「eBPF」专场

bpf冷升级——让低版本内核用上新特性

丁天琛 阿里云操作系统团队

2023年5月28日

背景

eBPF上游高速发展与工业界生产环境稳定诉求的矛盾：

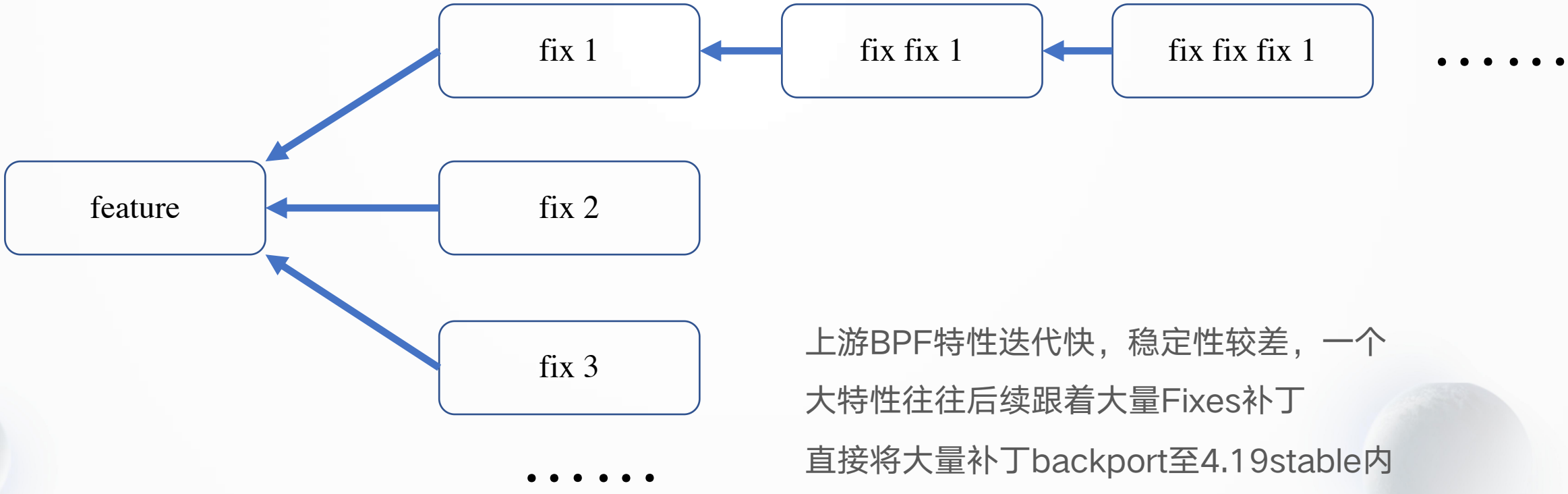
- 上游社区仍然在高速发展，不断支持新特性（扩展 eBPF 使用场景），特性分布在不同内核版本
- 生产环境追求稳定、安全生产，内核版本相对低，不随意变更，存在大量低版本内核
- 用户想使用 eBPF 新特性，但内核版本变更代价非常大，容易引入新的问题
- 生产环境存在多个内核版本，一些版本差异较大，需要维护多套bpf程序（内核特性不同、CO-RE支持不完善等）

需求

以Linux 4.19内核为例，我们收集到的一些用户需求包括：

• BPF stack map / queue map (Linux 4.20)	新增栈和队列类型的map
• BPF spinlock (Linux 5.1)	支持自旋锁
• global data (Linux 5.2)	支持使用全局变量
• bounded loop (Linux 5.3)	支持有限for循环
• tp_btf (Linux 5.5)	tracepoint支持直接访存，无需probe_read
• BPF trampoline (fentry / fexit) (Linux 5.5)	比kprobe/kretprobe更高效，且支持直接访存
• BPF array mmap (Linux 5.5)	支持map数组映射至用户态，直接访问更高效
• BPF map batch ops (Linux 5.6)	支持map批量操作
• BPF ring buffer (Linux 5.8)	BPF自研缓冲区，比perf buffer更高效
• bloom filter map (Linux 5.16)	新增集合类型的map

