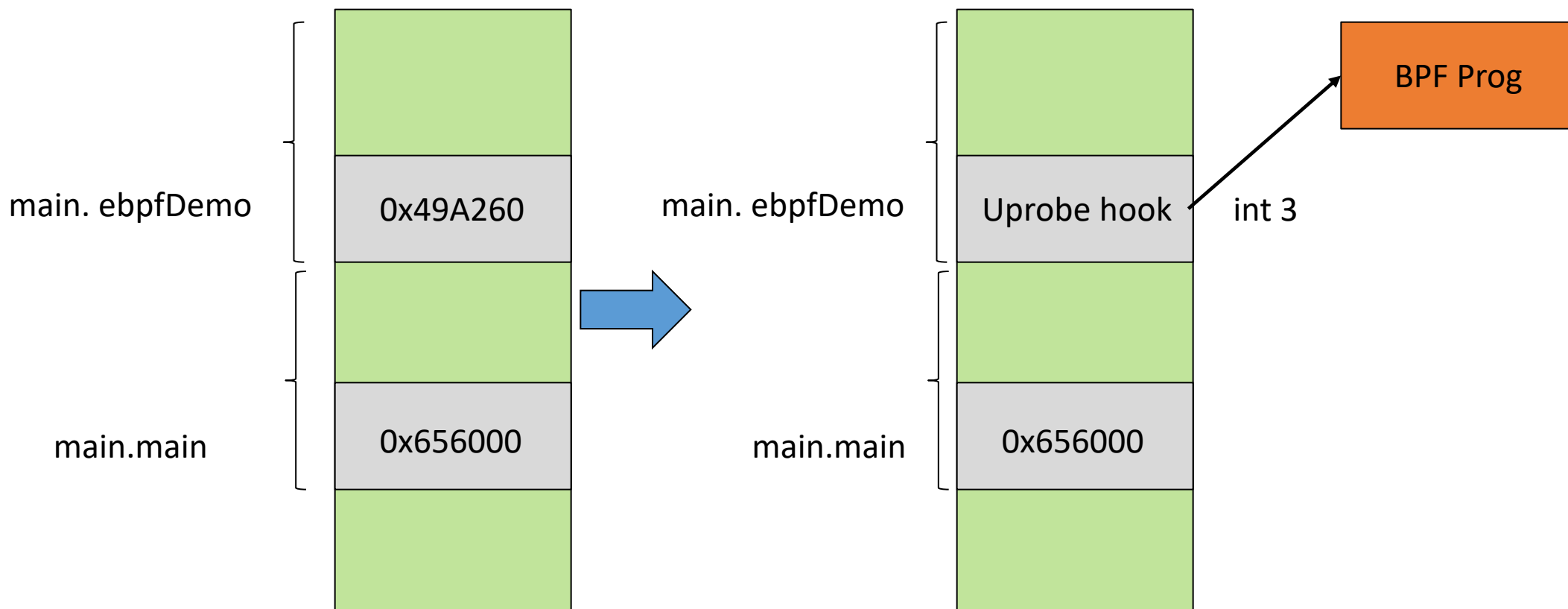
```
[ubuntu@ubuntu]$ objdump --syms trace | grep main.ebpfDemo
00000000004dc590 g      0 .rodata          0000000000000018 main.ebpfDemo.stkobj
000000000049a620 g      F .text       0000000000000165 main.ebpfDemo
```

```
000000000049a620 <main.ebpfDemo>:
49a620:      64 48 8b 0c 25 f8 ff      mov     %fs:0xfffffffffffffffff8,%rcx
49a627:      ff ff
49a629:      48 8d 44 24 f0            lea     -0x10(%rsp),%rax
49a62e:      48 3b 41 10              cmp     0x10(%rcx),%rax
49a632:      0f 86 41 01 00 00        jbe     49a779 <main.ebpfDemo+0x159>
49a638:      48 81 ec 90 00 00 00      sub     $0x90,%rsp
49a63f:      48 89 ac 24 88 00 00      mov     %rbp,0x88(%rsp)
49a646:      00
49a647:      48 8d ac 24 88 00 00      lea     0x88(%rsp),%rbp
```



