# 浅谈 eBPF 数据传输之 perfbuf 与 ringbuf

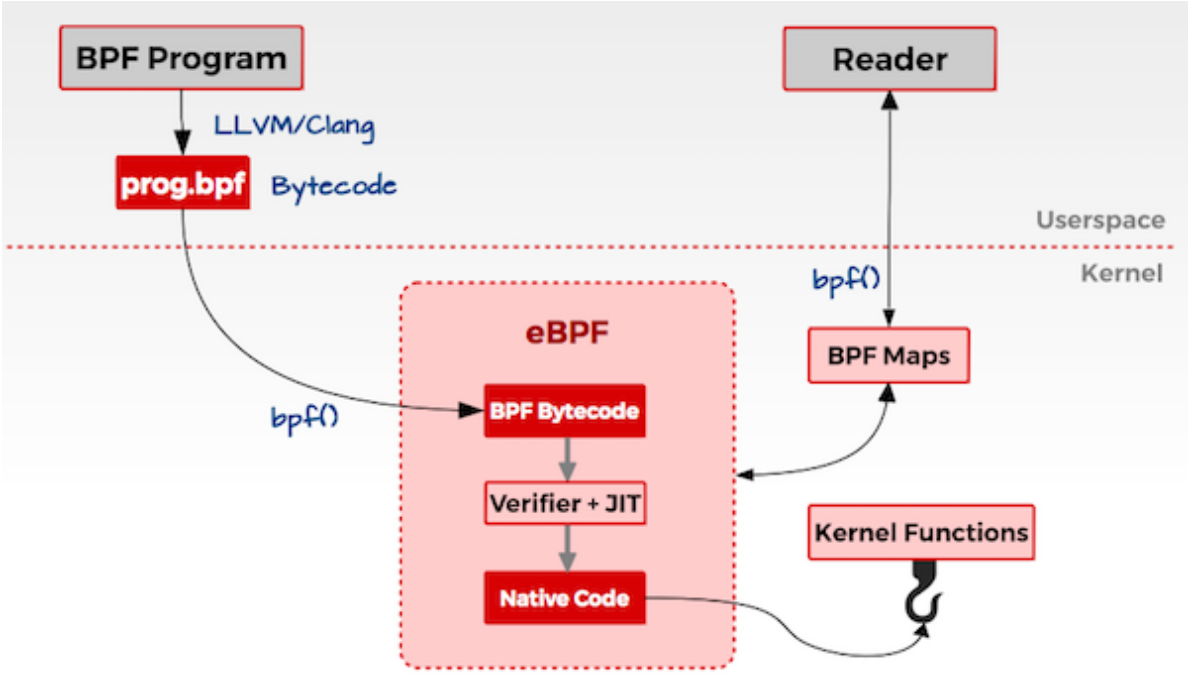
> 最近在学习 eBPF 下相关代码，对于数据传输部分有一些疑惑，在本篇文章进行跟进。直接参考 man bpf。本片文章大部分内容基于 nakryiko's blog 的文章，大部分能找到的资料均和这个 blog 相关~

## 前置内容

> 简单介绍一下 eBPF 以及 BPF maps

eBPF 可以简单的理解为：在内核中运行沙箱程序，其优势在于安全性（有 verifier 等机制）以及高效率等。程序中能使用到的数据类型与函数，会随着 kernel 版本不断丰富增强。对于k8s 化的今天，越来越多监测、加速等应用使用了 eBPF。

