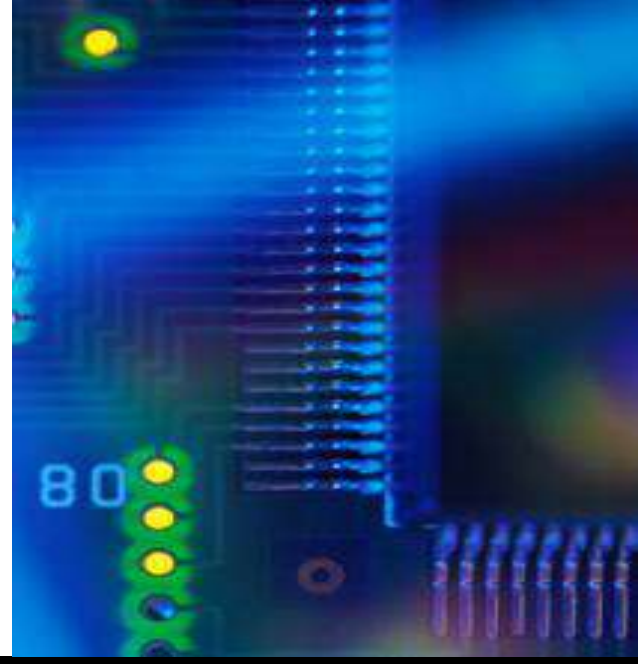
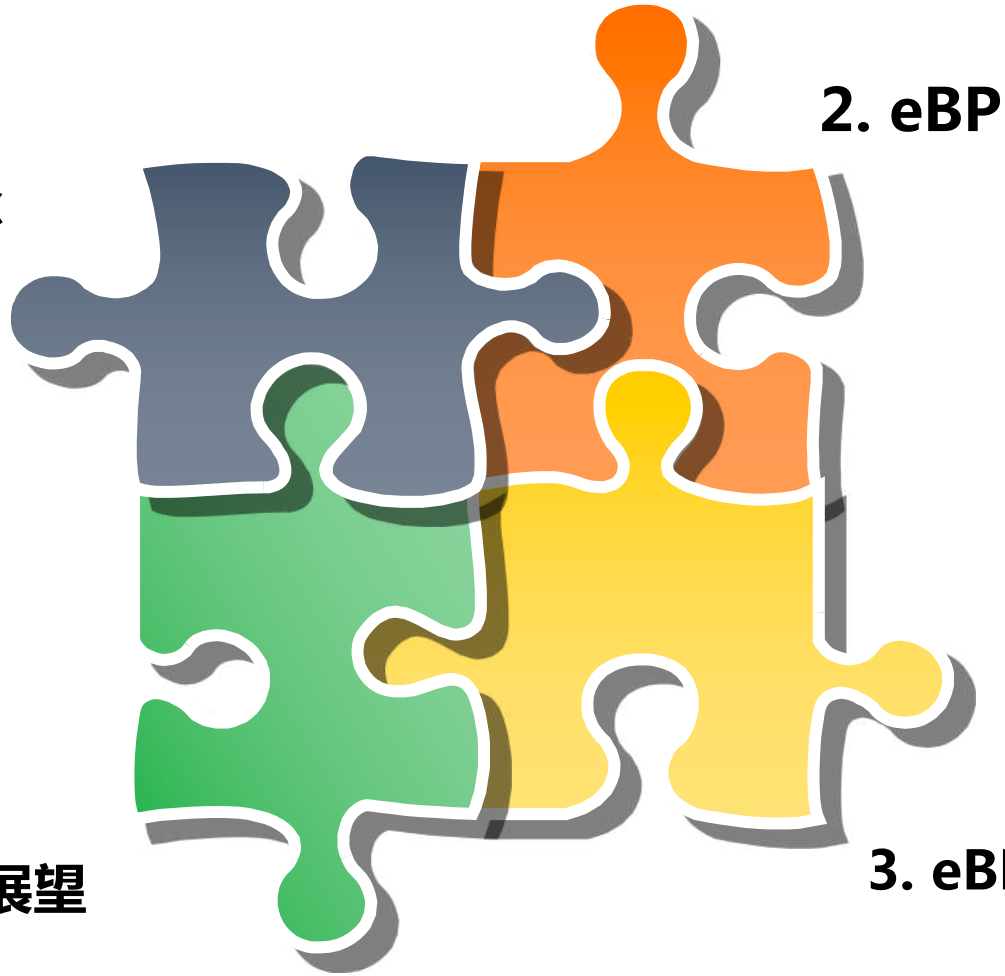


eBPF概念及在可观测性、网络、安全和优化领域的应用

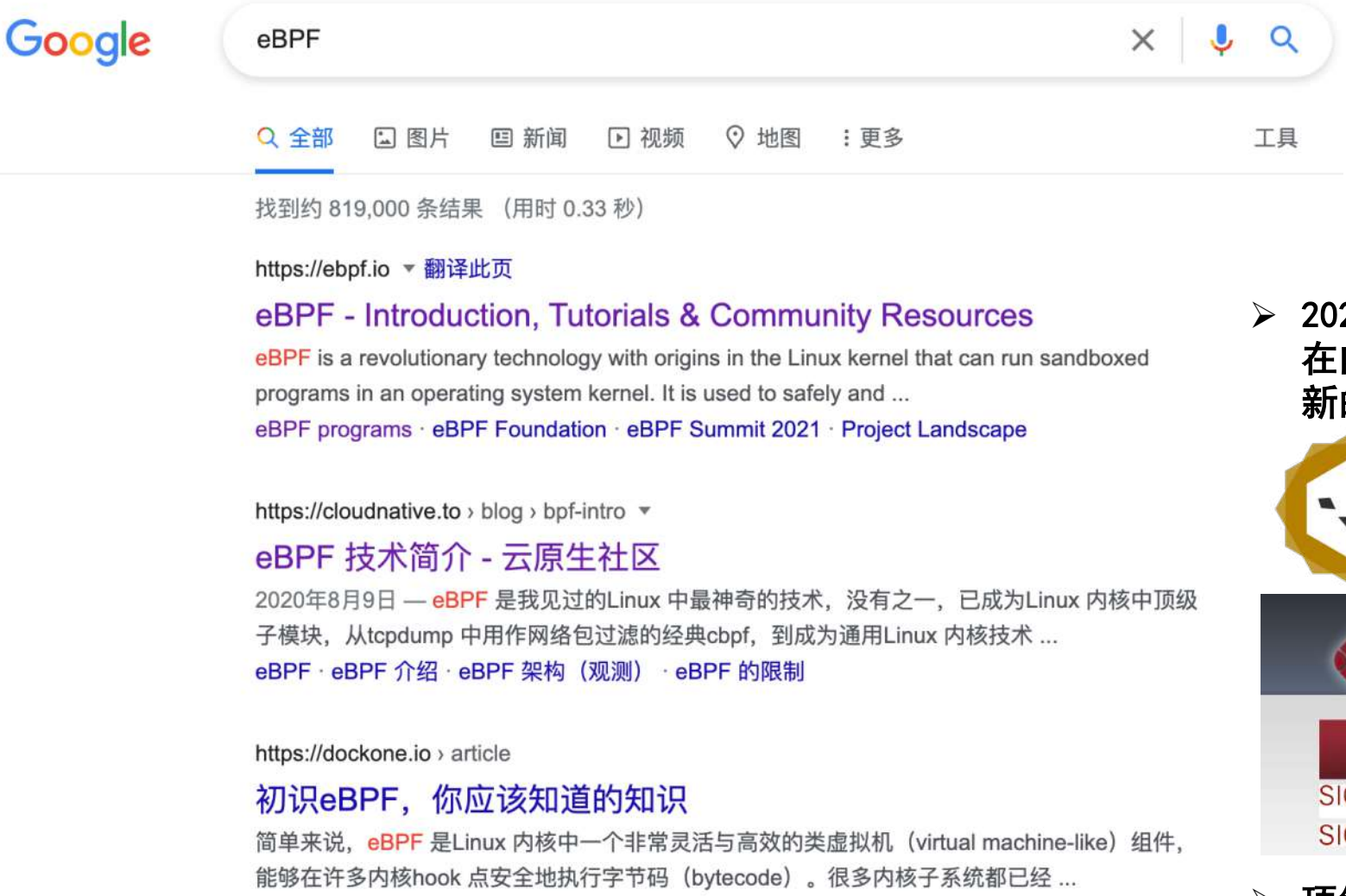
报告人：陈鹏飞
单位：中山大学计算机学院
邮箱：chenpf7@mail.sysu.edu.cn
网址：https://cse.sysu.edu.cn/content/3747
时间：2022.05.29

目录



● eBPF关注度

➢ Google 搜索引擎的页面数



eBPF Foundation

Founding Members

FACEBOOK Google ISOVALENT Microsoft NETFLIX

➢ 2021年，包括微软、谷歌、脸书、网飞、Isovalent在内的多家企业在 Linux 基金会麾下成立了一个全新的 eBPF 基金会。

THE WEB CONFERENCE ACM

FAST'22 20th USENIX Conference on File and Storage Technologies

USENIX ATC '22 2022 USENIX Annual Technical Conference

➢ 顶级学术会议的支持

1. eBPF概念

● eBPF概念

➢ eBPF正在努力让操作系统内核可编程化，成为云原生时代软件系统的“瑞士军刀”；

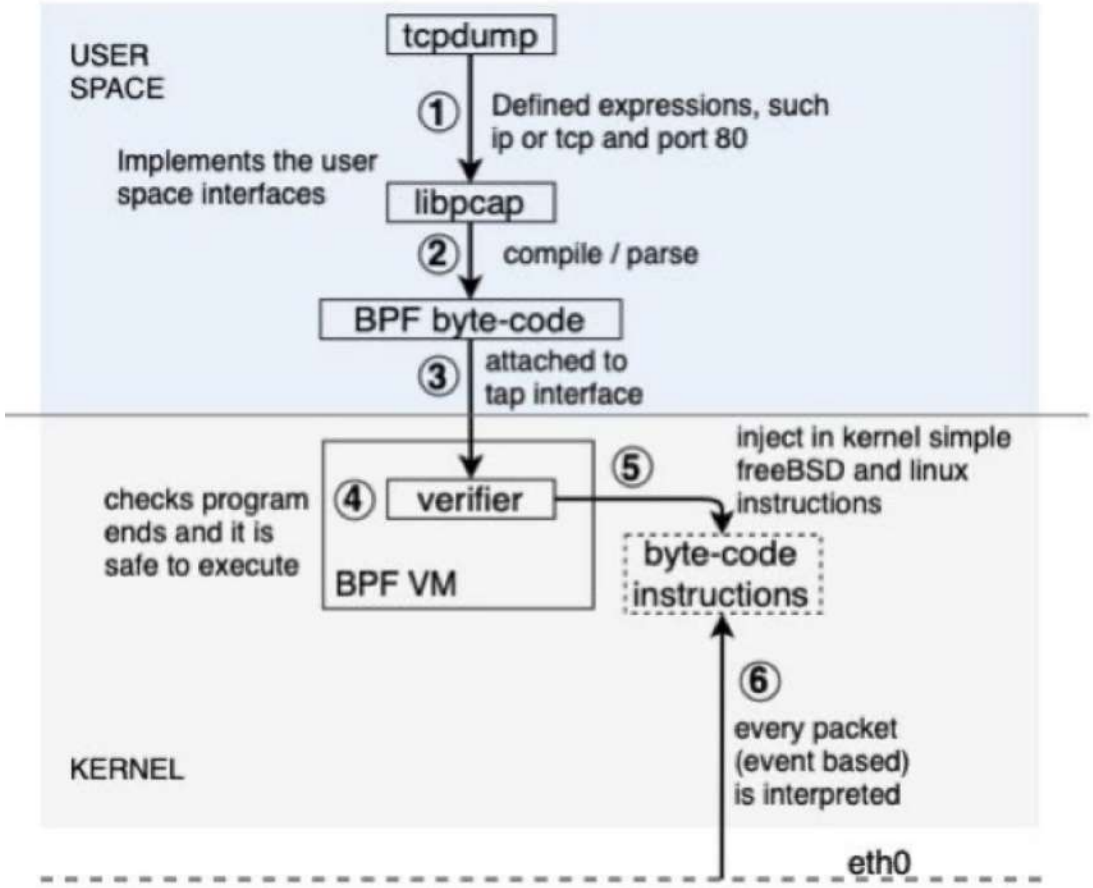
● eBPF概念

- eBPF（extended Berkeley Packet Filter），是Linux内核中一种高度灵活、高效的类似虚拟机的结构，允许以安全的方式在各种挂钩点执行字节码。它用于许多Linux内核子系统，最突出的是网络、跟踪和安全（例如沙箱）。

- eBPF的前身是BPF（cBPF），于1992年由Berkeley的McCanne and Jacobson 在Linux 2.2中引入；
- cBPF主要用于网络包过滤，后来用于构建安全沙箱，与seccomp联合使用；

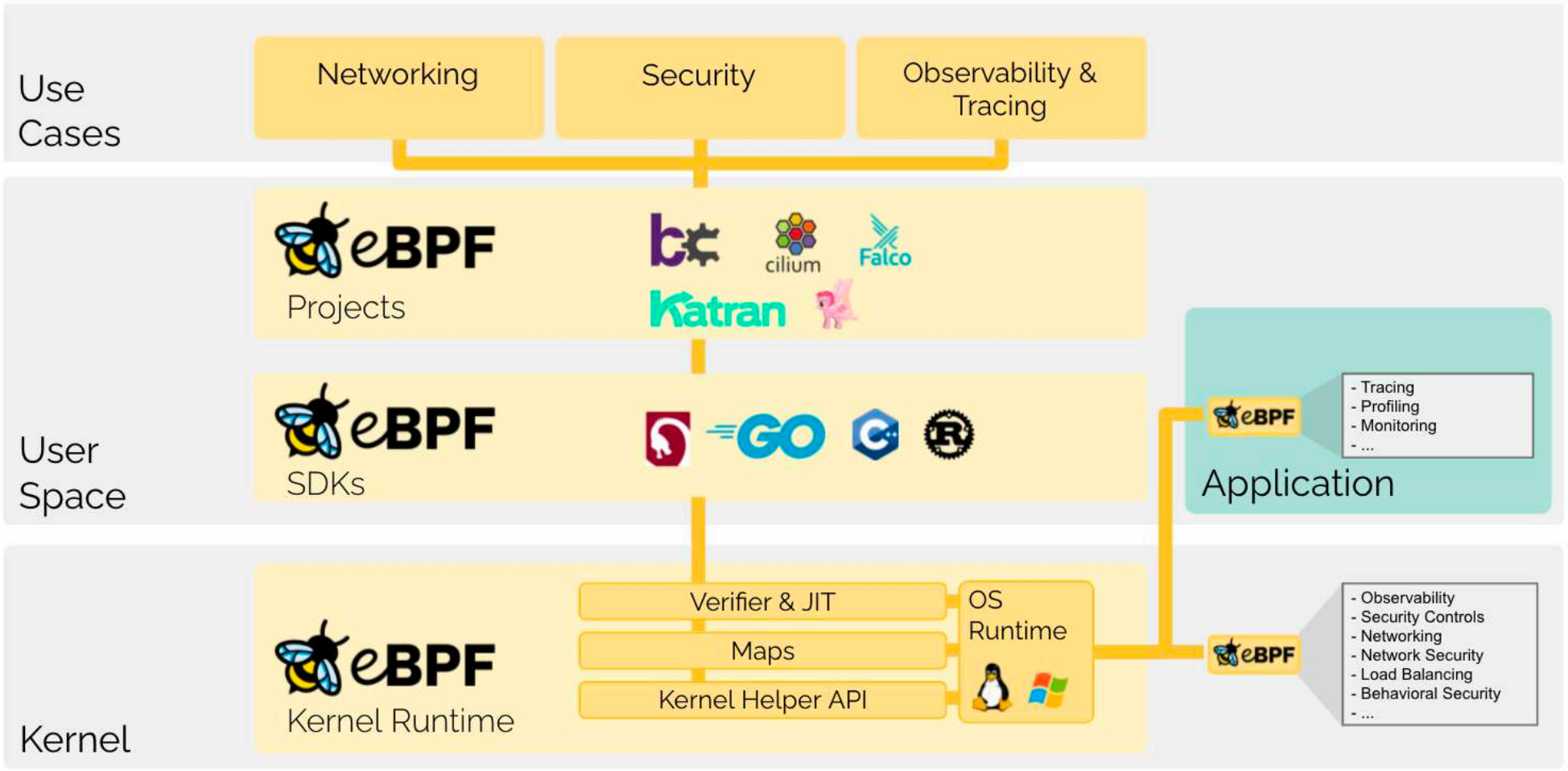
```
$ tcpdump -d 'ip and dst 186.173.190.239'
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 5
(002) ld       [30]
(003) jeq      #0xbaadbeef      jt 4    jf 5
(004) ret      #262144
(005) ret      #0
```