直接向稳定版内核回合补丁的风险



上游BPF特性迭代快，稳定性较差，一个大特性往往后续跟着大量Fixes补丁

直接将大量补丁backport至4.19stable内核风险较大

直接向稳定版内核回合补丁的风险

anolis4.19 实际案例

CVE修复补丁 from upstream 4.19.y：

verifier判定剪枝更严，搜索更复杂

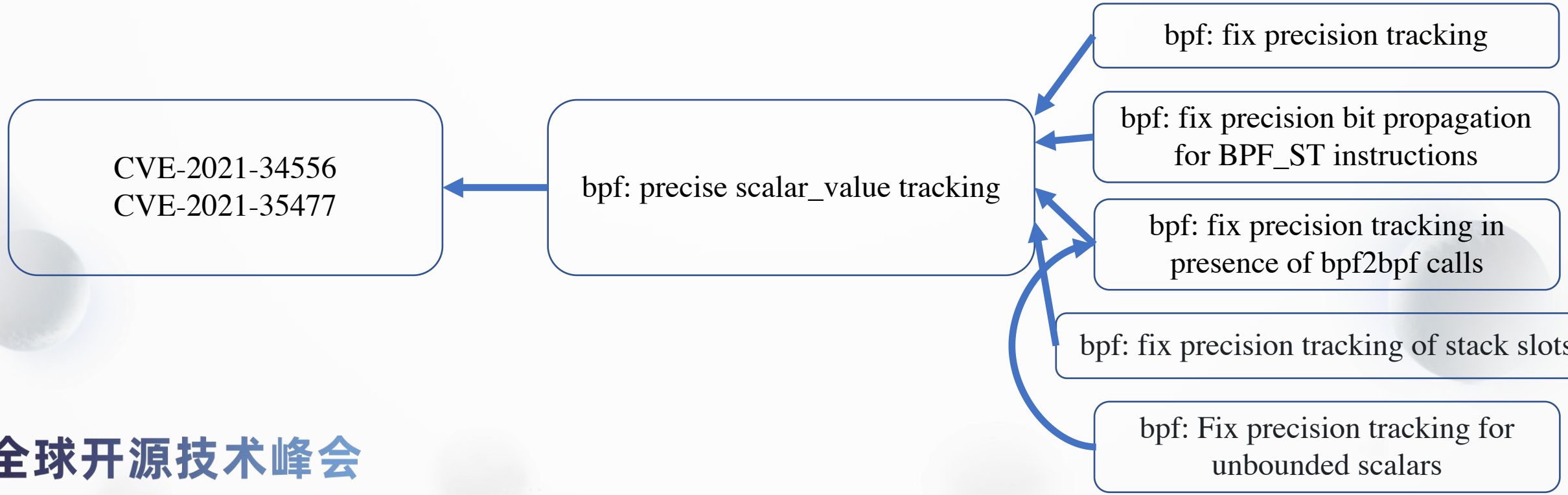
一些程序超出搜索上限，验证拒绝

引入优化补丁：

放宽判定条件，更多剪枝优化

但造成一些路径的遗漏，误判为dead code

后续修复补丁较多，且还存在前置依赖，全部backport风险较大



如何应对这些问题？

直接往稳定版内核backport BPF代码可能存在风险

不同用户对BPF新特性的需求不一样

如果生产环境出现问题，需要有比回滚内核更轻的兜底措施

我们的方案：做成模块！

plugbpf介绍

基于plugsched，将内核BPF子系统模块化

应用场景

- 产品可以根据不同的场景来定制化BPF
- 开发者可以专注于迭代开发，发布的负担比发布操作系统轻

安全

- 不修改内核：修改内核会引入很多对系统的侵入性
- 原子性：集成新BPF模块，要么全部成功，要么全部失败回滚
- 栈安全：当内核处于安全状态时，才进行修改和升级