借用一张图表示一下：




其中 BPF Maps 为内核态 BPF 程序和用户态程序交互的数据桥梁。在内核中，对于创建映射的设置，在 bpf.h 中定义，如下所示：

```
union bpf_attr {
    struct { /* anonymous struct used by BPF_MAP_CREATE command */
        __u32   map_type;   /* one of enum bpf_map_type */
        __u32   key_size;   /* size of key in bytes */
        __u32   value_size; /* size of value in bytes */
        __u32   max_entries;    /* max number of entries in a map */
        __u32   map_flags;  /* BPF_MAP_CREATE related
                * flags defined above.
                */
        __u32   inner_map_fd;   /* fd pointing to the inner map */
        __u32   numa_node;  /* numa node (effective only if
                * BPF_F_NUMA_NODE is set).
                */
        char    map_name[BPF_OBJ_NAME_LEN];
        __u32   map_ifindex;    /* ifindex of netdev to create on */
        __u32   btf_fd;     /* fd pointing to a BTF type data */
```

```
        __u32   btf_key_type_id;    /* BTF type_id of the key */
        __u32   btf_value_type_id;  /* BTF type_id of the value */
        __u32   btf_vmlinux_value_type_id;/* BTF type_id of a kernel-
                        * struct stored as the
                        * map value
                        */
        /* Any per-map-type extra fields
         *
         * BPF_MAP_TYPE_BLOOM_FILTER - the lowest 4 bits indicate the
         * number of hash functions (if 0, the bloom filter will default
         * to using 5 hash functions).
         */
        __u64   map_extra;
    };
}
```

我们在预定义一个 map 时，会设置 bpf_attr，在后面的程序中会展现。

# BPF map 类型

在 bpf.h 中定义
[ 注 ]：部分文章或者参考的资料（例如《linux内核观测技术.pdf》）时间较早，最新的需要去 kernel source 下看

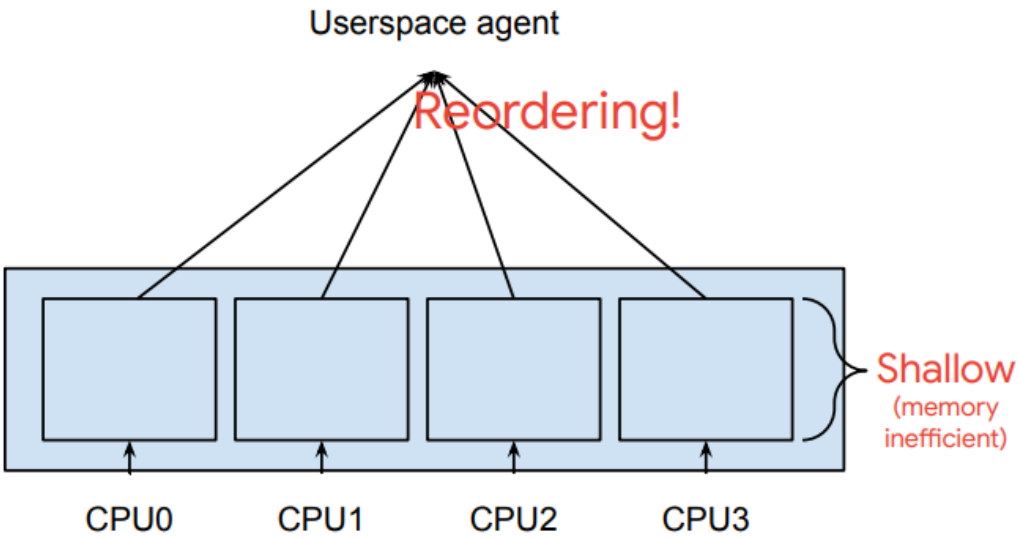
```
// 定义
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC = 0,
    BPF_MAP_TYPE_HASH = 1,
    BPF_MAP_TYPE_ARRAY = 2,
    BPF_MAP_TYPE_PROG_ARRAY = 3,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY = 4, // linux kernel 4.4
    BPF_MAP_TYPE_PERCPU_HASH = 5,
    BPF_MAP_TYPE_PERCPU_ARRAY = 6,
    BPF_MAP_TYPE_STACK_TRACE = 7,
    BPF_MAP_TYPE_CGROUP_ARRAY = 8,
    BPF_MAP_TYPE_LRU_HASH = 9,
    BPF_MAP_TYPE_LRU_PERCPU_HASH = 10,
    BPF_MAP_TYPE_LPM_TRIE = 11,
    BPF_MAP_TYPE_ARRAY_OF_MAPS = 12,
    BPF_MAP_TYPE_HASH_OF_MAPS = 13,
    BPF_MAP_TYPE_DEVMAP = 14,
    BPF_MAP_TYPE_SOCKMAP = 15,
    BPF_MAP_TYPE_CPUMAP = 16,
    BPF_MAP_TYPE_XSKMAP = 17,
    BPF_MAP_TYPE_SOCKHASH = 18,
    BPF_MAP_TYPE_CGROUP_STORAGE = 19,
    BPF_MAP_TYPE_REUSEPORT_SOCKARRAY = 20,
    BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE = 21,
    BPF_MAP_TYPE_QUEUE = 22,
    BPF_MAP_TYPE_STACK = 23,
    BPF_MAP_TYPE_SK_STORAGE = 24,
    BPF_MAP_TYPE_DEVMAP_HASH = 25,
    BPF_MAP_TYPE_STRUCT_OPS = 26,
    BPF_MAP_TYPE_RINGBUF = 27, // linux kernel 5.8
    BPF_MAP_TYPE_INODE_STORAGE = 28,
};
```