- cBPF的工作模式为：过滤规则→字节码→解释器；
- 在内核中执行的虚拟机，执行字节码，提供有限的指令，基于寄存器；



● eBPF概念

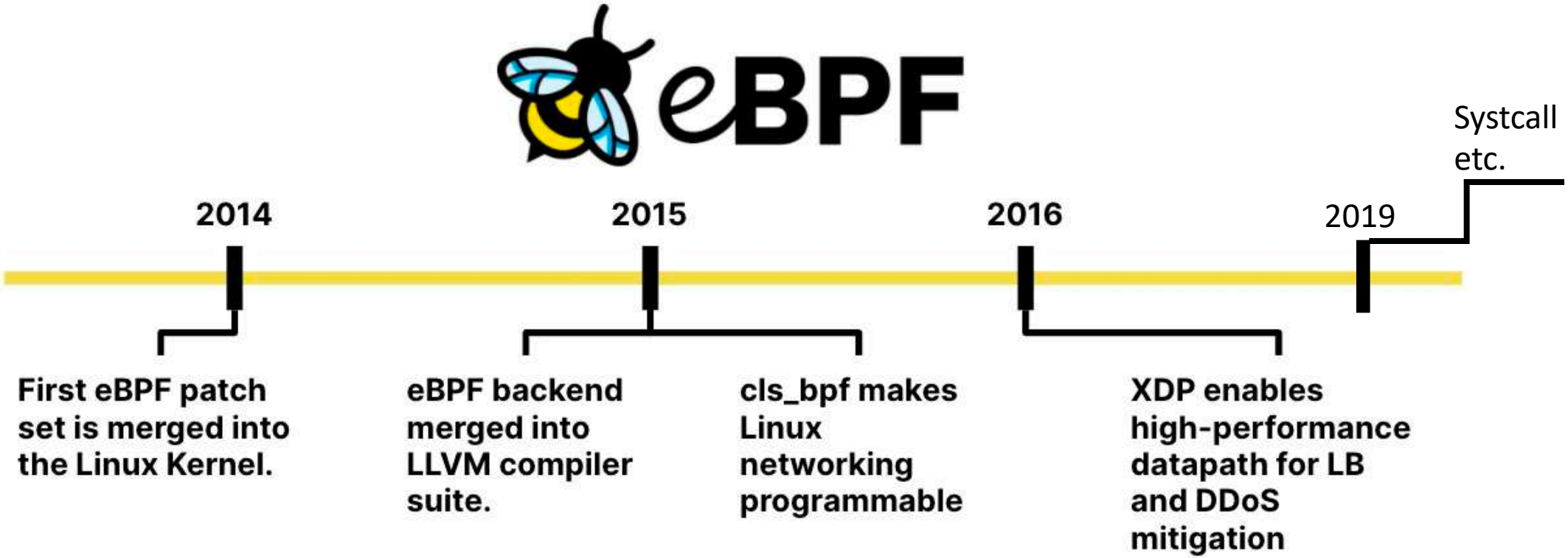
- 用户自定义程序，编译成字节码；
- 字节码被装载到内核VM中；
- 基于类RISC指令的VM运行在内核空间；
- 字节码通过多层验证机制确保内核安全；
- 验证后的字节码通过JIT机制翻译成机器指令；
- 通过eBPF Maps与用户空间交互；
- 提供有限的循环编程能力；
- 支持用户空间和内核空间的挂载，通用能力；



eBPF首次在Linux 内核3.18中出现，并在之后的版本中不断完善

● eBPF概念

➢ eBPF的演化过程；



https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md



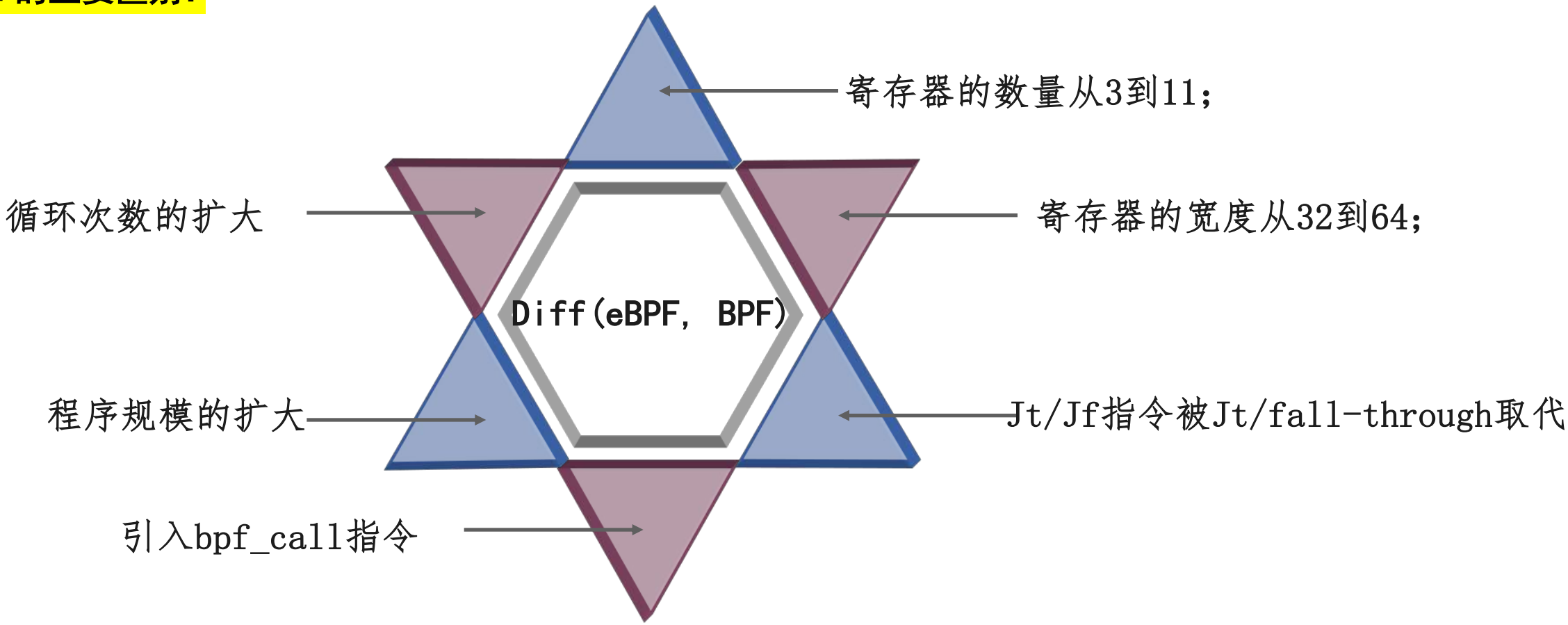
● eBPF概念

➢ eBPF的演化过程；

Feature	Kernel version
`AF_PACKET` (libpcap/tcpdump, `cls_bpf` classifier, netfilter's `xt_bpf`, team driver's load-balancing mode...)	3.15
Kernel helpers	3.15
`bpf()` syscall	3.18
Tables (*a.k.a.* Maps; details below)	3.18
BPF attached to sockets	3.19
BPF attached to `kprobes`	4.1
`cls_bpf` / `act_bpf` for `tc`	4.1
Tail calls	4.2
Non-root programs on sockets	4.4
Persistent maps and programs (virtual FS)	4.4
`tc` 's `direct-action` (`da`) mode	4.4
`tc` 's `clsact` qdisc	4.5
BPF attached to tracepoints	4.7
Direct packet access	4.7
XDP (see below)	4.8
BPF attached to perf events	4.9
Hardware offload for `tc` 's `cls_bpf`	4.9
Verifier exposure and internal hooks	4.9
BPF attached to cgroups for socket filtering	4.10
Lightweight tunnel encapsulation	4.10
BPF attached to cgroups as device controller	4.15
bpf2bpf function calls	4.16
BPF used for monitoring socket RX/TX data	4.17
BPF attached to raw tracepoints	4.17
BPF attached to `bind()` system call	4.17
BPF Type Format (BTF)	4.18
AF_XDP	4.18
bpfilter	4.18
End.BPF action for seg6local LWT	4.18
BPF attached to LIRC devices	4.18
Pass map values to map helpers	4.18
BPF socket reuseport	4.19
BPF flow dissector	4.20
BPF 1M insn limit	5.2
BPF cgroup sysctl	5.2
BPF raw tracepoint writable	5.2
BPF trampoline	5.5
BPF LSM hook	5.7
BPF iterator	5.8
BPF socket lookup hook	5.9
Sleepable BPF programs	5.10

● eBPF概念

➢ eBPF与BPF的主要区别：



https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html#

● eBPF概念

➢ 内核虚拟机： 基于RISC寄存器虚拟机，共有11个64位寄存器、一个程序计数器和512字节固定大小堆栈。9个寄存器是通用的读写寄存器，一个是只读堆栈指针，程序计数器是隐式的。

r0:	stores return values, both for function calls and the current program exit code
r1-r5:	used as function call arguments, upon program start r1 contains the "context" argument pointer
r6-r9:	these get preserved between kernel function calls
r10:	read-only pointer to the per-eBPF program 512 byte stack

➢ JIT: Just In Time， 指令即时翻译

```
bpf_mov R6, R1 /* save ctx */
bpf_mov R2, 2
bpf_mov R3, 3
bpf_mov R4, 4
bpf_mov R5, 5
```

eBPF伪指令

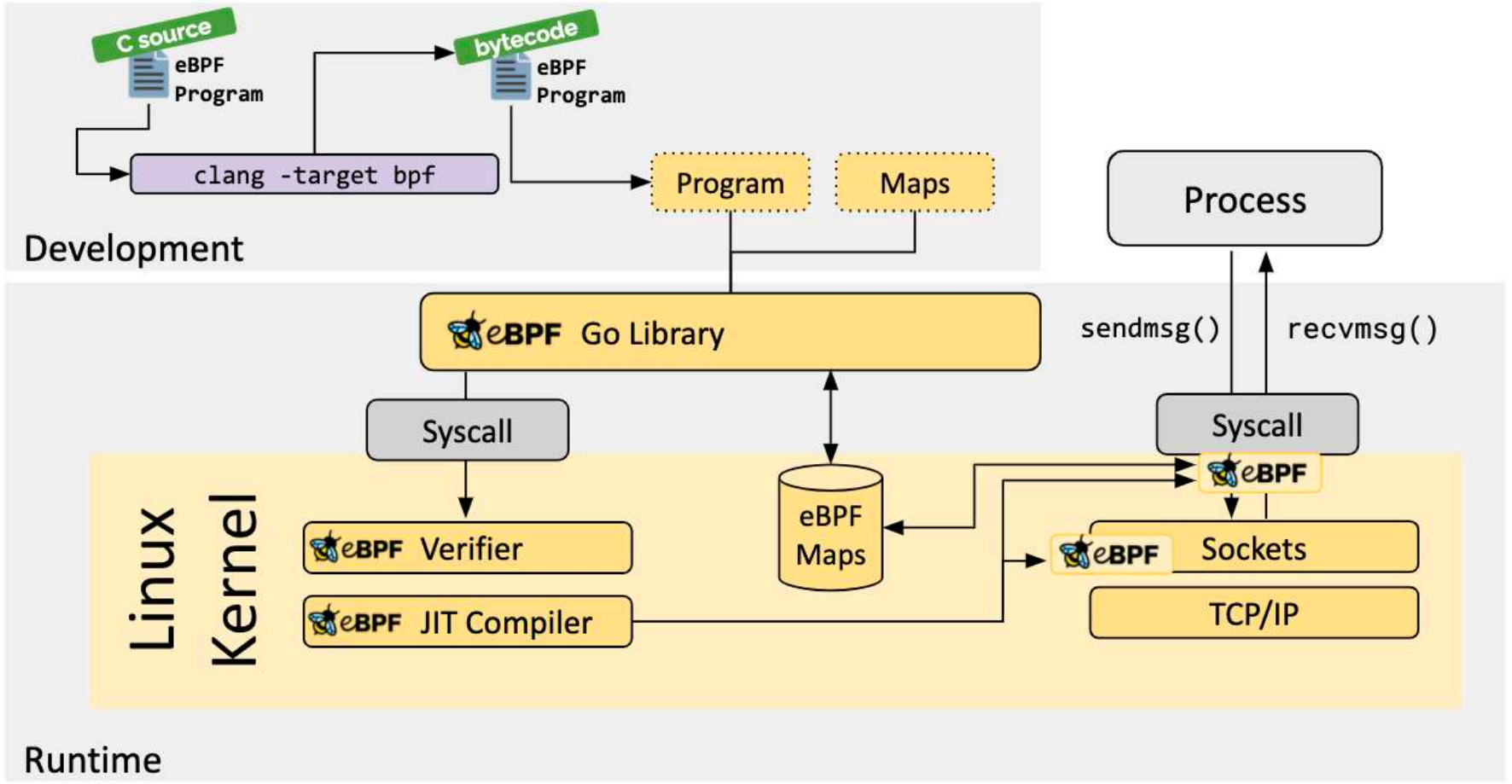
```
push %rbp
mov %rsp,%rbp
sub $0x228,%rsp
mov %rbx,-0x228(%rbp)
mov %r13,-0x220(%rbp)
mov %rdi,%rbx
mov $0x2,%esi
```

X86_64机器指令

● eBPF概念

int bpf(int cmd, union bpf_attr *attr, unsigned int size);

➢ **程序的装载过程：通过bpf()系统调用将eBPF程序装载到内核中，包含多种bpf命令，用于用户空间与内核VM交互；**




```
enum bpf_cmd {
        BPF_MAP_CREATE,
        BPF_MAP_LOOKUP_ELEM,
        BPF_MAP_UPDATE_ELEM,
        BPF_MAP_DELETE_ELEM,
        BPF_MAP_GET_NEXT_KEY,
        BPF_PROG_LOAD,
        BPF_OBJ_PIN,
        BPF_OBJ_GET,
        BPF_PROG_ATTACH,
        BPF_PROG_DETACH,
        BPF_PROG_TEST_RUN,
        BPF_PROG_RUN = BPF_PROG_TEST_RUN,
        BPF_PROG_GET_NEXT_ID,
        BPF_MAP_GET_NEXT_ID,
        BPF_PROG_GET_FD_BY_ID,
        BPF_MAP_GET_FD_BY_ID,
        BPF_OBJ_GET_INFO_BY_FD,
        BPF_PROG_QUERY,
        BPF_RAW_TRACEPOINT_OPEN,
        BPF_BTF_LOAD,
        BPF_BTF_GET_FD_BY_ID,
        BPF_TASK_FD_QUERY,
        BPF_MAP_LOOKUP_AND_DELETE_ELEM,
        BPF_MAP_FREEZE,
        BPF_BTF_GET_NEXT_ID,
        BPF_MAP_LOOKUP_BATCH,
        BPF_MAP_LOOKUP_AND_DELETE_BATCH,
        BPF_MAP_UPDATE_BATCH,
        BPF_MAP_DELETE_BATCH,
        BPF_LINK_CREATE,
        BPF_LINK_UPDATE,
        BPF_LINK_GET_FD_BY_ID,
        BPF_LINK_GET_NEXT_ID,
        BPF_ENABLE_STATS,
        BPF_ITER_CREATE,
        BPF_LINK_DETACH,
        BPF_PROG_BIND_MAP,
};
```

bpf命令



● eBPF概念

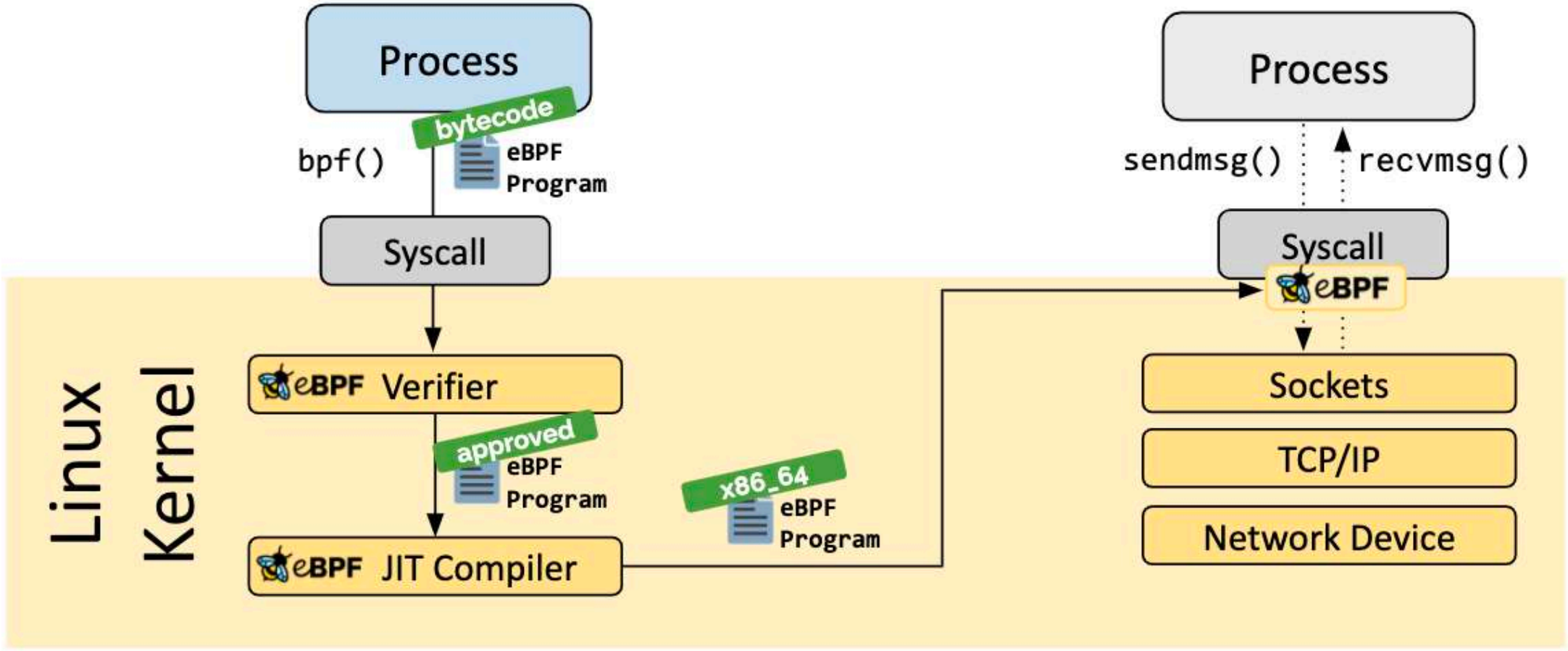
➢ eBPF程序类型：Linux 本身的能力非常多，而提供给外部可调用的方法是很少的，需要为不同的场景和能力，开放不同的系统调用能力，哪些能力在哪些类型下可以被调用，跟 eBPF 的程序类型有关系