高效

- 停机时间短：停机时间必须尽可能小（毫秒级）
- 低开销：对CPU、内存、内核几乎不引入任何开销

范围

- 适用于Linux 4.19/5.10 x86_64内核，arm64支持工作进行中…

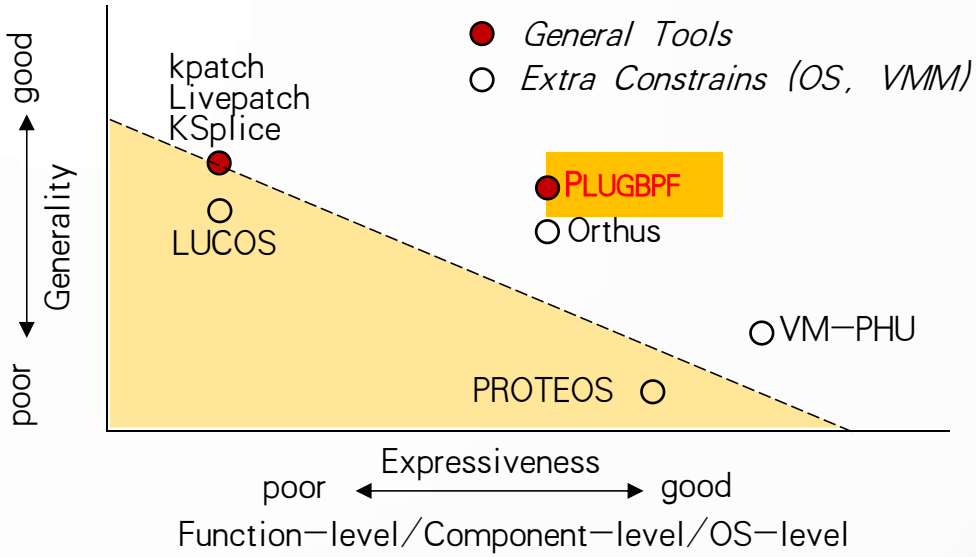
限制

- 考虑到升级前后数据继承的复杂性，需要先移除所有已load的BPF程序

与其他方案对比

业界流行解决方案（Kpatch, livepatch, Ksplice）
- 函数级 (livepatch/kpatch) / 指令级 (Ksplice)
  - 支持修改少量代码（一般少于100 LOC）

其他方案（PROTEOS, Orthus, LUCOS）
- 组件级 (PROTEOS)
  - 支持修改多个函数
  - 依赖微内核，不能用于商业Linux服务器
- 系统级 (Orthus, LUCOS, VM-PHU)
  - 依靠 virtual machine manager (VMM)



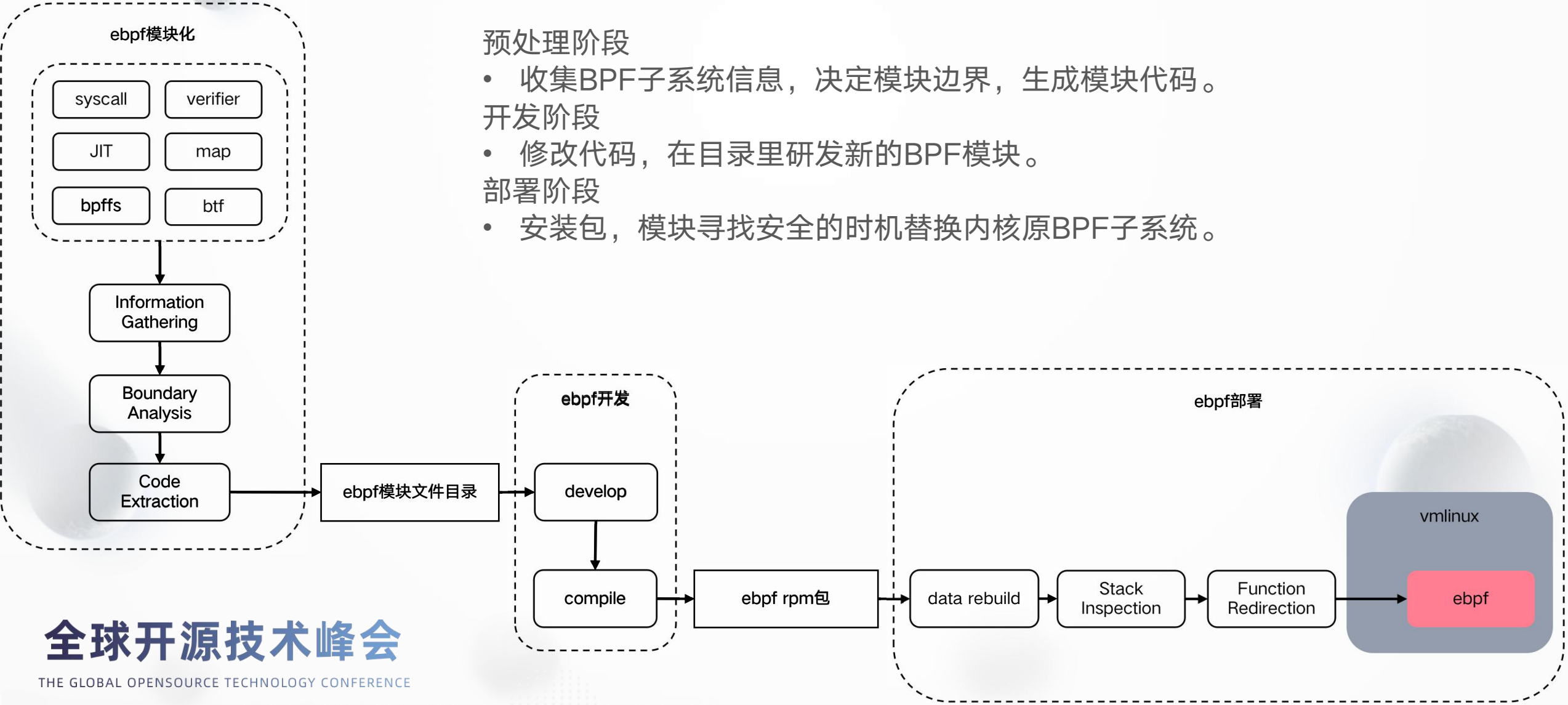
注：引用自plugsched在ASPLOS 23上发表的论文
《Efficient Scheduler Live Update for Linux Kernel with Modularization》