BPF目前有29种map，许多地方在实际使用的时候，需要我们注意，由于我在用 go 做开发，有很多地方也需要社区组件的支持。本片文章我们重点关注，PERF_EVENT_ARRAY & RINGBUF

# Perf Buffer

> 参考《linux内核观测技术.pdf》

借用一下 google 的 ppt 中的图片




perf buffer 可以看作是每个 CPU 上的一个 buffer 的集合，也是我们在 eBPF 程序中几乎使用最多的一种映射类型。我们通过 perf buffer 来实现内核 eBPF 程序和用户态数据交互。

下面是一个 tracepoint 的例子，先看一下 bpf 程序：

```
// 预定义，SEC("maps") 宏告诉内核该结构是 BPF 映射，并创建相应的映射
struct bpf_map_def SEC("maps") perf_events = {
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(u32), // 代表的是对应的 fd
    // .max_entries = 512, // 最大 fd 数量，[注]: 这里 fd 在使用 libbpf 的时候可以不用设置，因为会默认设置成最大 CPU 数
    // 对于这个地方其实很有意思，我看了一下 issue 在 5 月 cilium/ebpf 还在issue 里有讨论过这个问题，结论是
    // So we should do the min(max_elems, CPUs) logic.
    // https://github.com/cilium/ebpf/pull/300
    // https://github.com/cilium/ebpf/issues/209
    // 总之设定成 CPU 最大数，保证 a buffer per cpu，或者设置一个大值，在 userspace 的 go 程序里设置成 MAX_CPU
    // 对应的代码在 https://github.com/cilium/ebpf/blob/02ebf28c2b0cd7c2c6aaf56031bc54f4684c5850/map.go 的函数 clampPerfEventArraySize() 里面
};

SEC("tracepoint/syscalls/sys_enter_execve")
int enter_execve(struct execve_entry_args_t *ctx)
{
    struct enter_execve_t enter_execve_data = {};
    enter_execve_data.type = 1;
    u64 id = bpf_get_current_uid_gid();
    enter_execve_data.uid = id;
```

```
    enter_execve_data.gid = id >> 32;
    id = bpf_get_current_pid_tgid();
    enter_execve_data.pid = id;
    enter_execve_data.tgid = id >> 32;
    struct task_struct *task;
    struct task_struct* real_parent_task;
    task = (struct task_struct*)bpf_get_current_task();
    bpf_get_current_comm(&enter_execve_data.comm,
sizeof(enter_execve_data.comm));
    bpf_probe_read(&real_parent_task, sizeof(real_parent_task), &task-
>real_parent );
    bpf_probe_read(&enter_execve_data.ppid, sizeof(enter_execve_data.ppid),
&real_parent_task->pid );
    bpf_probe_read_str(enter_execve_data.filename,
sizeof(enter_execve_data.filename), ctx->filename);
    const char* argp = NULL;

    for (__s32 i = 0; i < DEFAULT_MAXARGS; i++)
    {
        bpf_probe_read(&argp, sizeof(argp), &ctx->argv[i]);
        if (!argp) {
            return 0;
        }
        enter_execve_data.argsize = bpf_probe_read_str(enter_execve_data.args,
ARGSIZE, argp);
        // perf 数据上传
        bpf_perf_event_output(ctx, &perf_events, BPF_F_CURRENT_CPU,
&enter_execve_data, sizeof(enter_execve_data));
    }
    return 0;
}

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

我们重点看一下和 perf_event 相关的地方，一个是定义，一个是 bpf_perf_event_output 向用户态传输，我们需要在传输的时候指定当前的 CPU（可以在读取的时候根据 CPU 区分），对应的还有 read 相关从用户态获取。
下面是对应的用户态的 Go 读取程序，依赖在 github.com/cilium/ebpf 下（注意，这里没考虑到多 CPU 下乱序的情况）：