- BPF_PROG_TYPE_SOCKET_FILTER: a network packet filter
- BPF_PROG_TYPE_KPROBE: determine whether a kprobe should fire or not
- BPF_PROG_TYPE_SCHED_CLS: a network traffic-control classifier
- BPF_PROG_TYPE_SCHED_ACT: a network traffic-control action
- BPF_PROG_TYPE_TRACEPOINT: determine whether a tracepoint should fire or not
- BPF_PROG_TYPE_XDP: a network packet filter run from the device-driver receive path
- BPF_PROG_TYPE_PERF_EVENT: determine whether a perf event handler should fire or not
- BPF_PROG_TYPE_CGROUP_SKB: a network packet filter for control groups
- BPF_PROG_TYPE_CGROUP_SOCK: a network packet filter for control groups that is allowed to modify socket options
- BPF_PROG_TYPE_LWT_*: a network packet filter for lightweight tunnels
- BPF_PROG_TYPE_SOCK_OPS: a program for setting socket parameters
- BPF_PROG_TYPE_SK_SKB: a network packet filter for forwarding packets between sockets
- BPF_PROG_CGROUP_DEVICE: determine if a device operation should be permitted or not

当前eBPF支持的程序类型，与eBPF_helper提供的开放函数相关联，可以自定义添加

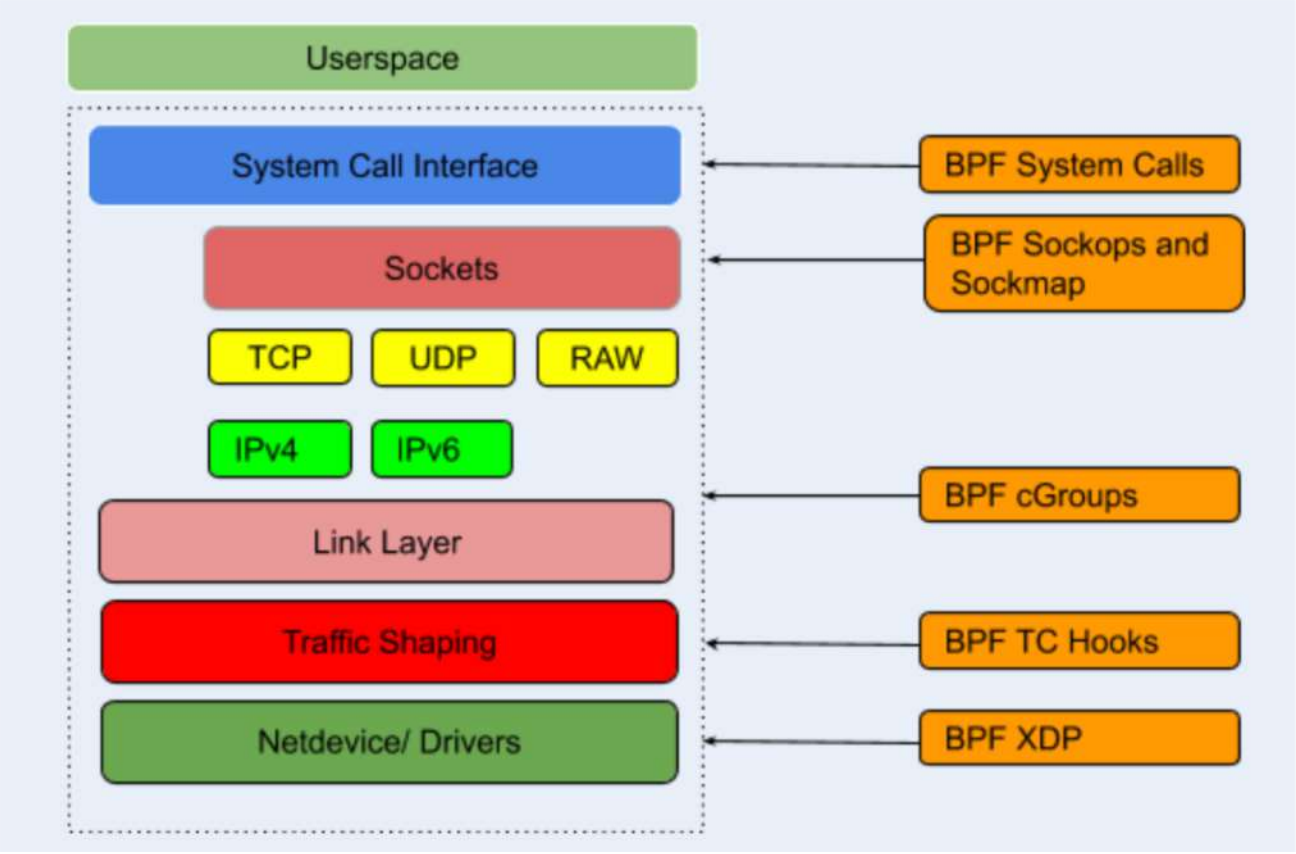
● eBPF概念

➢ 程序验证：保证程序的安全执行，不破坏内核。主要验证程序的权限、程序不能对内核有损害、程序能够结束（循环有限）等；

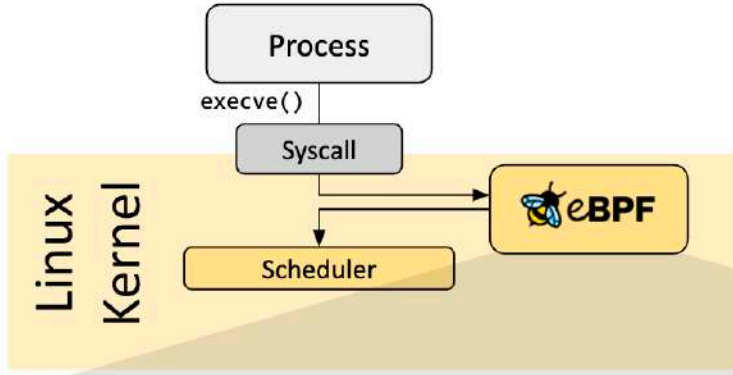


● eBPF概念

➢ eBPF钩子程序（Hook）：eBPF程序是事件驱动的，当内核或应用程序通过某个挂钩点时运行。预定义的钩子包括系统调用、函数入口/出口、内核跟踪点、网络事件和其他钩子；



eBPF可钩挂的函数

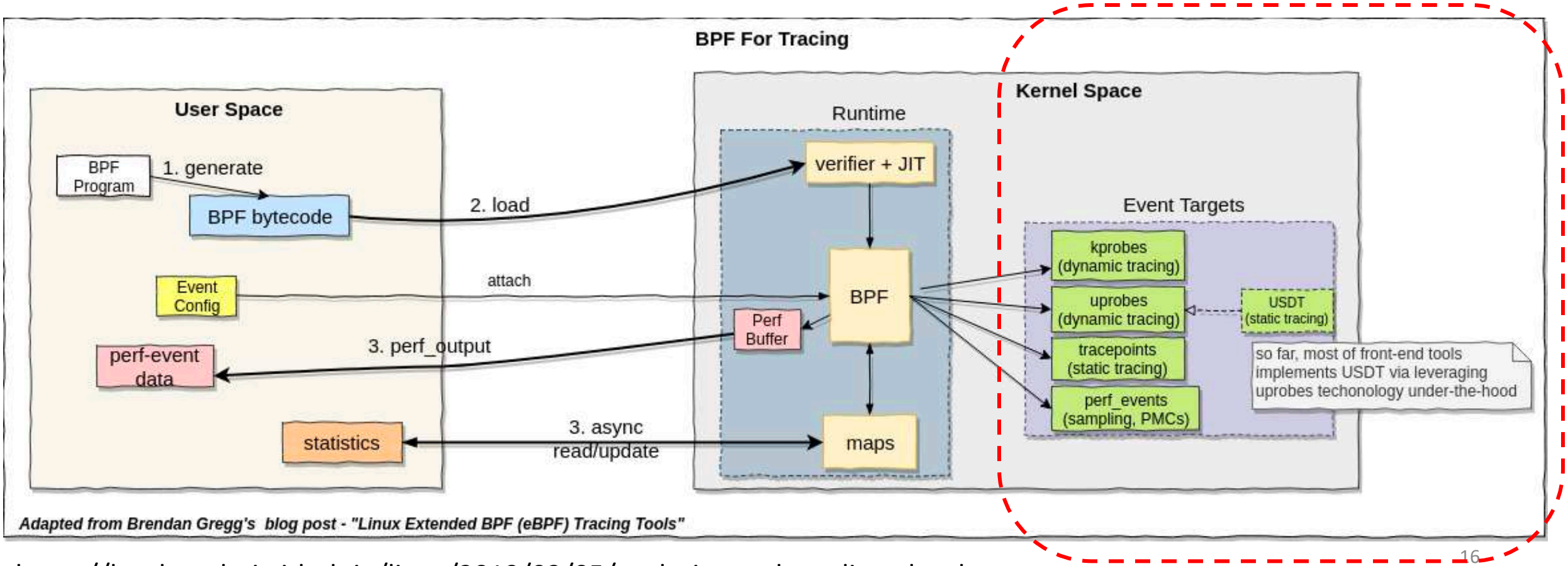



```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

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

        return 0;
}
```

eBPF 钩挂过程

● eBPF概念

➢ eBPF钩子程序（Hook）：主要涉及Kprobe、Tracepoint、Uprobe以及USDT；



https://leezhenghui.github.io/linux/2019/03/05/exploring-usdt-on-linux.html

● eBPF概念

➢ eBPF钩子程序（Hook）：主要涉及Kprobe、Tracepoint、Uprobe以及USDT；

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

#define MAX_SYMBOL_LEN  64
static char symbol[MAX_SYMBOL_LEN] = "_do_fork";
module_param_string(symbol, symbol, sizeof(symbol), 0644);

/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
    .symbol_name  = symbol,
};
static void __exit kprobe_exit(void)
{
    unregister_kprobe(&kp);
    pr_info("kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

```
#include <stdio.h>
#include <stdlib.h>

void f()
{
  printf("f() called\n");
}
int main()
{
   f();
  return 0;
}
```

echo 'p /home/test/uprobe/test:0x
/sys/kernel/debug/tracing/uprobe_e

```
#define DECLARE_TRACE(name, proto, args)                \
    __DECLARE_TRACE(name, PARAMS(proto), PARAMS(args),  \
            cpu_online(raw_smp_processor_id()),         \
            PARAMS(void *__data, proto),                \
            PARAMS(__data, args))


#define __DECLARE_TRACE(name, proto, args, cond, data_proto, data_args) \
    extern struct tracepoint __tracepoint_##name;       \
    static inline void trace_##name(proto)              \
    {                                                   \
        if (static_key_false(&__tracepoint_##name.key))         \
            __DO_TRACE(&__tracepoint_##name,            \
                TP_PROTO(data_proto),                   \
                TP_ARGS(data_args),                     \
                TP_CONDITION(cond), 0);                 \
        if (IS_ENABLED(CONFIG_LOCKDEP) && (cond)) {             \
            rcu_read_lock_sched_notrace();              \
            rcu_dereference_sched(__tracepoint_##name.funcs);\
            rcu_read_unlock_sched_notrace();            \
        }                                               \
    }                                                   \
```

```
$ find /usr/lib/jvm -name libjvm.so -exec
readelf -n {} + | grep -A2 NT_STAPSDT
  stapsdt              0x00000078       NT_STAP
SDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: mem__pool__gc__begin
--
  stapsdt              0x0000004d       NT_STAP
SDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: class__loaded
--
  stapsdt              0x0000004e       NT_STAP
SDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: object__alloc
--
  stapsdt              0x00000065       NT_STAP
SDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: thread__start
…
```

➢ Kprobe内核探针，可以在内核中的任意位置定义，包括函数进入、退出以及函数中间；

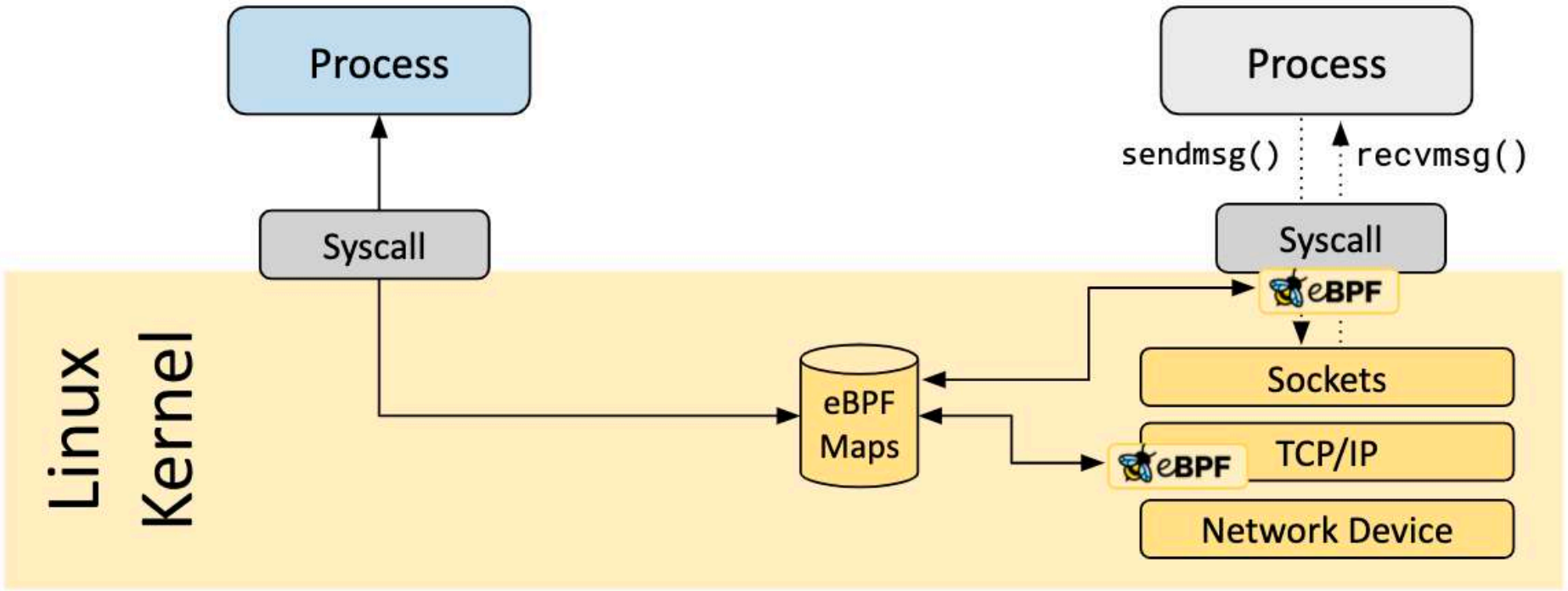
➢ Uprobe用户探针，在用户空间定义的探针，可以定义在用户函数的任意位置；

➢ Tracepoint事先定义在内核中的追踪点，相对固定，一般在函数的入口和出口；

➢ USDT：用户态静态定义的追踪点，目前已经在各种软件中大量使用；

● eBPF概念

- eBPF Maps：eBPF程序的一个重要方面是共享收集的信息和存储状态的能力。为此，eBPF程序可以利用eBPF Map来存储和检索大量数据结构中的数据。



eBPF Maps的处理流程

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        BPF_MAP_TYPE_SK_STORAGE,
        BPF_MAP_TYPE_DEVMAP_HASH,
        BPF_MAP_TYPE_STRUCT_OPS,
        BPF_MAP_TYPE_RINGBUF,
        BPF_MAP_TYPE_INODE_STORAGE,
        BPF_MAP_TYPE_TASK_STORAGE,
        BPF_MAP_TYPE_BLOOM_FILTER,
};
```

● eBPF概念

➢ eBPF Helper函数：eBPF程序不能调用任意内核函数。如果允许这样做会将eBPF程序绑定到特定的内核版本，并会使程序的兼容性复杂化。相反，eBPF程序可以对helper函数进行调用，这是内核提供的一种稳定的API，可自定义；

eBPF Helper函数的执行过程

```
long bpf_skb_store_bytes(struct sk_buff *skb, u32 offset, const
void *from, u32 len, u64 flags)

        Description
               Store len bytes from address from into the packet
               associated to skb, at offset. flags are a
               combination of BPF_F_RECOMPUTE_CSUM (automatically
               recompute the checksum for the packet after storing
               the bytes) and BPF_F_INVALIDATE_HASH (set
               skb->hash, skb->swhash and skb->l4hash to 0).

               A call to this helper is susceptible to change the
               underlying packet buffer. Therefore, at load time,
               all checks on pointers previously done by the
               verifier are invalidated and must be performed
               again, if the helper is used in combination with
               direct packet access.