总体设计

预处理阶段
- 收集BPF子系统信息，决定模块边界，生成模块代码。

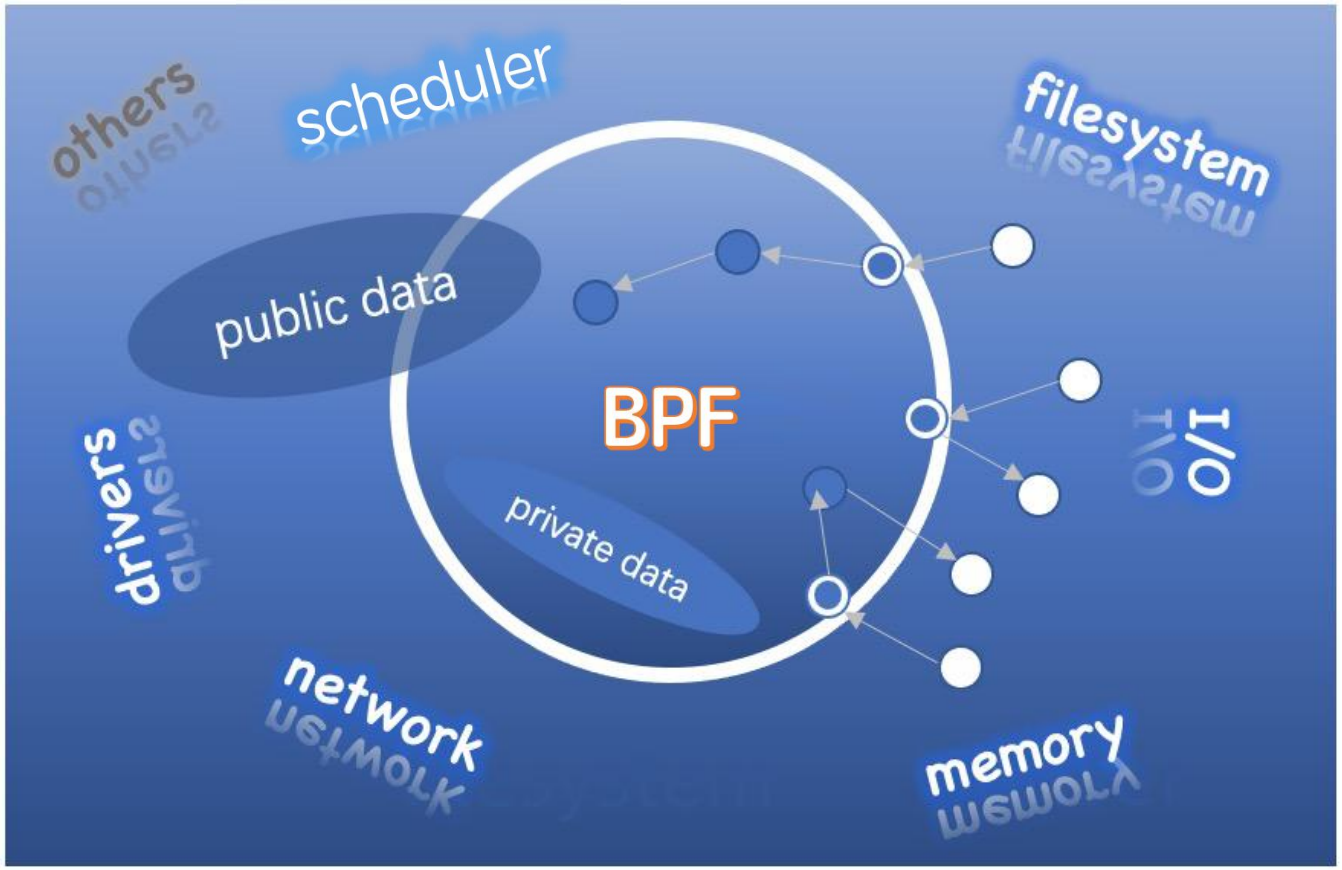
开发阶段
- 修改代码，在目录里研发新的BPF模块。

部署阶段
- 安装包，模块寻找安全的时机替换内核原BPF子系统。



系统设计——模块化

边界划分



Algorithm 1: Boundary Analysis.

(1) Initialization:
Load user-defined configurations ($F_{interface}$) and compiler generated information (F_{mod}, G)
(2) Inflect:
$F' \leftarrow F_{mod} - F_{interface}$
$F'_{external} \leftarrow \varnothing$
while *True* **do** /* Coverage */
  $F^* \leftarrow \varnothing$
  foreach $e_{f_i,f_j} \in G$ **do**
    if $f_i \notin F' \wedge f_i \notin F_{interface} \wedge f_j \in F'$ **then**
      $F^* \leftarrow F^* \cup \{f_j\}$
    end
  end
  if $F^* = \varnothing$ **then**
    break
  end
  $F'_{external} \leftarrow F'_{external} \cup F^*, F' \leftarrow F' - F^*$
end
(3) Save Results:
$F_{internal} \leftarrow F', F_{external} \leftarrow F'_{external}$

系统设计——栈安全检查

栈安全检查
- 所有线程的函数调用路径上没有出现被替换的函数

检查点（stop machine）
- 停止所有CPU的执行（注意不是freeze）
- 停止之后，让指定CPU执行handler

算法优化
- 线程切分
- 二分匹配函数

Algorithm 2: The Procedures of General Stack Inspection

```