```
...

func Tracepoint3() {
    // Load pre-compiled programs and maps into the kernel.
    objs := sysExecveObjects{}
    if err := loadSysExecveObjects(&objs, nil); err != nil {
        log.Fatalf("loading objects: %v", err)
    }
    defer objs.Close()
    tp, err := link.Tracepoint("syscalls", "sys_enter_execve", objs.EnterExecve)
    if err != nil {
        log.Fatalf("opening tracepoint: %s", err)
    }
    defer tp.Close()

    // 第二个参数为每一个 CPU 对应的 buffer 大小，必须是 页 的倍数
    rd, err := perf.NewReader(objs.PerfEvents, 2*os.Getpagesize())
```

```
    if err != nil {
        fmt.Println(err)
    }
    defer rd.Close()

    var event enter_execve_t

    log.Println("Waiting for events..")
    var args string
    var pid uint32
    var count int
    for {
        // 读取数据
        record, err := rd.Read()
        if err != nil {
            if errors.Is(err, perf.ErrClosed) {
                return
            }
            log.Printf("reading from perf event reader: %s", err)
            continue
        }

        if record.LostSamples != 0 {
            log.Printf("perf event ring buffer full, dropped %d samples",
record.LostSamples)
            continue
        }

        // 解析到对应的数据结构体
        if err := binary.Read(bytes.NewBuffer(record.RawSample),
binary.LittleEndian, &event); err != nil {
            log.Printf("parsing perf event: %s", err)
            continue
        }
        if pid == 0 {
            pid = event.Pid
        }
        if pid == event.Pid {
            args = args + string(event.Args[:event.Argsize-1]) + " "
        } else {
            fmt.Printf("[INFO] pid: %d, comm: %s, argv: %s\n", event.Pid,
string(event.Comm[:]), strings.Trim(args, " "))
            count = count + 1
            pid = event.Pid
            args = string(event.Args[:]) + " "
        }
    }
}
```

这也就是为什么我在单个 CPU 的机器上不会碰到乱序的问题，因为只有一个生产者。如果在多 CPU 的场景下，可能会出现乱序问题。例如 fork、exec、exit 可能不会以顺序的形式被读取，所以我们在读程序的时候需要额外增添一段代码逻辑来处理这个问题。

[ 注 ]：这其实有点麻烦，例如在本地通过 LRU 构建进程树，父子进程顺序颠倒会导致进程子进程不能按需构建进程数。为了解决，我觉得可能得实现一个长度合适的 优先队列 来解决这个问题，后续会尝试一下并完善代码~

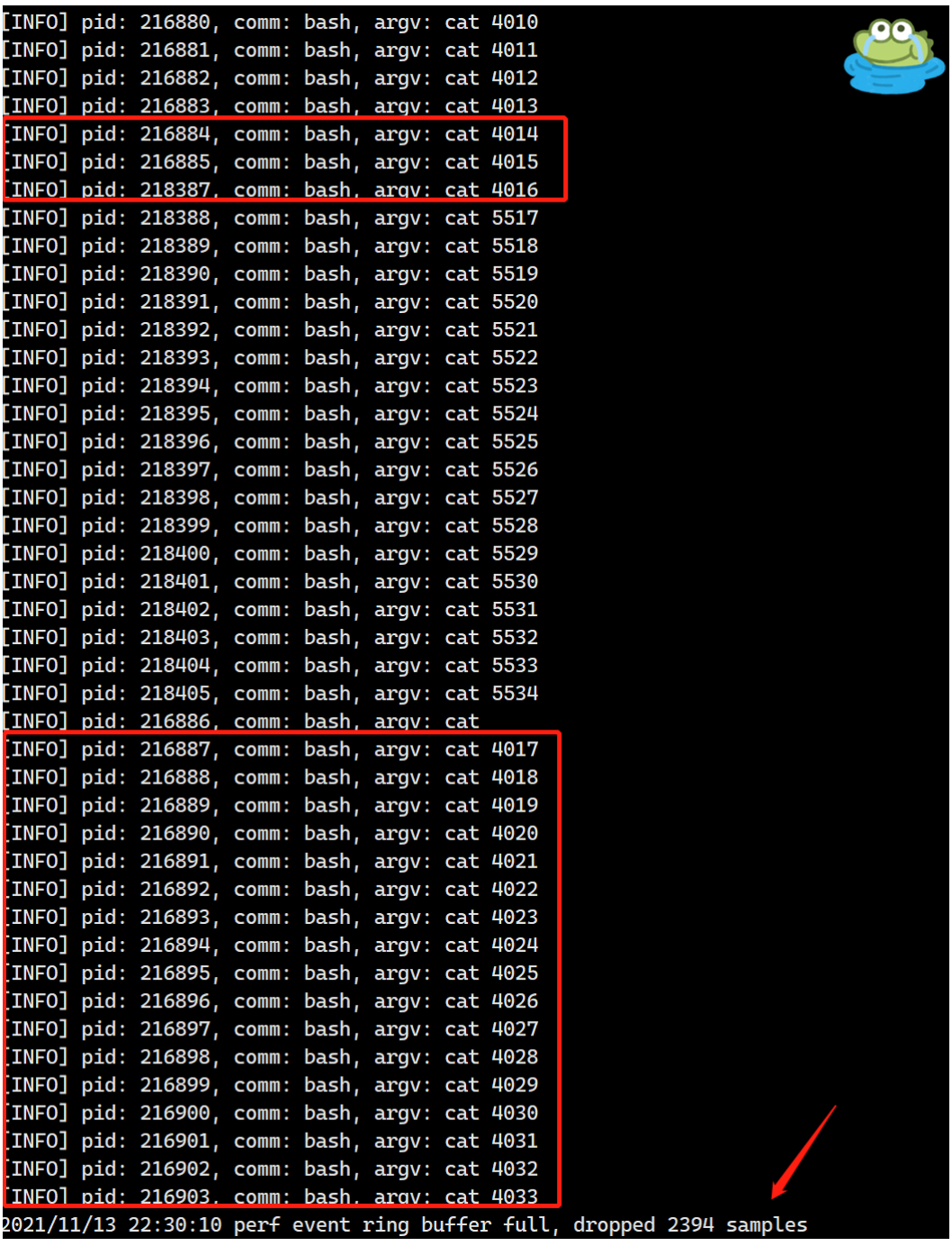
## 验证问题

> 没 google 到任何关于 re-ordering 问题的图片，于是在腾讯云上购买了一个双核的机器，我们简单跑一下 bash。我们测试的bash脚本为：