        Return 0 on success, or a negative error in case of
               failure.

long bpf_l3_csum_replace(struct sk_buff *skb, u32 offset, u64
from, u64 to, u64 size)
```

Helper函数举例



● eBPF概念

- eBPF 安全防护：如何保障eBPF程序的安全性，不造成信息的泄露；
  - 需要超级权限，只有授信的程序才能装载eBPF程序；
  - 验证器，采用多种验证规则，保障程序的正确性、安全性 （最小攻击面）；
  - 安全加固技术，如程序执行保护、常量混淆等；
  - 抽象运行时上下文，隔离数据，通过eBPF Helper函数访问；

最终实现操作系统内核的可编程性！

2. eBPF编程



● eBPF编程

BCC 编程

- BCC 依靠运行时间汇编，将整个大型 LLVM/Clang 库带入并嵌入其中；
- 编译过程中资源利用量大（内存和 CPU），可能会中断繁忙服务器上的主要工作流；
- 依赖于内核头包，该包必须安装在每个目标主机上；
- 即使是微不足道的编译时错误也只在运行时间检测到：这显著缩短了开发迭代时间；

入门！

Libbpf+BPF+CORE 编程

- BPF程序与任何"正常"用户空间程序没有太大区别：
- 汇编成小型二进制文件，然后以紧凑的形式进行部署，以适应不同环境；
- Libbpf 扮演 BPF 程序装载机的角色，执行平凡的设置工作（重定位、加载和验证 BPF 程序、创建 BPF map、连接到 BPF 挂钩等）；
- 开发人员只需要关系 BPF 程序的正确性和性能。消除沉重的依赖关系，使整体开发人员体验更加愉快。

进阶！



● eBPF编程

➢BCC 编程

- 安装最新的操作系统内核，建议5.10以上（包括新型eBPF特征），Ubuntu 20.04 OS；
- 确保内核中的配置已经开启；
- 安装bcc工具；

sudo apt-get install bpfcc-tools linux-headers-$(uname -r)

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_FTRACE_SYSCALLS=y
CONFIG_FUNCTION_TRACER=y
CONFIG_HAVE_DYNAMIC_FTRACE=y
CONFIG_DYNAMIC_FTRACE=y
CONFIG_HAVE_KPROBES=y
CONFIG_KPROBES=y
CONFIG_KPROBE_EVENTS=y
CONFIG_ARCH_SUPPORTS_UPROBES=y
CONFIG_UPROBES=y
CONFIG_UPROBE_EVENTS=y
CONFIG_DEBUG_FS=y
CONFIG_DEBUG_INFO_BTF=y
```

特性	引入版本	功能介绍	应用场景
Tc-bpf	4.1	eBPF 重构内核流分类	网络
XDP	4.8	网络数据面编程技术（主要面向 L2/L3 层业务）	网络
Cgroup socket	4.10	Cgroup 内 socket 支持 eBPF 扩展逻辑	容器
AF_XDP	4.18	网络原始报文直送用户态（类似 DPDK）	网络
Sockmap	4.20	支持 socket 短接	容器
Device JIT	4.20	JIT/ISA 解耦，host 可以编译指定 device 形态的 ISA 指令	异构编程
Cgroup sysctl	5.2	Cgroup 内支持控制系统调用权限	容器
Struct ops Prog ext	5.3	内核逻辑可动态替换 eBPF Prog 可动态替换	框架基础
Bpf trampoline	5.5	三种用途：1.内核中代替 K(ret)probe,性能更优 2.eBPF Prog 中使用，解决 eBPF Prog 调试问题 3.实现 eBPF Prog 动态链接功能（未来功能）	性能跟踪
KRSI（lsm + eBPF）	5.7	内核运行时安全策略可定制	安全
Ring buffer	5.8	提供 CPU 间共享的环形 buffer，并能实现跨 CPU 的事件保序记录。用以代替 perf/ftrace 等 buffer。	跟踪/性能分析

● eBPF编程

➢BCC 编程案例

eBPF code

```
5  from bcc import BPF
6  from bcc.utils import printb
7
8  # define BPF program
9  prog = """
10 #include <linux/sched.h>
11
12 // define output data structure in C
13 struct data_t {
14     u32 pid;
15     u64 ts;
16     char comm[TASK_COMM_LEN];
17 };
18 BPF_PERF_OUTPUT(events);
19
20 int hello(struct pt_regs *ctx) {
21     struct data_t data = {};
22
23     data.pid = bpf_get_current_pid_tgid();
24     data.ts = bpf_ktime_get_ns();
25     bpf_get_current_comm(&data.comm, sizeof(data.comm));
26
27     events.perf_submit(ctx, &data, sizeof(data));
28
29     return 0;
30 }
31 """
32
33 # load BPF program
34 b = BPF(text=prog)
35 b.attach_kprobe(event=b.get_syscall_fnname("clone"), fn_name="hello")
36
37 # header
38 print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "MESSAGE"))
```

```
40 # process event
41 start = 0
42 def print_event(cpu, data, size):
43     global start
44     event = b["events"].event(data)
45     if start == 0:
46             start = event.ts
47     time_s = (float(event.ts - start)) / 1000000000
48     printb(b"%-18.9f %-16s %-6d %s" % (time_s, event.comm, event.pid,
49         b"Hello, perf_output!"))
50
51 # loop with callback to print_event
52 b["events"].open_perf_buffer(print_event)
53 while 1:
54     try:
55         b.perf_buffer_poll()
56     except KeyboardInterrupt:
57         exit()
```

● eBPF编程

- Libbpf+BTF+CORE编程
  - 与内核版本解耦；
  - 利用原生C语言编写eBPF程序；
  - 安装bpftool，用于生成Linux内核对应的vmlinux.h；

```
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
...) = (void *)BPF_FUNC_trace_printk;

SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(void *ctx) {
char msg[] = "Hello, BPF World!";
bpf_trace_printk(msg, sizeof(msg));
return 0;
}

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

内核空间

```
dockerd-1594963 [000] d... 135666.605635: bpf_trace_printk: Hello, BPF World!
dockerd-1594969 [000] d... 135666.613114: bpf_trace_printk: Hello, BPF World!
flanneld-1594970 [003] d... 135668.418537: bpf_trace_printk: Hello, BPF World!
flanneld-1594971 [002] d... 135668.420136: bpf_trace_printk: Hello, BPF World!
flanneld-1594972 [002] d... 135668.422139: bpf_trace_printk: Hello, BPF World!
flanneld-1594973 [003] d... 135668.424652: bpf_trace_printk: Hello, BPF World!
```

```
static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args)
{
return vfprintf(stderr, format, args);
}

static void bump_memlock_rlimit(void)
{
struct rlimit rlim_new = {
.rlim_cur = RLIM_INFINITY,
.rlim_max = RLIM_INFINITY,
};


if (setrlimit(RLIMIT_MEMLOCK, &rlim_new)) {
fprintf(stderr, "Failed to increase RLIMIT_MEMLOCK limit!\n");
exit(1);
}
}

int main(int argc, char **argv)
{
struct hello_bpf *skel;
int err;

/* Set up libbpf errors and debug info callback */
libbpf_set_print(libbpf_print_fn);

/* Bump RLIMIT_MEMLOCK to allow BPF sub-system to do anything */
bump_memlock_rlimit();

/* Open BPF application */
skel = hello_bpf__open();
if (!skel) {
fprintf(stderr, "Failed to open BPF skeleton\n");
return 1;
}

/* Load & verify BPF programs */
err = hello_bpf__load(skel);
if (err) {
fprintf(stderr, "Failed to load and verify BPF skeleton\n");
goto cleanup;
}