1  (1) Initialization:
2  begin
3      Acquire the set of updated functions (F_update);
4      Sort F_update according to the beginning address;
5  end
6  (2) stop_machine:/* main loop of safety checking */
7  Choose a core as primary
8  foreach core in the machine do /* Multi-core parallel */
9      Collect all tasks in the core as T;
10     foreach task t in T do
11         foreach call stack frame f ∈ t do
12             if f ∈ F_update then /* Binary Search */
13                 return Failed
14             end
15         end
16     end
17     if is primary core then
18         Check the return status of all cores;
19         return status;
20     end
21 end
```

系统设计——函数重定向

传统方案

- 替换所有修改了的函数
- Livepatch
  - 运行时重映射另一个虚拟地址到同一物理地址
  - 有 TLB flush的额外开销
- Kpatch
  - 用ftrace子系统hook原函数，劫持到新函数
  - 开销无法忽略，尤其当热函数数量增加到子系统规模
- Ksplice
  - 对比打补丁前后的object，识别修改的函数

plugsched/plugbpf 方案

- 只替换接口函数（Finterface）
- wp flag = 0 (CR0 register) 关闭 page protection mode
- 修改函数的prologue为jmp指令
- wp flag = 1 (CR0 register) 打开 page protection mode
- 利用上下文切换进行核间同步