4. eBPF 与 Go

4.2 使用 eBPF 修改 Go 程序参数 – 原理



4. eBPF 与 Go

4.2 使用 eBPF 修改 Go 程序参数 - 原理

```
(gdb) display /4i 0x49a620
```

```
7: x/4i 0x49a620
```

```
0x49a620 <main.ebpfDemo>:  mov    %fs:0xfffffffffffffffff8,%rcx
0x49a629 <main.ebpfDemo+9>:  lea    -0x10(%rsp),%rax
0x49a62e <main.ebpfDemo+14>:  cmp    0x10(%rcx),%rax
0x49a632 <main.ebpfDemo+18>:  jbe    0x49a779 <main.ebpfDemo+345>
```

```
(gdb) x/4i 0x49a620
```

```
0x49a620 <main.ebpfDemo>:  int3
0x49a621 <main.ebpfDemo+1>:  mov    0xfffffffffffffffff8,%rcx
0x49a629 <main.ebpfDemo+9>:  lea    -0x10(%rsp),%rax
0x49a62e <main.ebpfDemo+14>:  cmp    0x10(%rcx),%rax
```

```
(gdb)
```

4.3 使用 Go SDK

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

4.3 使用 Go SDK

```
1 static int test_sock(void)
2 {
3     int sock = -1, map_fd, prog_fd, i, key;
4     long long value = 0, tcp_cnt, udp_cnt, icmp_cnt;
5
6     map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key), sizeof(value), 256, 0);
7
8     struct bpf_insn prog[] = {
9         BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
10        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */)
11        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
12        // ...
13    };
14
15    size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
16    prog_fd = bpf_load_program(BPF_PROG_TYPE_SOCKET_FILTER, prog, insns_cnt,
17                               "GPL", 0, bpf_log_buf, BPF_LOG_BUF_SIZE);
18
19
20    sock = open_raw_sock("lo");
21    setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd))
22
23    for (i = 0; i < 10; i++) {
24        key = IPPROTO_TCP;
25        assert(bpf_map_lookup_elem(map_fd, &key, &tcp_cnt) == 0);
26
27        // ...
28        sleep(1);
29    }
30    // ...
31 }
```