/* Attach tracepoint handler */
```

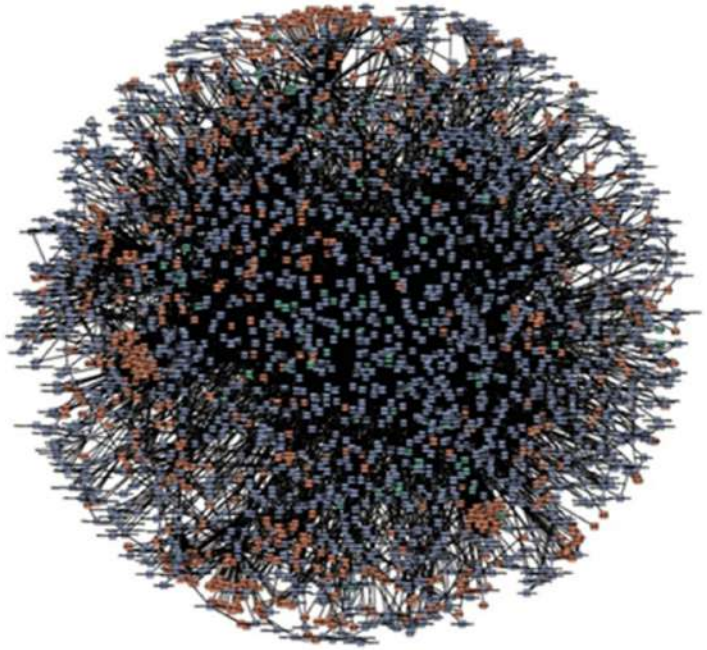
用户空间

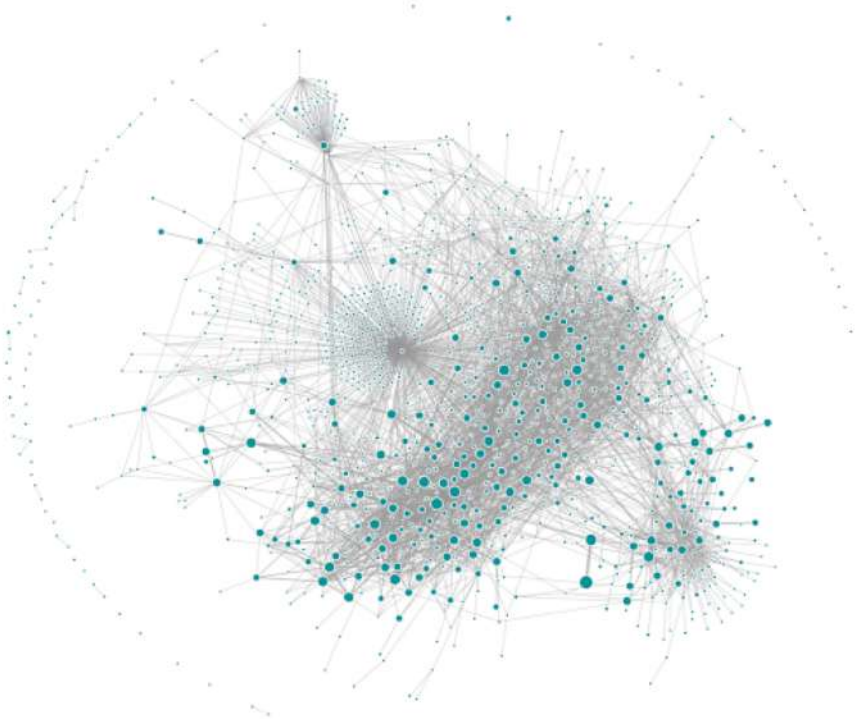
BCC向libbpf迁移！

3. eBPF应用

● 现代软件系统的复杂性

- The scale of modern software system increases exponentially, for example, **WeChat has about 3000 services, Netflix has 700 kinds of services, and Uber has about 2200 services.**
- The interactions amongst services are quite complex, the end-to-end tracing can span across **dozens of** services.



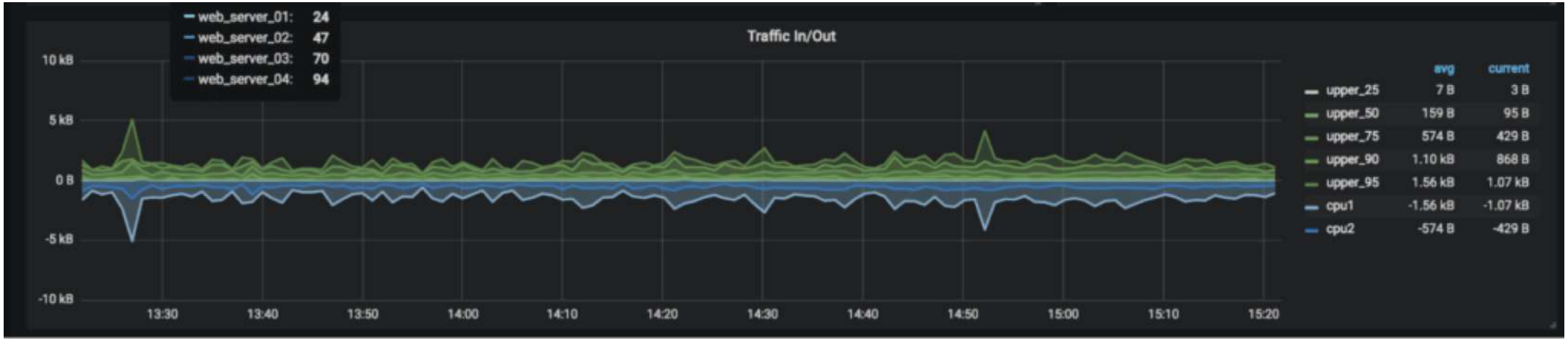


amazon.com

NETFLIX

Uber

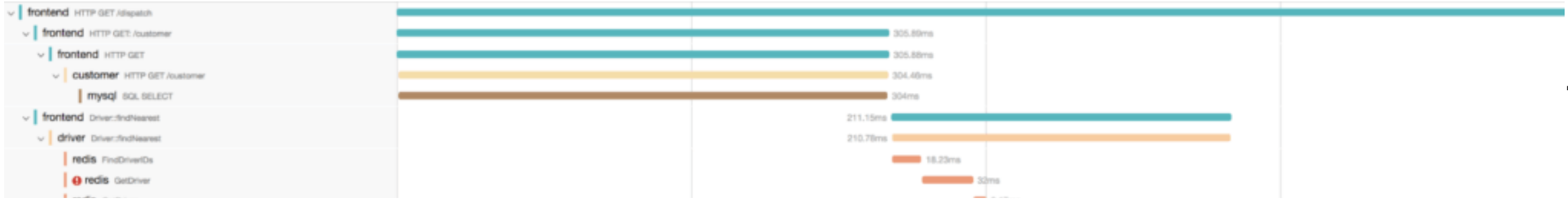
● 可观测性的三大支柱



Metrics

November 27th 2018, 15:37:10.992 elastic opbeans LOG duration: 0.435 ms bind S_14: select product1_.id as col_0_0_, product1_.sku as col_1_0_, product1_.name as col_2_0_, product1_.stock as col_3_0_, sum(orderline0_.amount) as col_4_0_ from order_lines orderline0_ left outer join products product1_ on

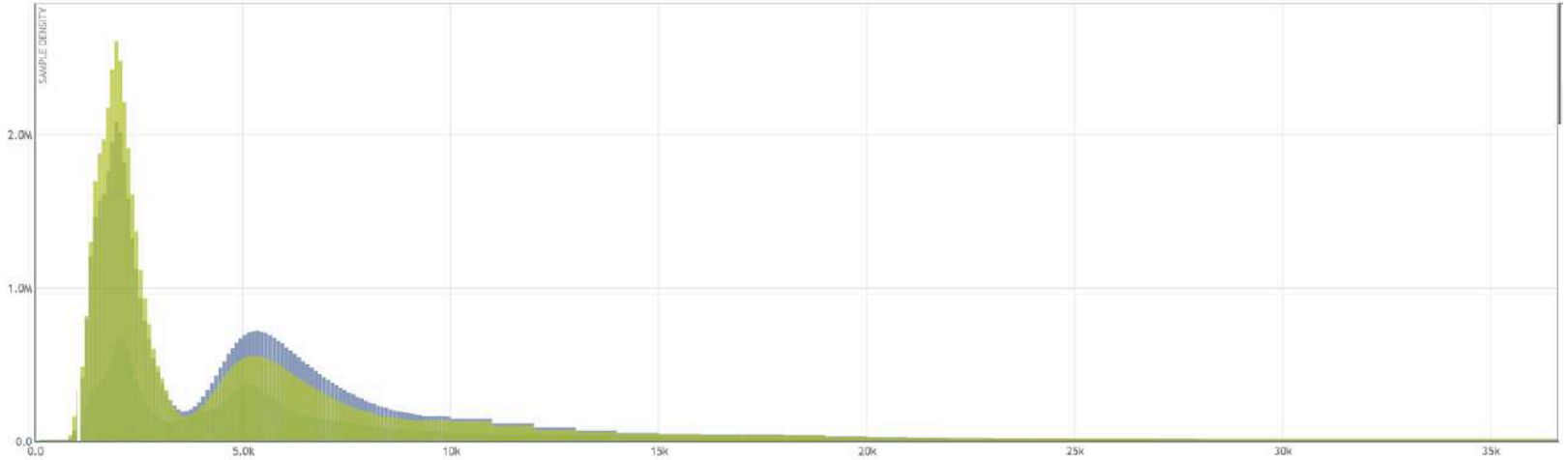
Logs



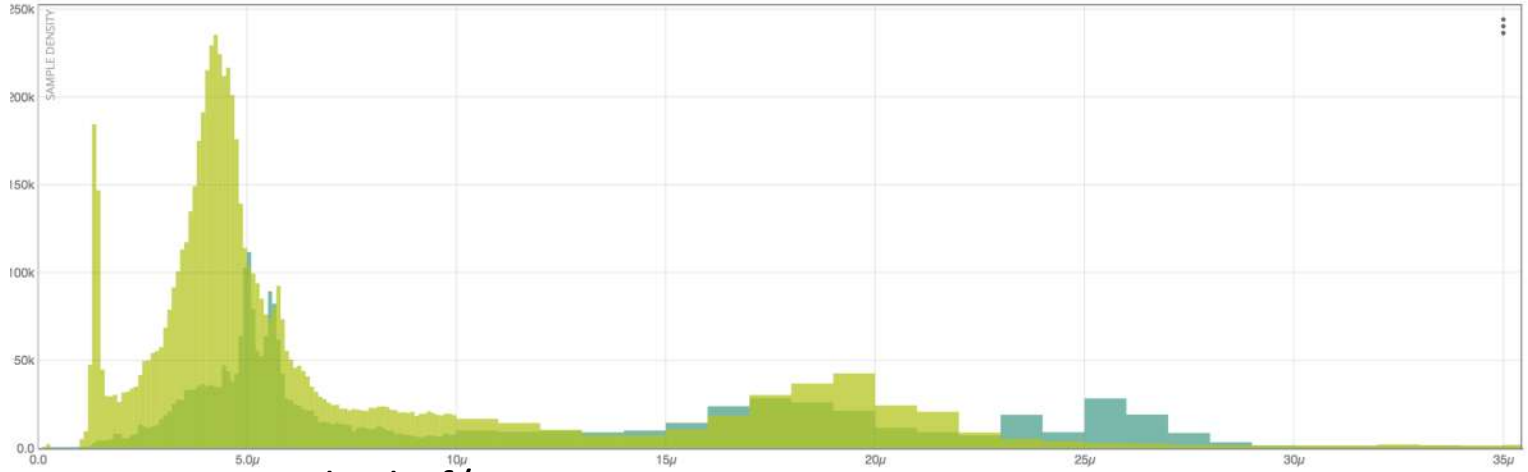
Trace

● 当前可观测性的局限性

- **Aggregated or coarse-grained data， lack of deep fine—grained data (Depth limitation);**



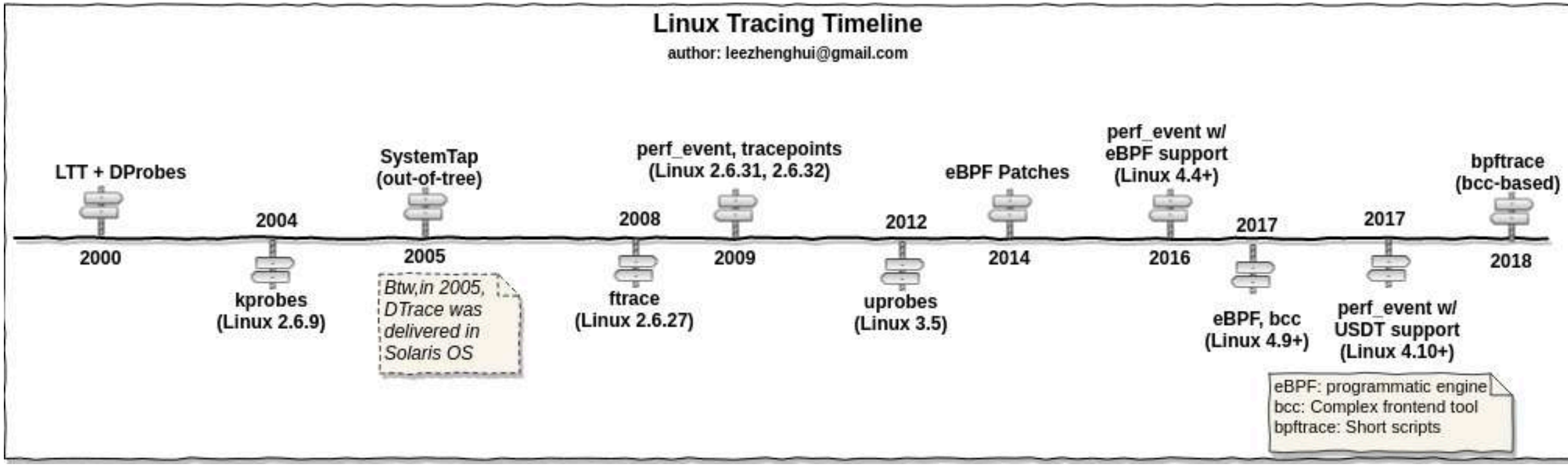