```
#!/bin/bash

COUNTER=0
while [ $COUNTER -lt 10000 ]
do
    cat "$COUNTER"
    let COUNTER+=1
done
```

把程序跑起来后，我们测试一下上述的 bash 脚本。在若干次测试下，抓到了一段这样的（代码可能有点小 BUG，后续修复，不过不影响我们看问题）

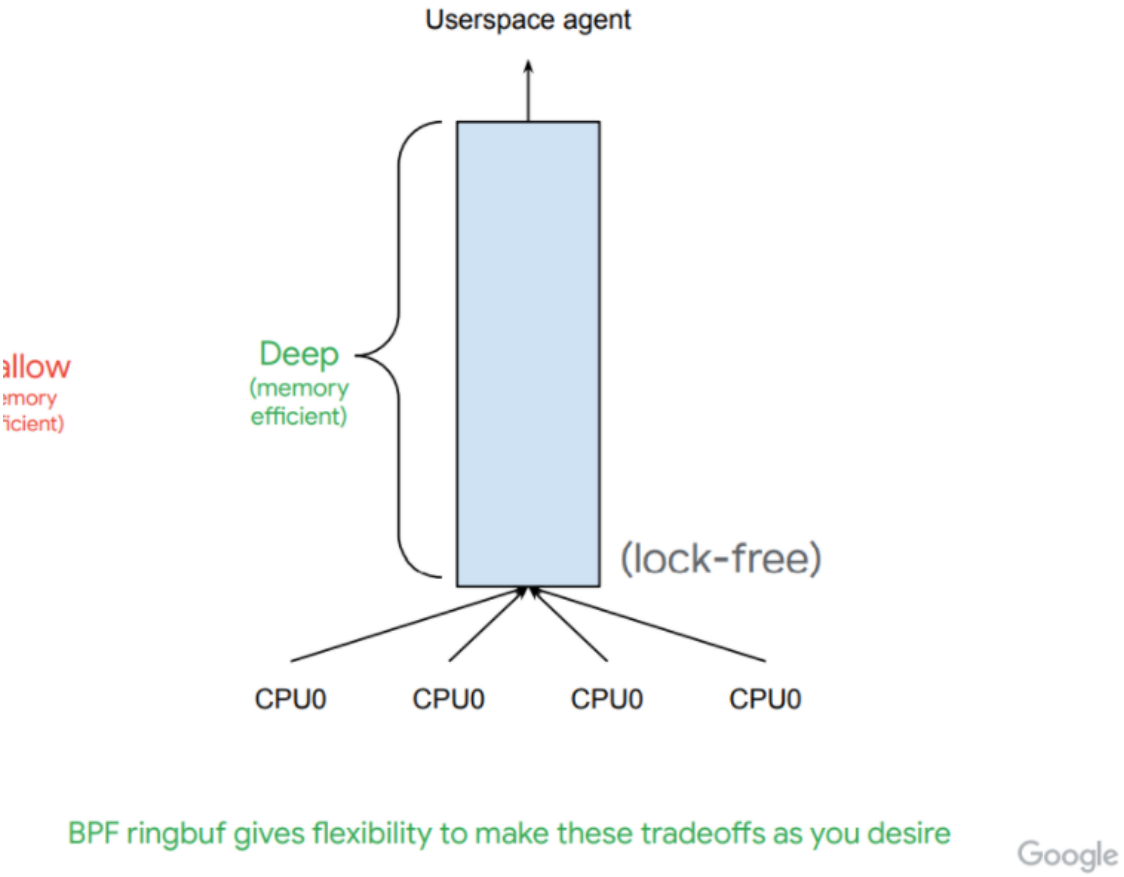


第一出现了乱序问题，下面方块的 cat 4017 应该和上面的 cat 4016 连接。假设在 HIDS 上，我们在本地构建进程树，这样的情况往往会导致我们进程树断裂（父进程子进程的情况）。

第二个就是 buffer 满的问题，当然这个不是 perf event 特有的问题，只是在这里展现一下。

# Ringbuf

> kernel.org
> Ringbuf 解决了上述所说的问题。这里再借用一下 google ppt 中的图片




ringbuf 内部使用了一个轻量的自旋锁，是一个多生产者单消费者的队列，ringbuf 有以下特点：

1. 在多 CPU 的情况下安全
2. 比 perfbuf 有更好的内存使用率和传输速率(参考)，减少内存开销
3. 传输的 data 数据长度可变
4. 因为其基于内存映射的形式不需要额外的内存拷贝或者 syscalls (更高效)

同时也解决了在多个 CPU 下乱序的问题，简单看来像是 perfbuf 的强化版

[ 感想 ]：对于第四点，在字节 HIDS 的群里，看到过类似的说法。他们的开发说后续数据交互方案会换成：新方案实现的类似 ringbuffer + io_uring(相关资料)，实现了memory zero copy, zero  syscall(更多关于 ring buffer 资料，后续会慢慢品读)。

ringbuf 和 perfbuf 在数据传输上的区别，在文章的 （Wasted work and extra data copying）里面提到，在 perf buffer 下至少要经历两次 copy，第一次 copy 数据到本地变量或者直接到 perCpu 的缓存里，再拷贝到 perfbuf 里面。这会有一个问题，如果在 perfbuf 已经满了的情况下，仍然会执行大量的 copy 动作。而在 ringbuf 下用两个指令（reservation 和 submit）解决了这个问题，如果 reservation 失败就代表没有空闲，直接 drop 就行，不会出现和 perfbuf 一样的场景

在 perfbuf 的场景下，为了要抗住瞬间增多的数据，我们需要增大每一个 CPU 上的 buffer，所以在使用 perf 的时候经常会出现两种抉择：要么使用更大内存，要么接受偶尔丢数据，我们以 cilium/ebpf 中的 ringbuf 代码为例子