数据结构

后端 BPF 程序

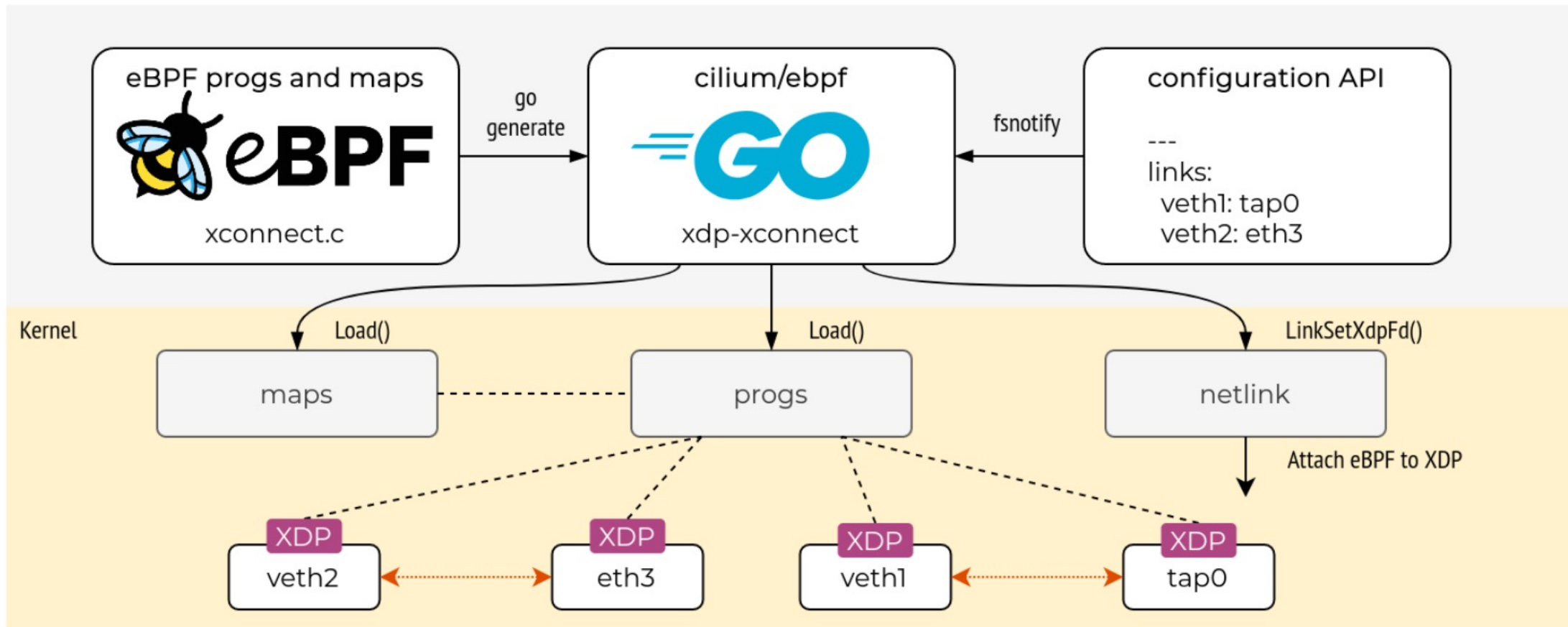
加载器

前端



4. eBPF 与 Go

4.3 使用 Go SDK





4. eBPF 与 Go

4.3 使用 Go SDK

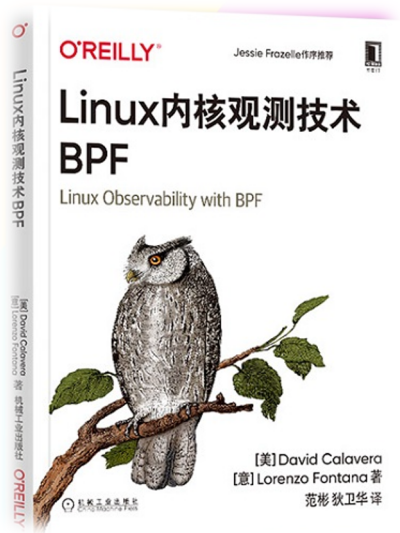
仓库名	描述	贡献者/核心贡献者	创建时间	更新时间	stars	备注
cilium/ebpf	eBPF Library for Go	43/7	2019-09-05	2021-08-10	1523	纯 Go 语言实现
dropbox/goebpf	Library to work with eBPF programs from Go	8/1	2019-01-24	2021-08-08	723	底层依赖libbpf（cgo）
iovisor/gobpf	Go bindings for creating BPF programs.	49/3	2016-11-18	2021-08-10	1409	底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

网址：

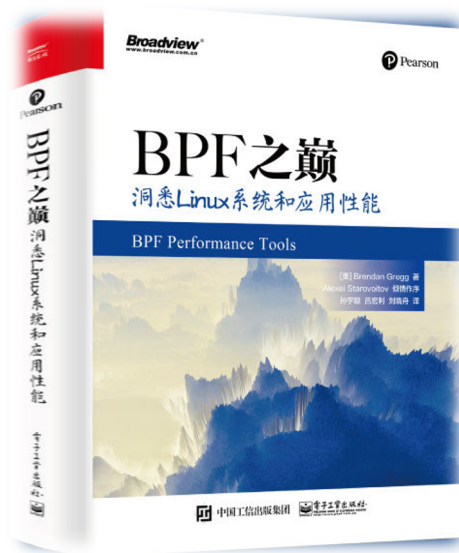
官方网址：<https://www.ebpf.io/>

深入浅出 BPF：<https://www.ebpf.top/>

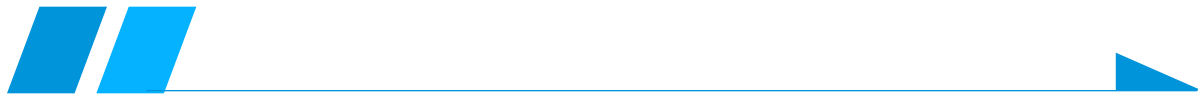
图书：



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!!!