Aggregated IO latency



https://www.circonus.com/2018/05/linux-system-monitoring-with-ebpf/ System call “Write” & “Read”

● eBPF应用

➢基于Tracing的可观测性

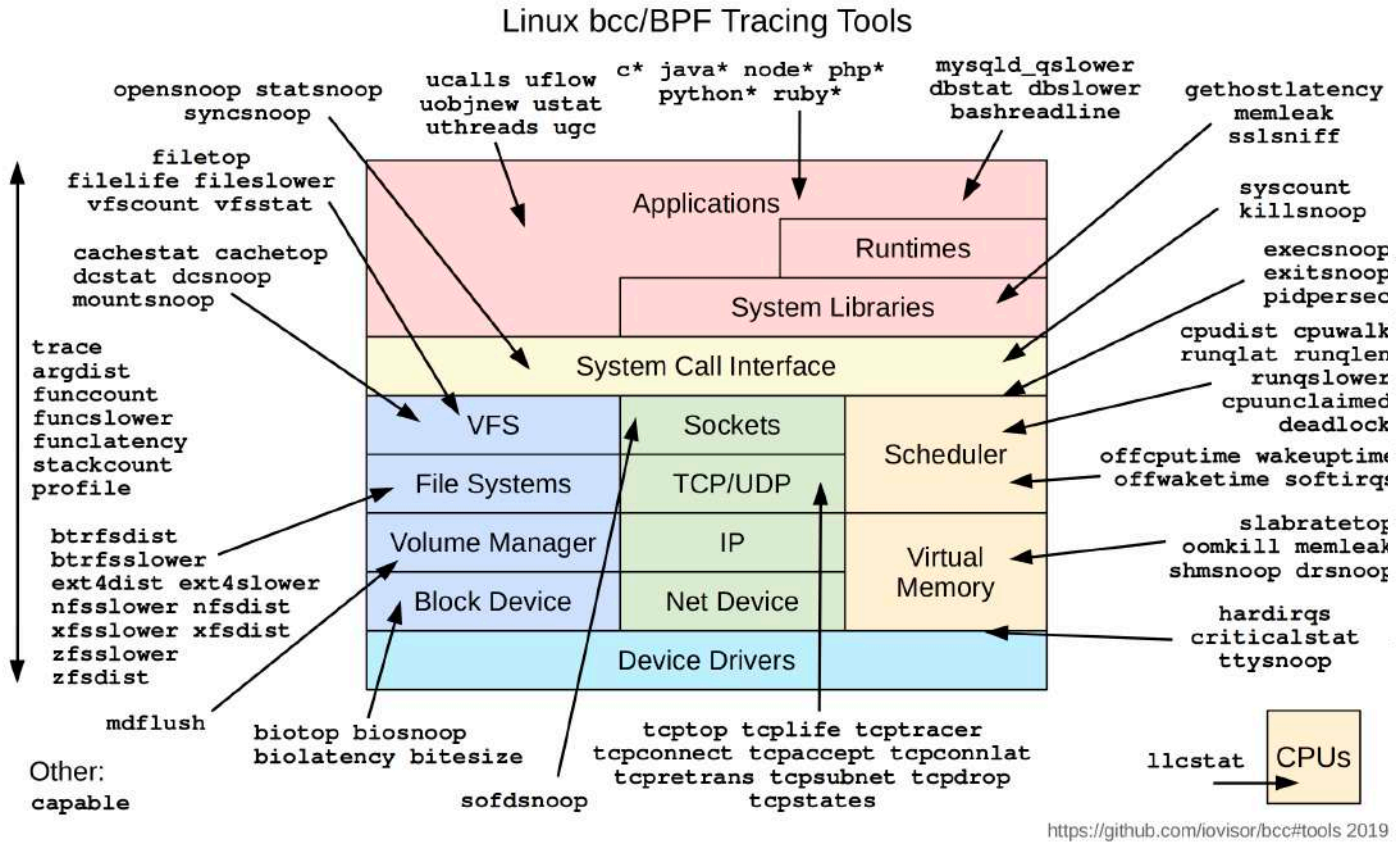


Linux tracing的发展过程

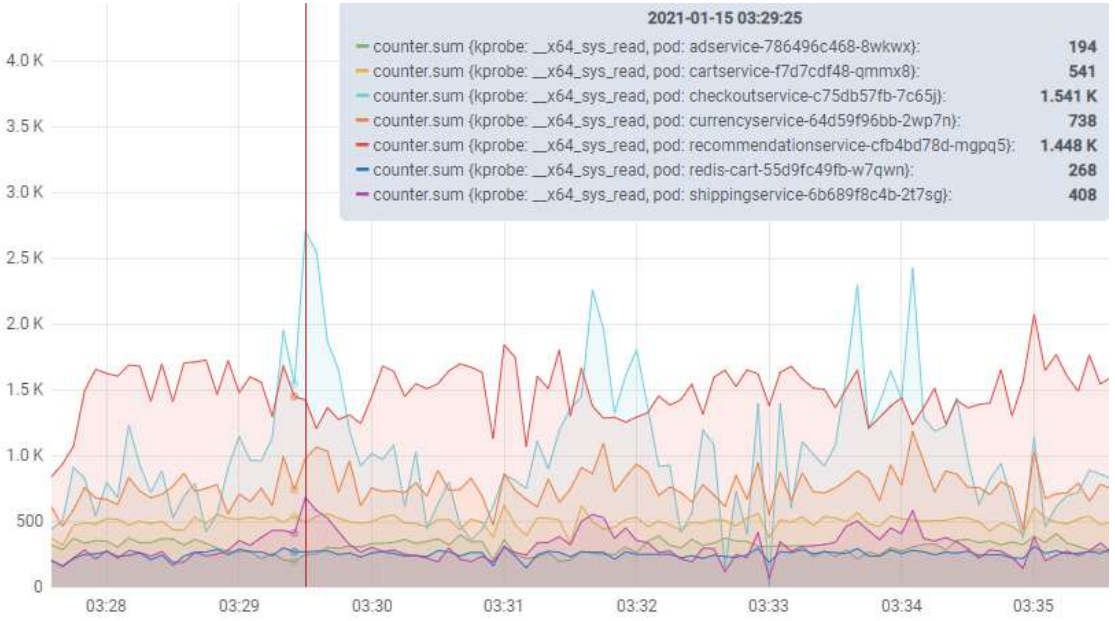
● eBPF应用

➢可观测性—指标

eBPF除了常规的指标监控如CPU、内存等，还可以监控细粒度的系统调用等信息，通过Kprobe或者Tracepoint实现；



eBPF追踪点

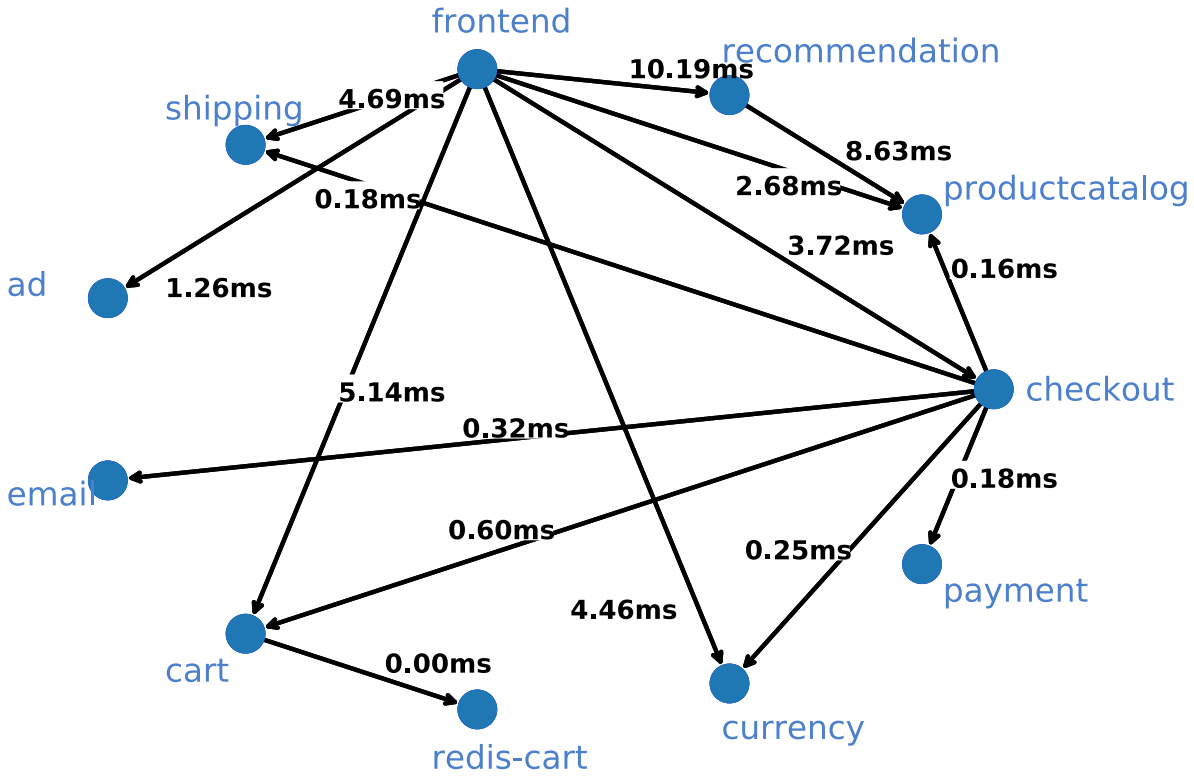


系统调用数量

● eBPF应用

➢可观测性—拓扑

通过拦截sock相关的send/recv操作，解析协议头，获得进程之间的调用关系，可进一步关联Kubernetes原信息，获得容器、服务之间的调用关系；



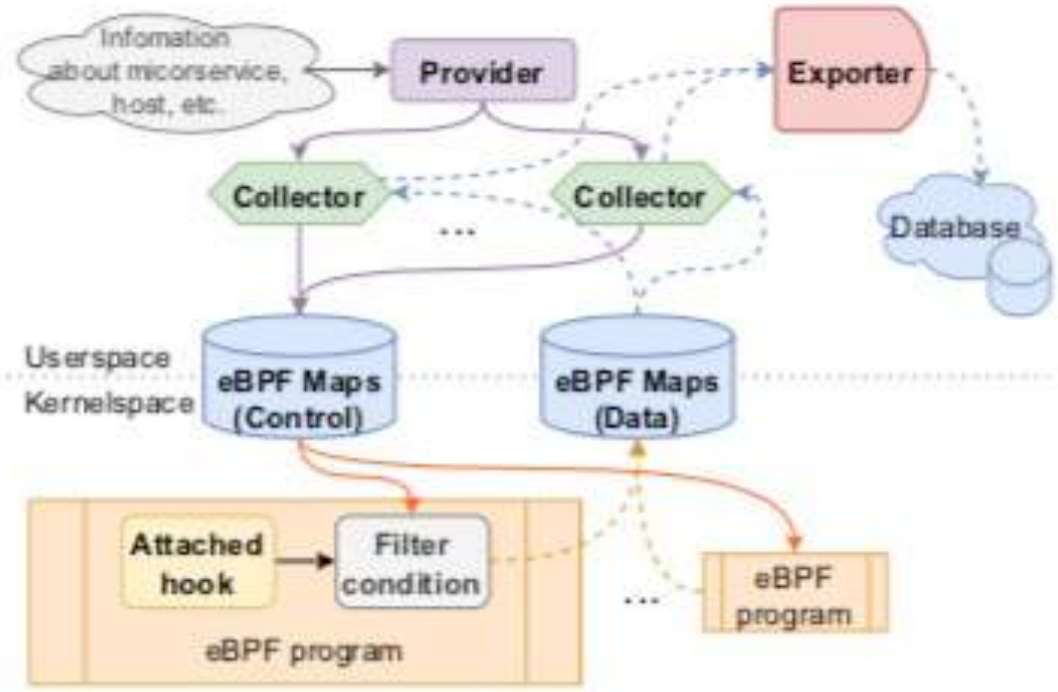
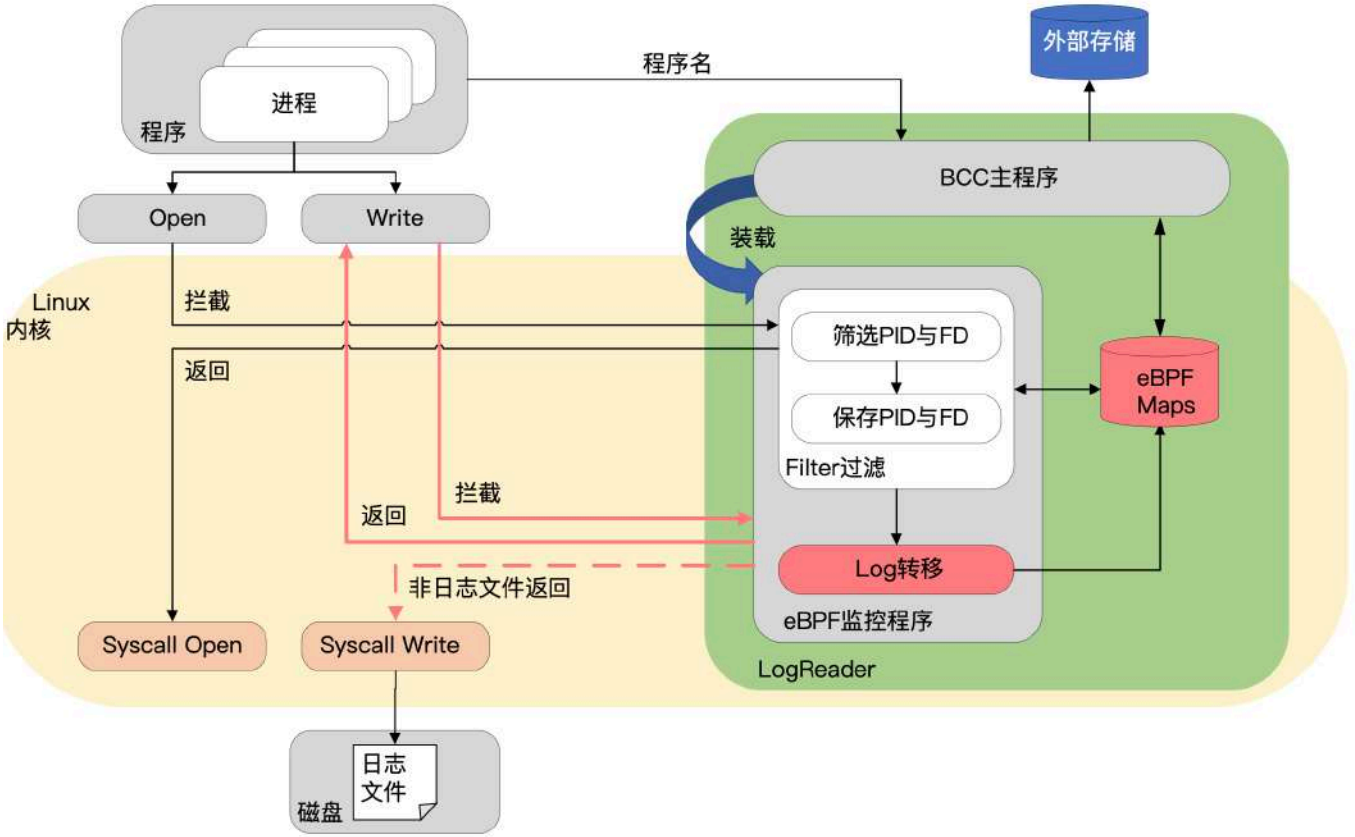


Fig. 2. The Architecture of a Kmon.

● eBPF应用

➢可观测性—日志

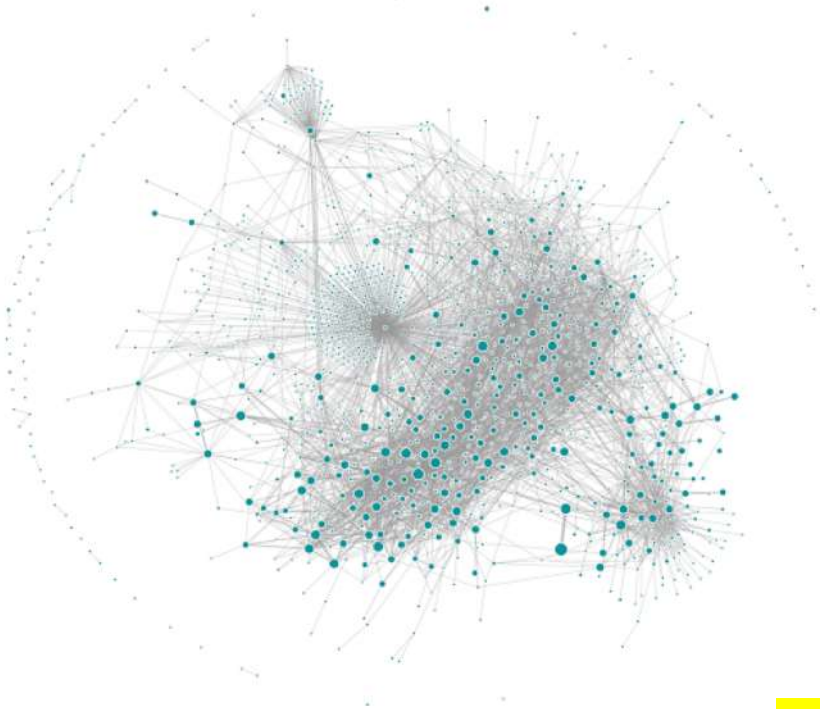
传统的日志监控系统需要写到硬盘，再集中收集，浪费I/O，吞吐量有一定的限制，利用eBPF拦截进程对文件的read/write操作获取日志信息，并通过修改write的返回值实现不落盘的日志收集。已申请专利。c



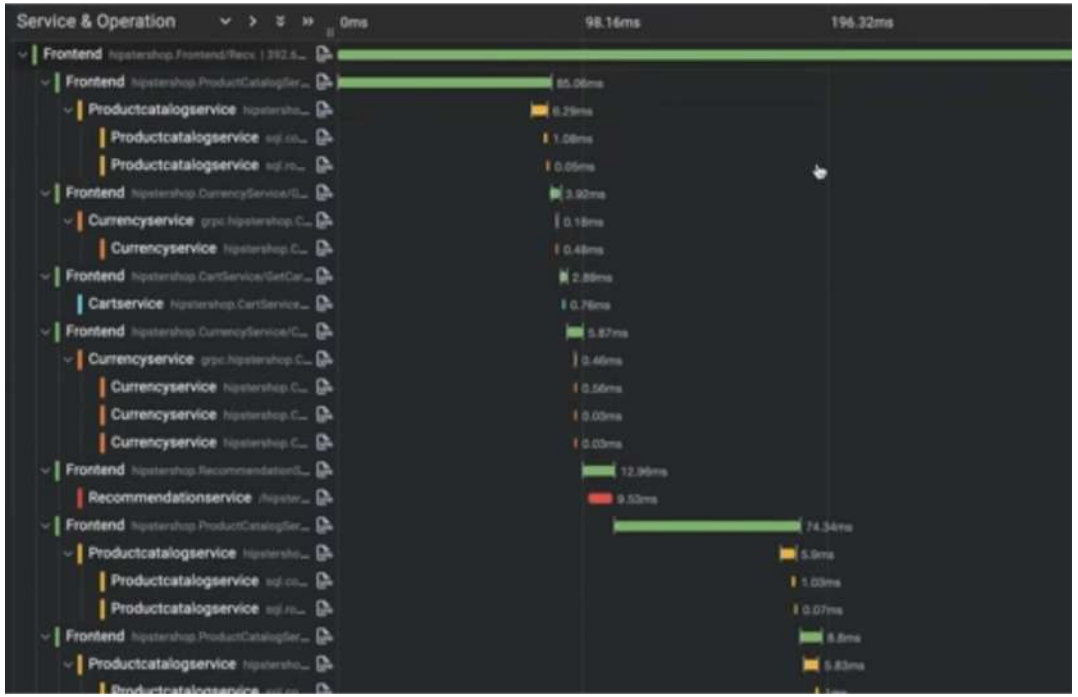
● eBPF应用

➢可观测性—调用链

— 现代微服务系统过于复杂，端到端的请求追踪已经成为缺省配置；
— 当前主流的追踪技术/系统包括：OpenTracing、OpenTelemetry、SkyWalking等；
— 对源代码有侵入性，学习周期长，且不够灵活，软件升级的带来的开销大；
— 无法实现跨语言的特性，不同语言需要提供不同的tracing库；



微服务系统



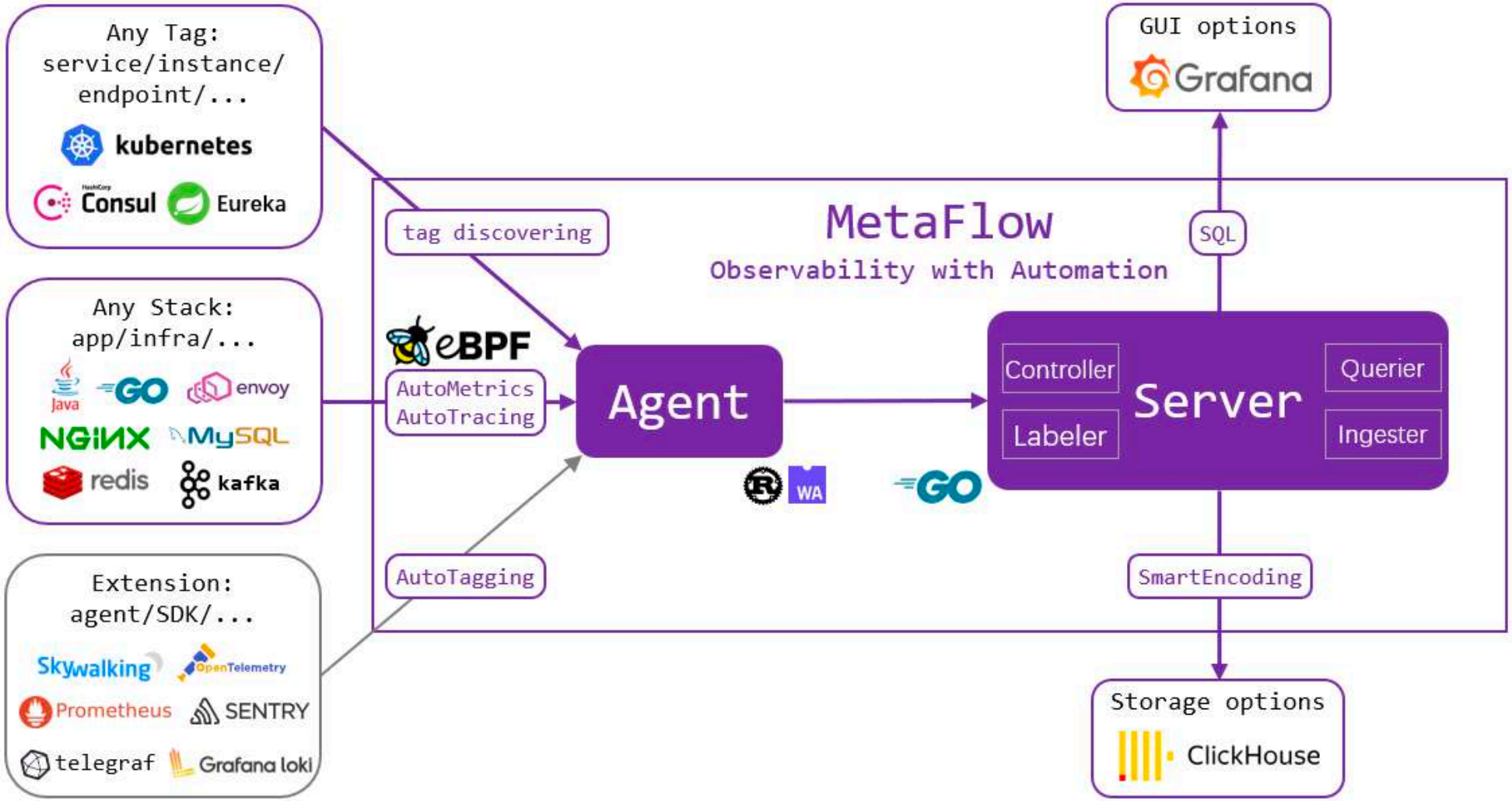
调用链实例

如何实现透明的调用链追踪？

● eBPF应用

https://github.com/metaflowys/metaflow/blob/main/README-CN.md

➢可观测性—调用链