```c
#include "common.h"
#include "bpf_helpers.h"

char __license[] SEC("license") = "Dual MIT/GPL";

struct event_t {
    u32 pid;
    char comm[80];
};

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24); // 这里的 size 代表的是 buffer 大小，含义不同。同样，同样需要以 页 为单位
} events SEC(".maps");

SEC("kprobe/sys_execve")
int kprobe_execve(struct pt_regs *ctx) {
    u64 id = bpf_get_current_pid_tgid();
    u32 tgid = id >> 32;
    struct event_t *task_info;
    // reserve 占位，如果失败就不继续（上述说的优势点）
    task_info = bpf_ringbuf_reserve(&events, sizeof(struct event_t), 0);
    if (!task_info) {
        return 0;
    }

    task_info->pid = tgid;
    bpf_get_current_comm(&task_info->comm, 80);
    bpf_ringbuf_submit(task_info, 0);
    return 0;
}
```

Golang 中的读取程序(简略)

```go
...
type Event struct {
    PID  uint32
    Comm [80]byte
}

func main() {
    // Name of the kernel function to trace.
    fn := "sys_execve"
    ...
    // Load pre-compiled programs and maps into the kernel.
    objs := bpfObjects{}
    if err := loadBpfObjects(&objs, nil); err != nil {
        log.Fatalf("loading objects: %v", err)
    }
    defer objs.Close()
    kp, err := link.Kprobe(fn, objs.KprobeExecve)
    if err != nil {
        log.Fatalf("opening kprobe: %s", err)
    }
    defer kp.Close()
```

```
    rd, err := ringbuf.NewReader(objs.Events)
    if err != nil {
        log.Fatalf("opening ringbuf reader: %s", err)
    }
    defer rd.Close()
    ...

    log.Println("Waiting for events..")

    for {
        record, err := rd.Read()
        if err != nil {
            if errors.Is(err, ringbuf.ErrClosed) {
                log.Println("Received signal, exiting..")
                return
            }
            log.Printf("reading from reader: %s", err)
            continue
        }
        // Parse the ringbuf event entry into an Event structure.
        var event Event
        if err := binary.Read(bytes.NewBuffer(record.RawSample),
binary.LittleEndian, &event); err != nil {
            log.Printf("parsing ringbuf event: %s", err)
            continue
        }
        log.Printf("pid: %d\tcomm: %s\n", event.PID,
unix.ByteSliceToString(event.Comm[:]))
    }
}
```

# 总结

首先使用的时候有容易混淆的地方：

1. perfbuf 的 buffer size 是在用户态定义的，而 ringbuf 的 size 是在 bpf 程序中预定义的
2. 同上问题，max_entries 的语义， perfbuf 是 buffer 数量(应该是这么理解)，ringbuf 中是单个 buffer 的 size

体验了解一番之后，ringbuf 确实各方面都强于 perfbuf，但是 ringbuf 太新了，kernel version 要求比较高。所以在看 osquery 的依赖，用的也是 perf_event_array，相比来说可能是兼容性更好吧

BPF 下还有很多的映射类型，以及使用等，后续使用到、学习到后会再做分享。