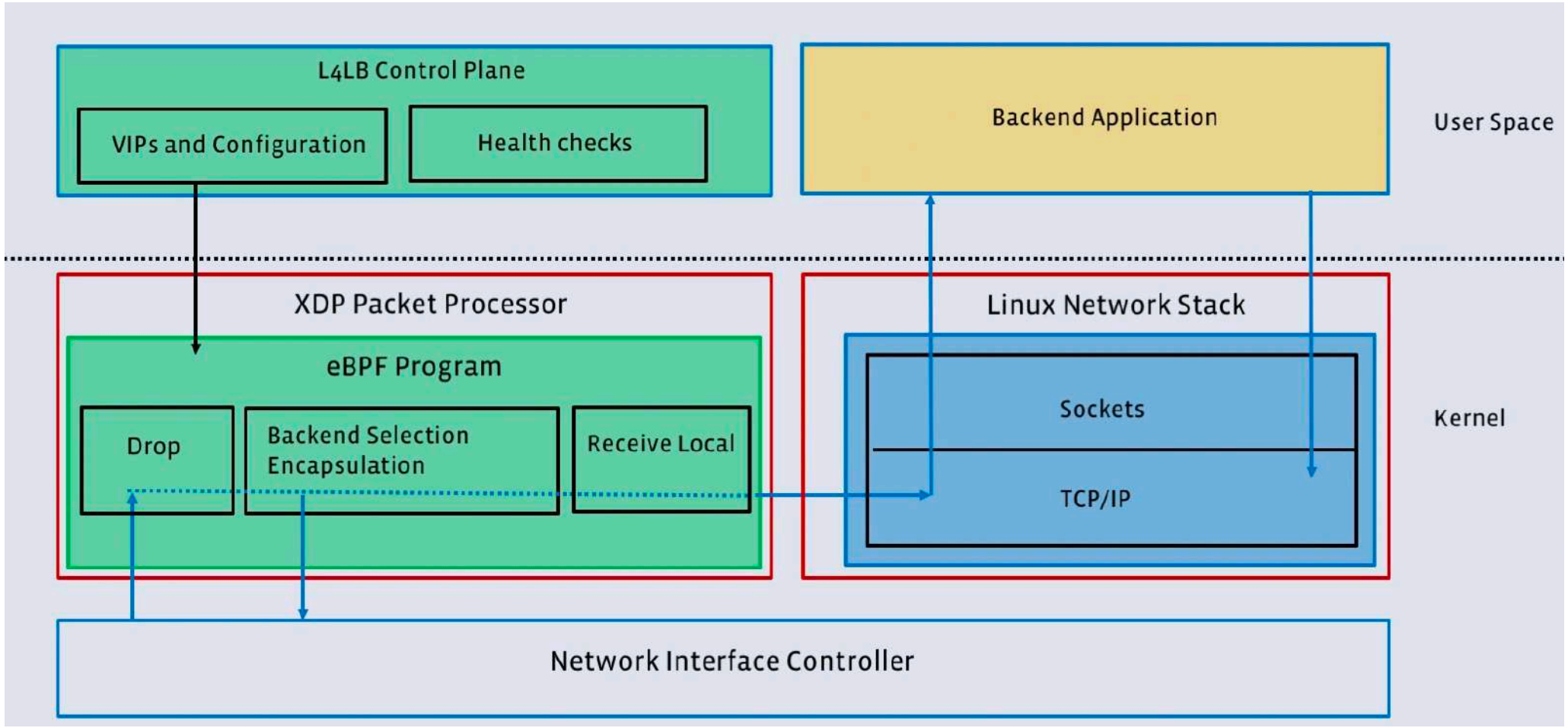


● eBPF应用

➢网络应用——负载均衡

利用eBPF 与XDP结合实现L4层的告诉负载均衡；

https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

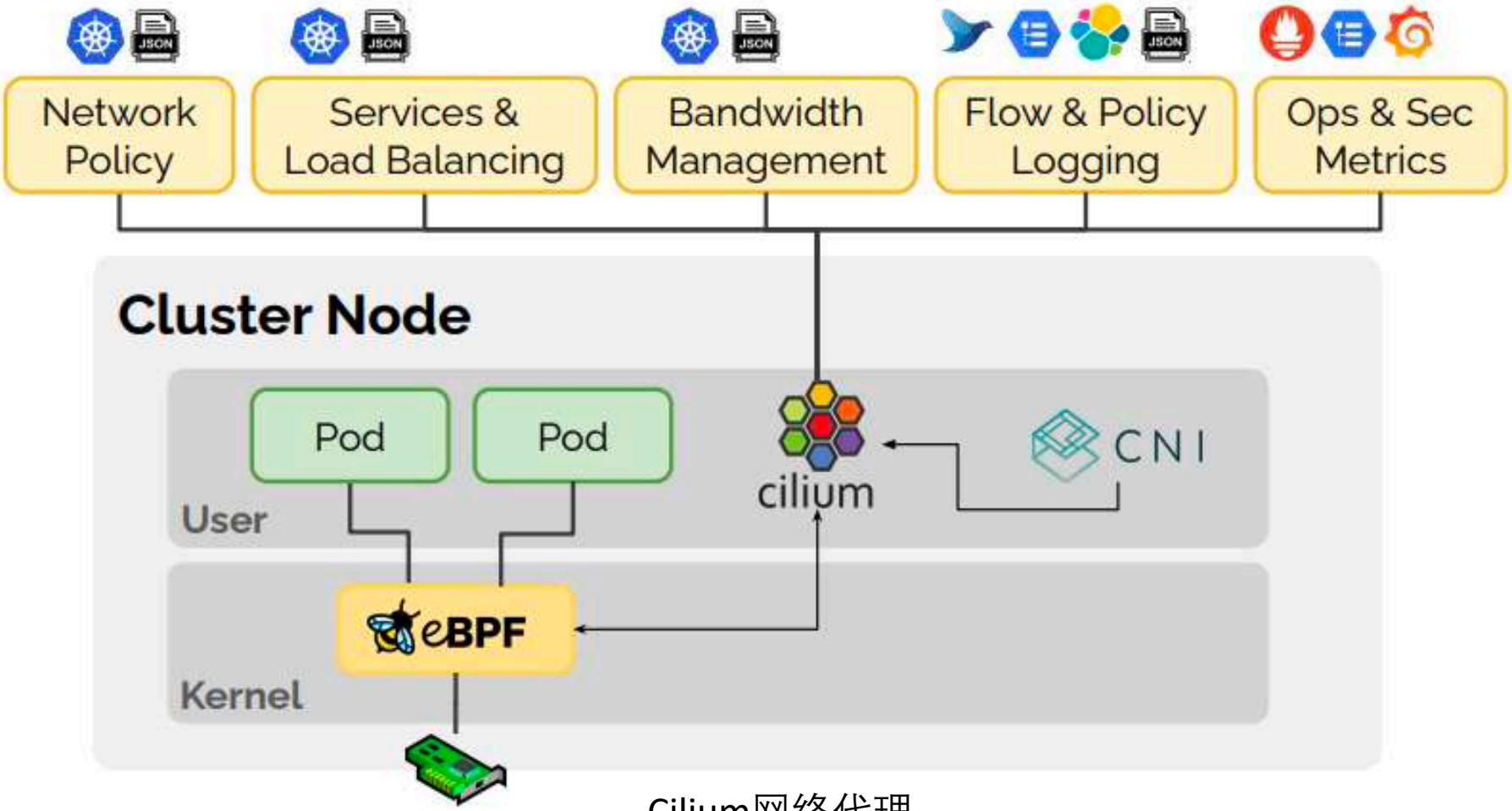


● eBPF应用

➢网络应用——负载均衡

https://github.com/cilium/cilium

利用eBPF 与XDP结合实现L4层的告诉负载均衡；



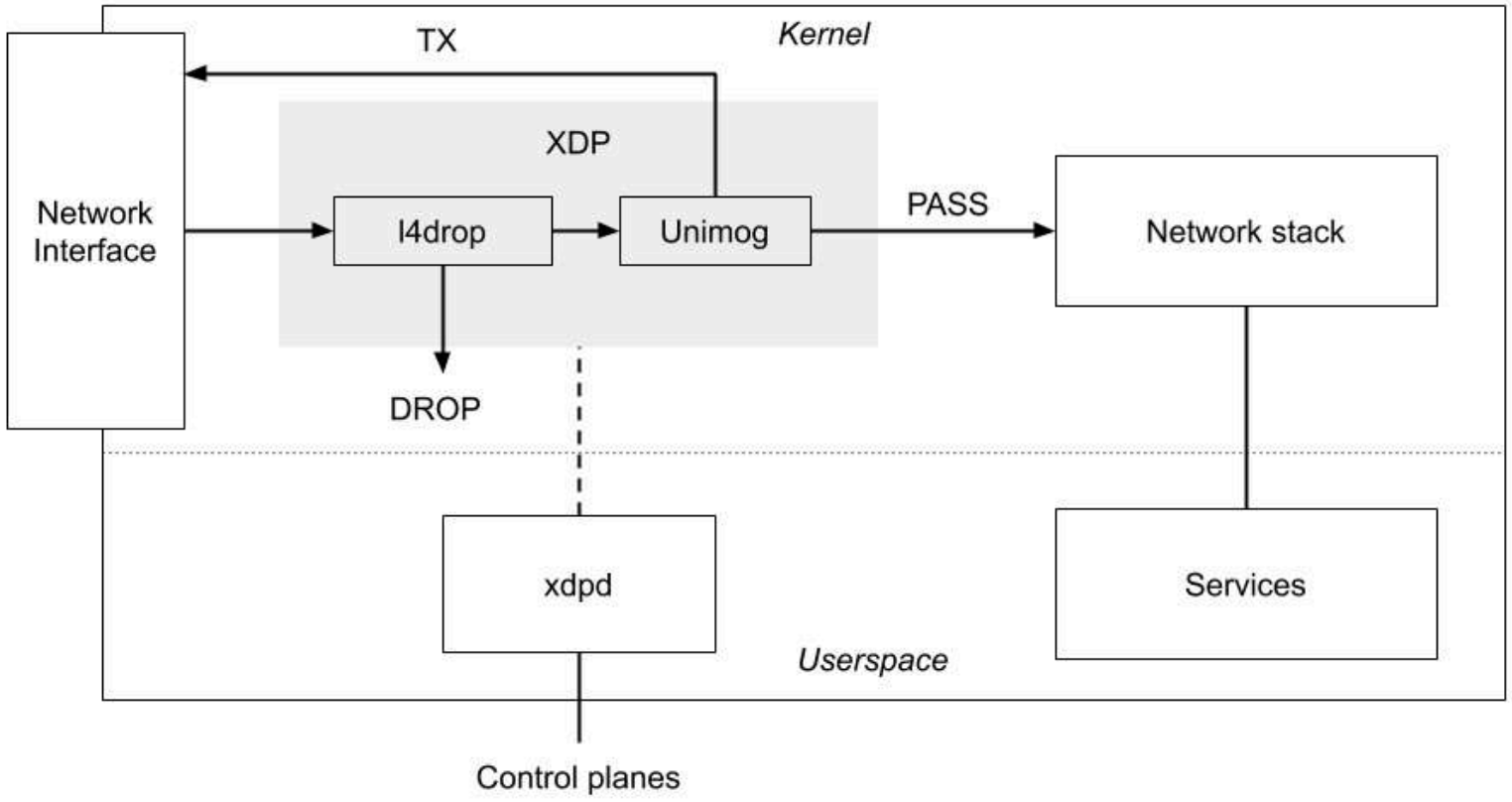
Cilium网络代理

● eBPF应用

➢网络应用——负载均衡

利用eBPF 与XDP结合实现L4层的告诉负载均衡；

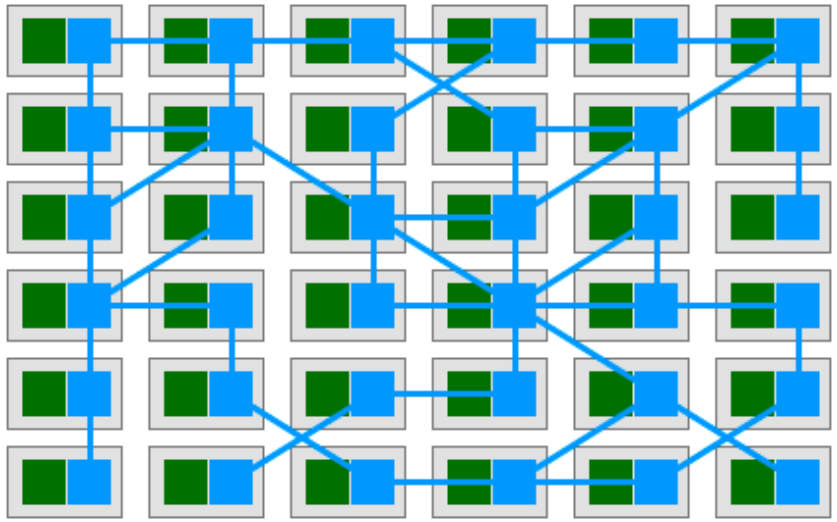
https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/



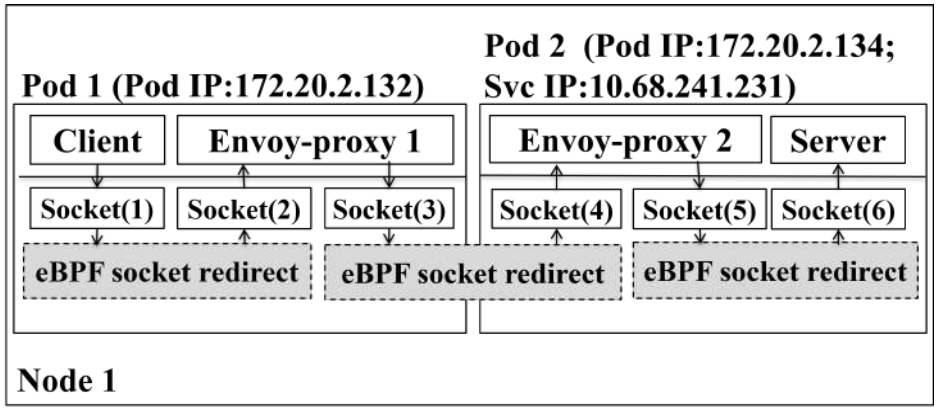
Cloudflare’s edge load balancer

● eBPF应用

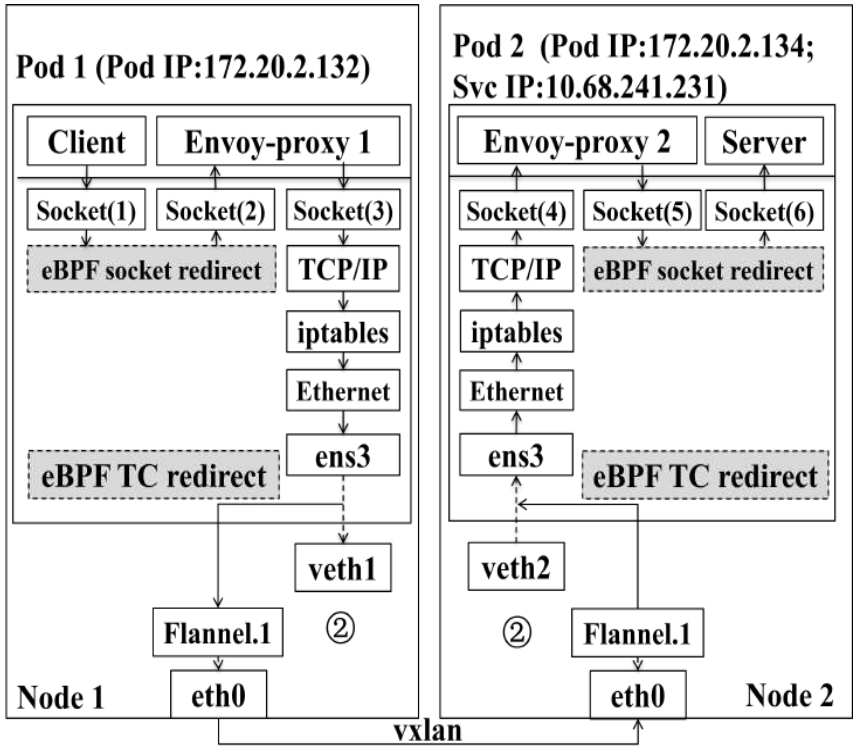
➢网络应用——网络优化



服务网格

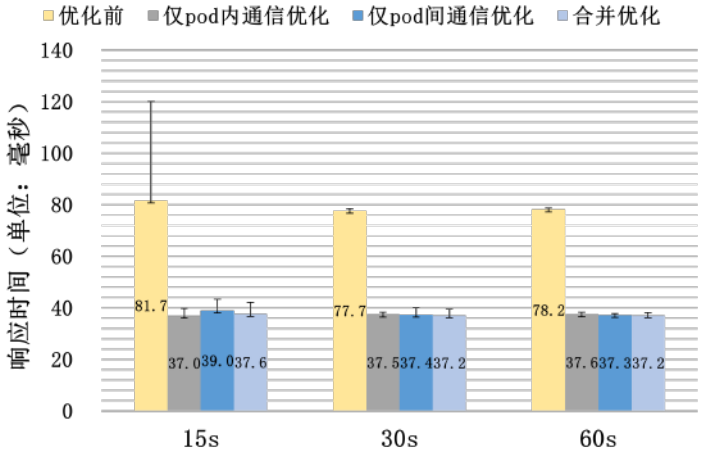


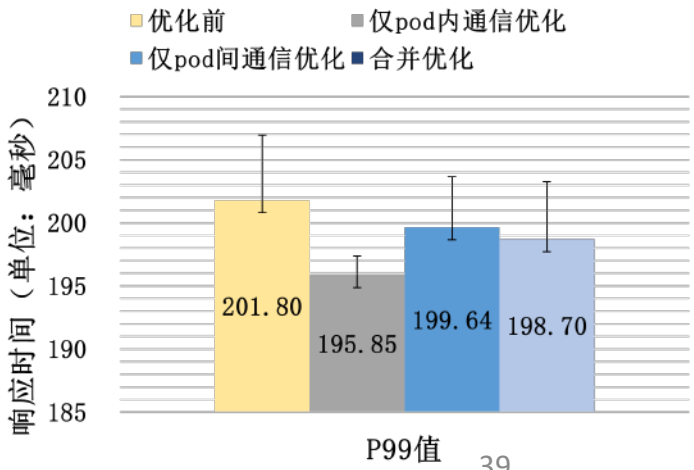
(a) 两个 pod 部署在同一个节点上



(b) 两个 pod 分别部署在不同的节点上

优化方法



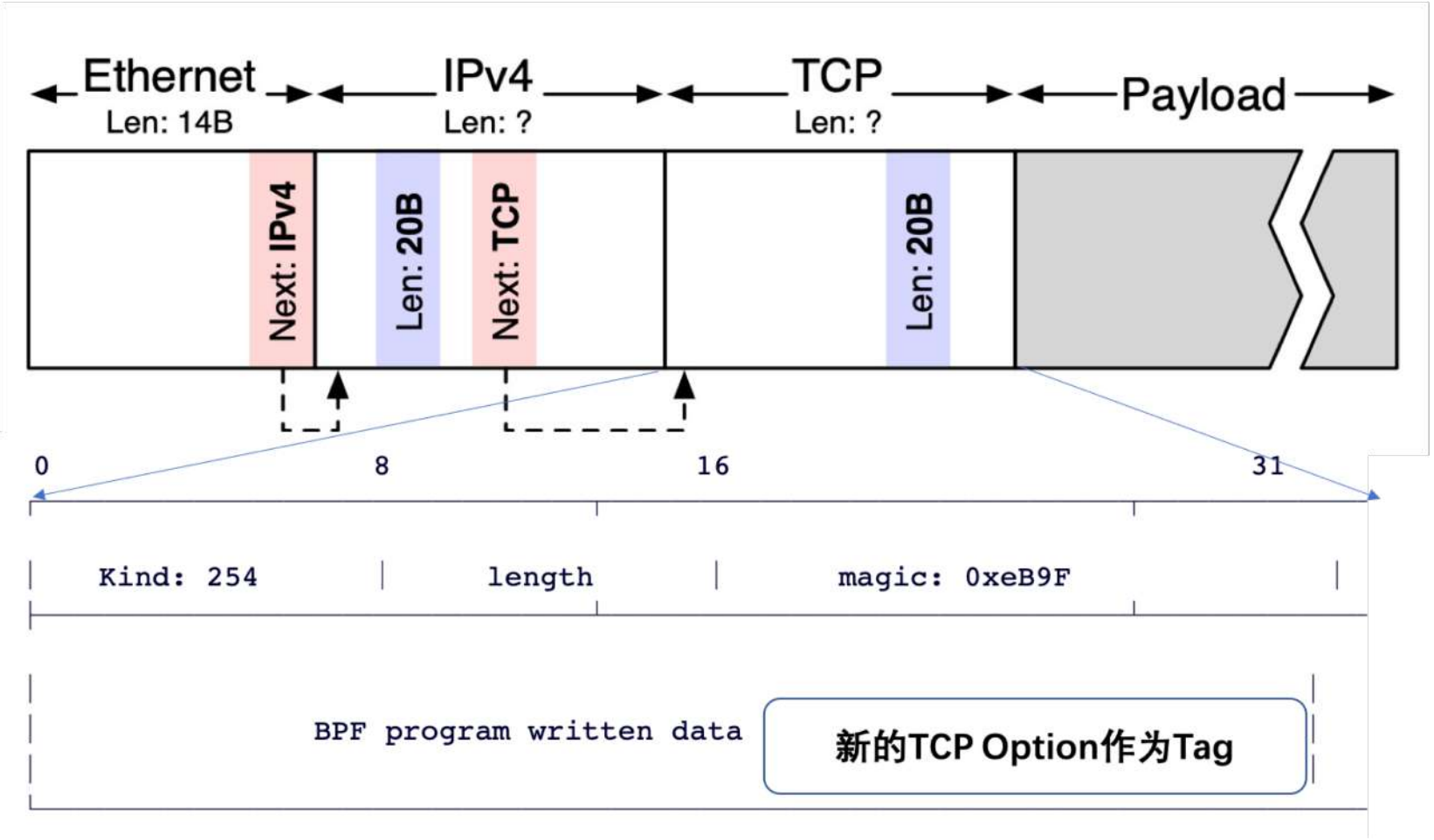


优化结果

● eBPF应用

➢网络应用——网络包重写

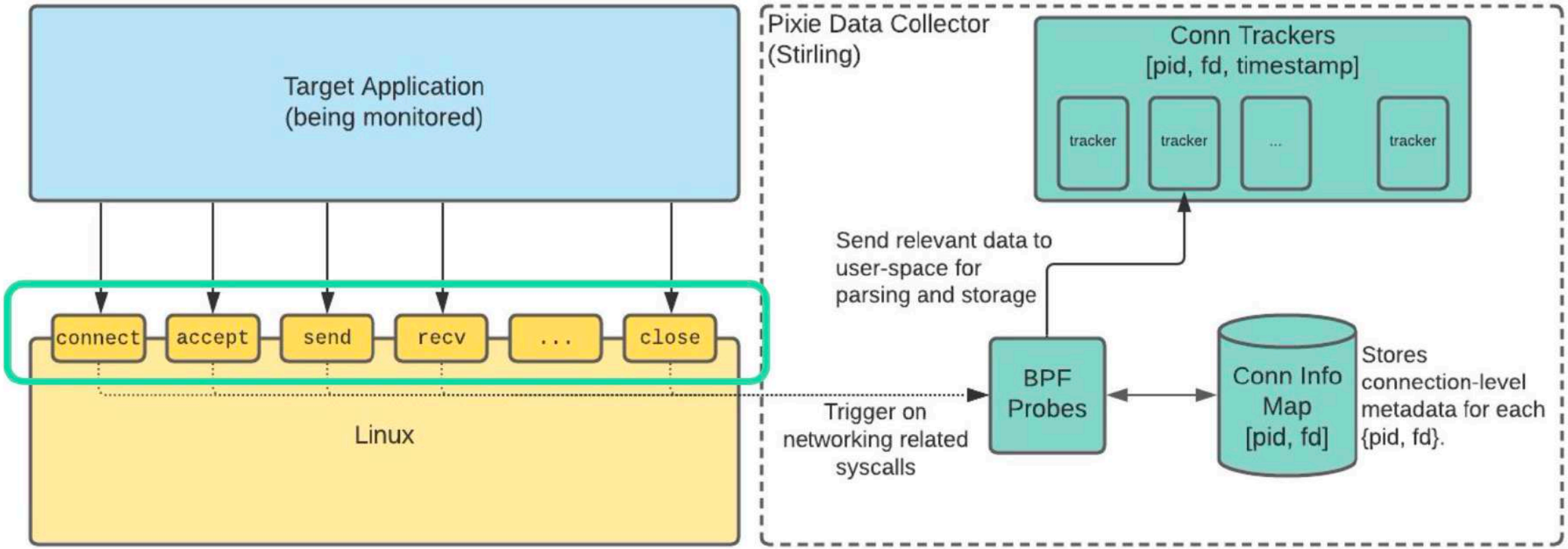
eBPF提供了可以重写TCP Option的help函数，实现对网络数据包的修改和操纵；



● eBPF应用

➢网络应用——协议解析

eBPF拦截socket，解析payload，分析出具体的L7层的应用协议如Http、mysql、mongo、gRPC，可实现APM的功能；

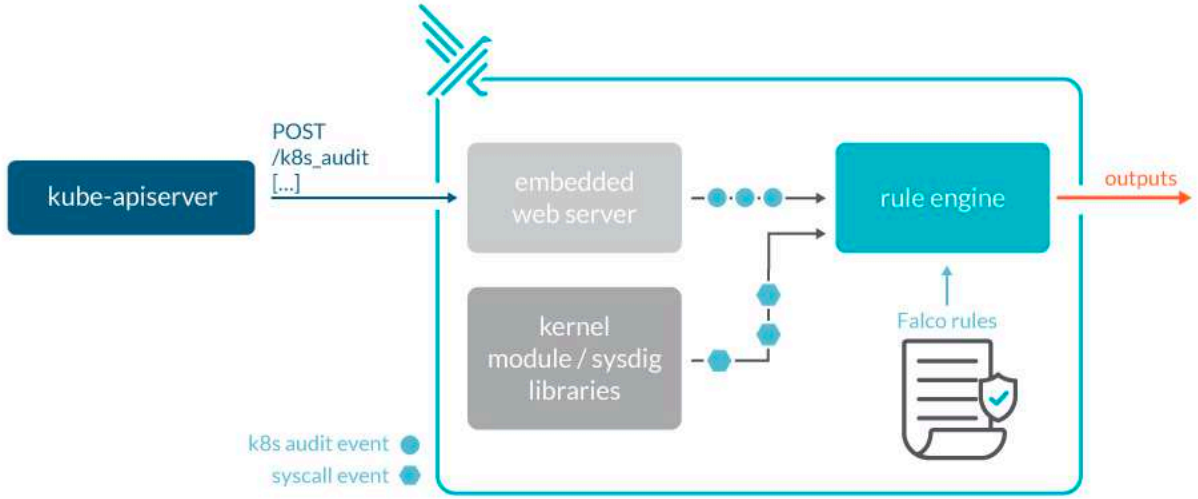


● eBPF应用

➢安全

eBPF + LSM （Linux Security Module）实现安全的访问控制, 将eBPF挂载到LSM上；

```
import os
import sys
import time

from bcc import BPF, libbcc

src = """
#include <linux/fs.h>
#include <uapi/asm-generic/errno-base.h>

LSM_PROBE(file_open, struct file *file) {
    bpf_trace_printk("LSM hook: file_open\\n");

    u32 pid = bpf_get_current_pid_tgid();
    if (pid != 1) {
        bpf_trace_printk("LSM hook: file_open: Denied\\n");
        return -EPERM;
    }
    bpf_trace_printk("LSM hook: file_open: Allowed\\n");
    return 0;
}
"""
```

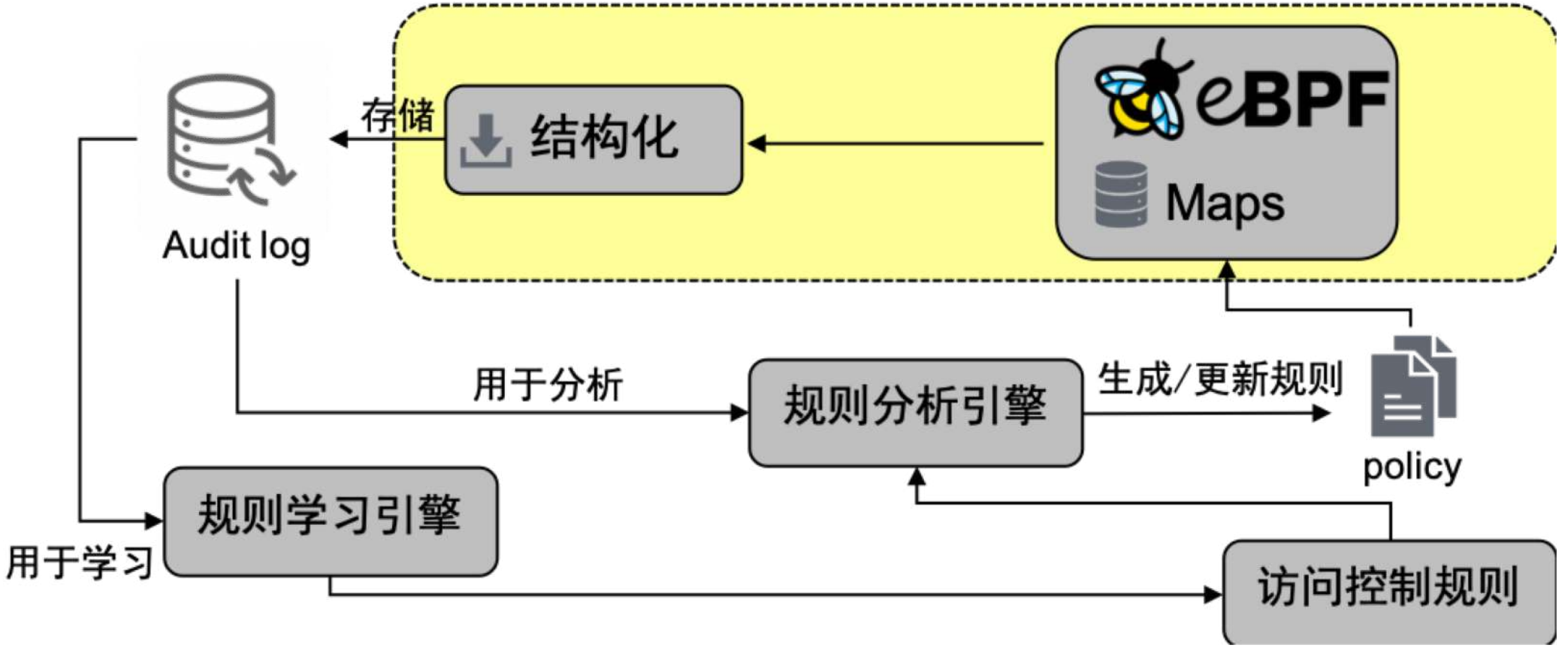



```
# ActiveMQ ports
- macro: activemq_cluster_port
  condition: fd.sport=61616
- macro: activemq_web_port
  condition: fd.sport=8161
- macro: activemq_port
  condition: activemq_web_port or activemq_cluster_port
```

● eBPF应用

➢安全

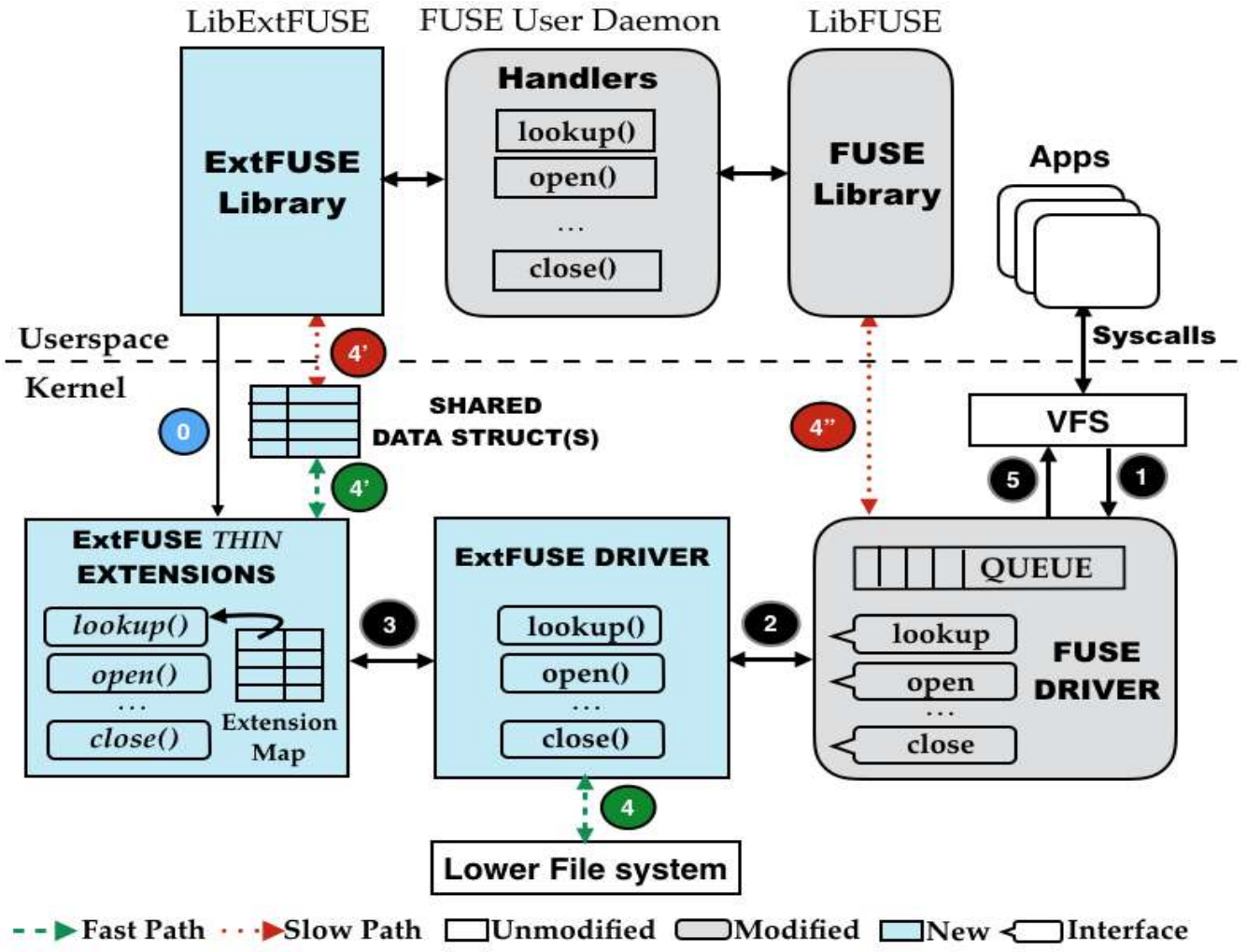
eBPF + LSM （Linux Security Module）实现安全的访问控制, 将eBPF挂载到LSM上；



● eBPF应用

➢系统优化
利用eBPF实现内核的数据缓存和计算；

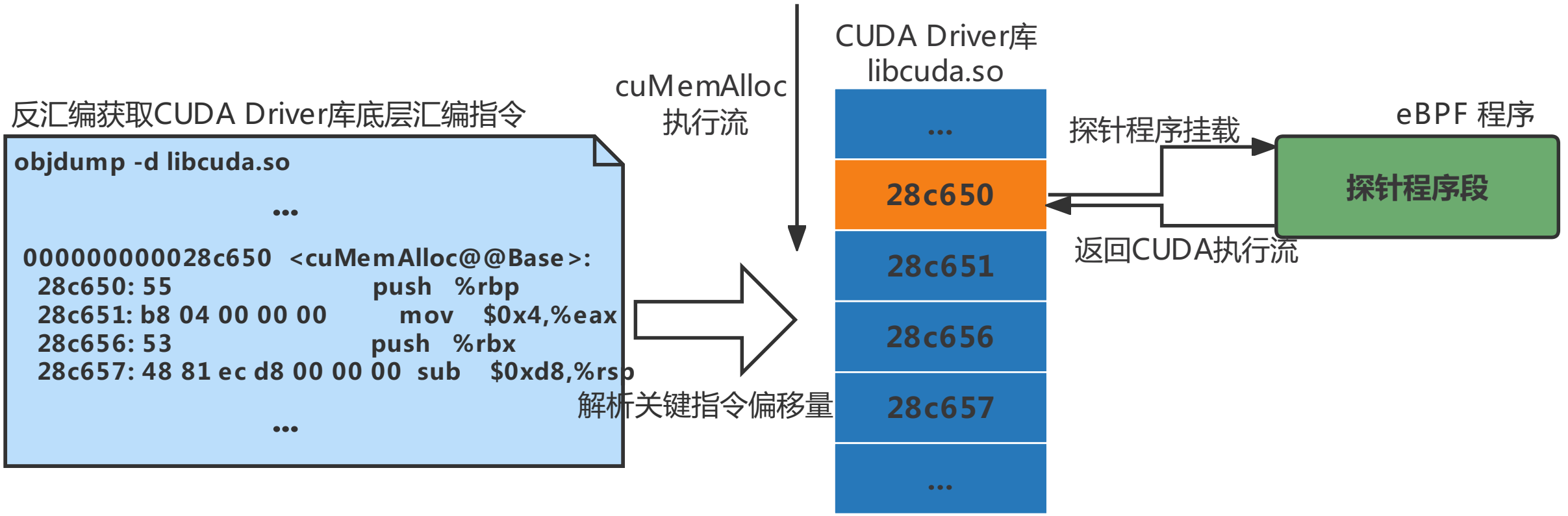
Extension Framework for File Systems in User space



● eBPF应用

➢系统优化

利用eBPF实现GPU虚拟化；AI应用对GPU的需求增加，需要增加平台的吞吐量



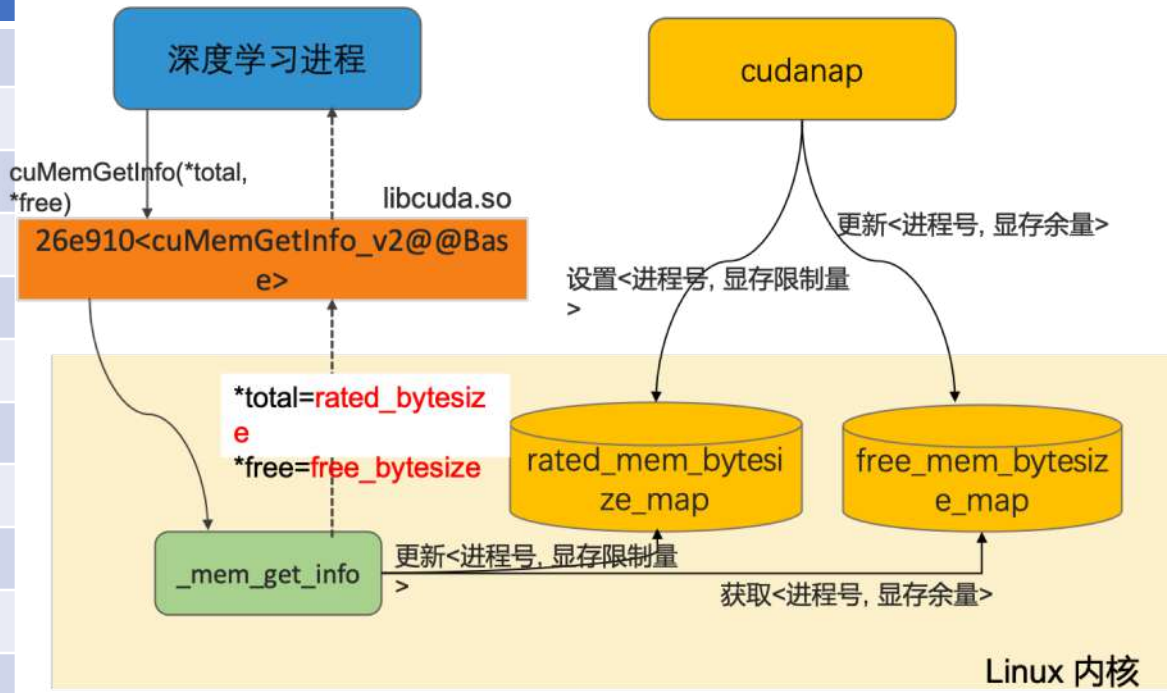
● eBPF应用

➢系统优化

利用eBPF实现GPU虚拟化、隔离和限制，需要增加平台的吞吐量；

	CUDA Driver API	描述
cdua初始化	cuInit	初始化
显存分配相关	cuMemAlloc	分配显存
	cuMemAllocPitch	分配显存
	cuArrayCreate	创建矩阵
	cuArray3DCreate	创建矩阵
	cuMipmappedArrayCreate	创建矩阵
	cuFree	显存释放
算力分配相关	cuLaunch	启动核函数
	cuLaunchKernel	启动核函数
	cuLaunchCooperativeKernel	启动核函数
	cuLaunchGrid	启动核函数
设备信息相关	cuMemGetInfo	获取显存总量和空余显存
	cuDeviceTotalMem	获取显存总量

CUDA API拦截



CUDA API限制

4. eBPF展望



● eBPF展望

- eBPF与硬件如FPGA的结合，加速eBPF指令的执行；
- eBPF与P4编程语言的结合，增加软件定义网络的可编程能力；
- eBPF程序的形式化与高效率验证问题；
- eBPF在网络传输性能优化的应用；
- eBPF的开放函数的边界问题，平衡功能与安全性；
- 基于eBPF的高性能应用层协议解析；
- 基于eBPF的系统安全策略的生成；
- 基于eBPF的云原生系统的治理；



谢谢！