系统设计——BTF段移植

问题

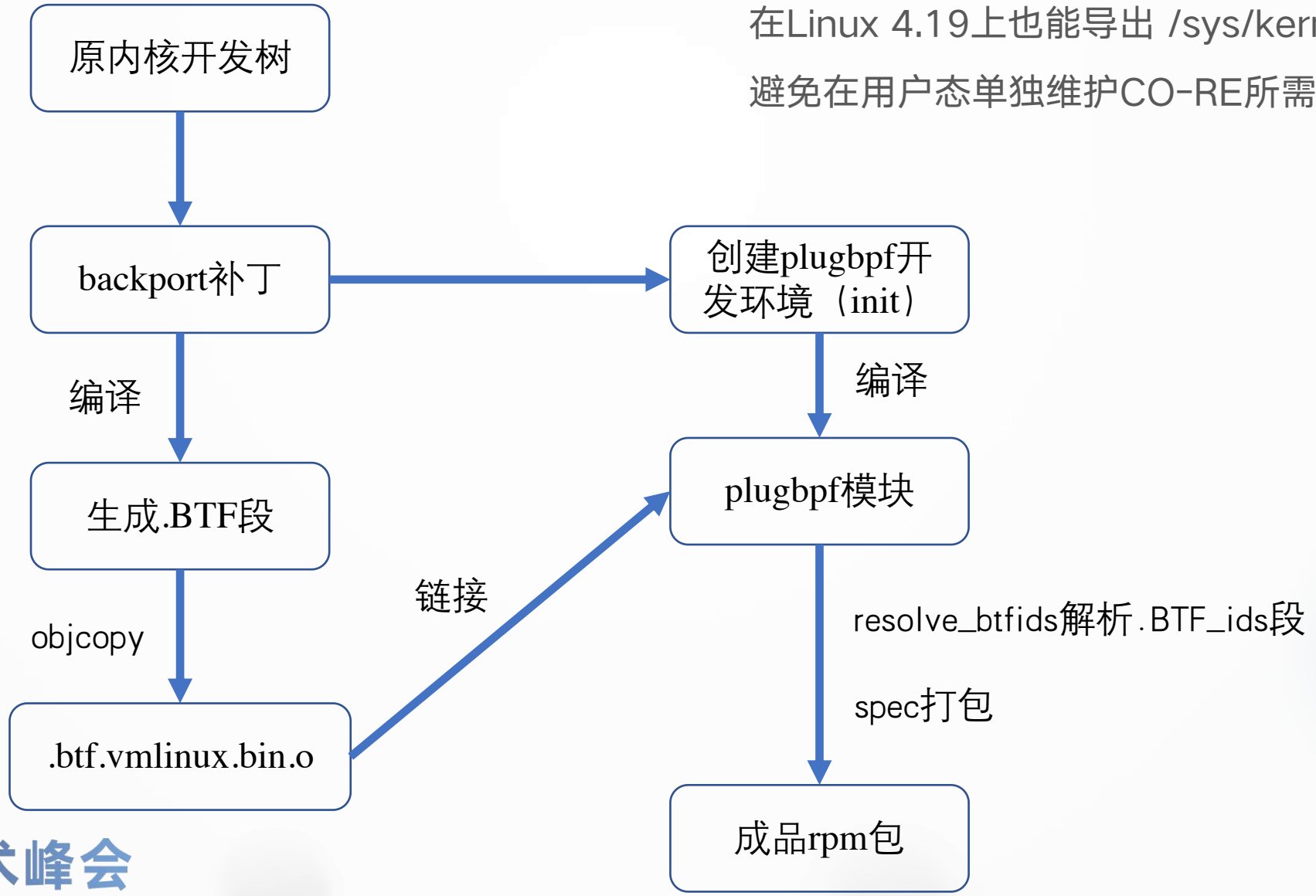
- 在Linux4.19上移植tp_bpf、fentry等特性时需要读取.BTF段
- 在Linux5.10上新增特性时需要更新.BTF及.BTF_ids段

以上问题无法通过Kpatch等函数级修改粒度的方案解决！

系统设计——BTF段移植

在Linux 4.19上也能导出 /sys/kernel/btf/vmlinux

避免在用户态单独维护CO-RE所需的BTF文件

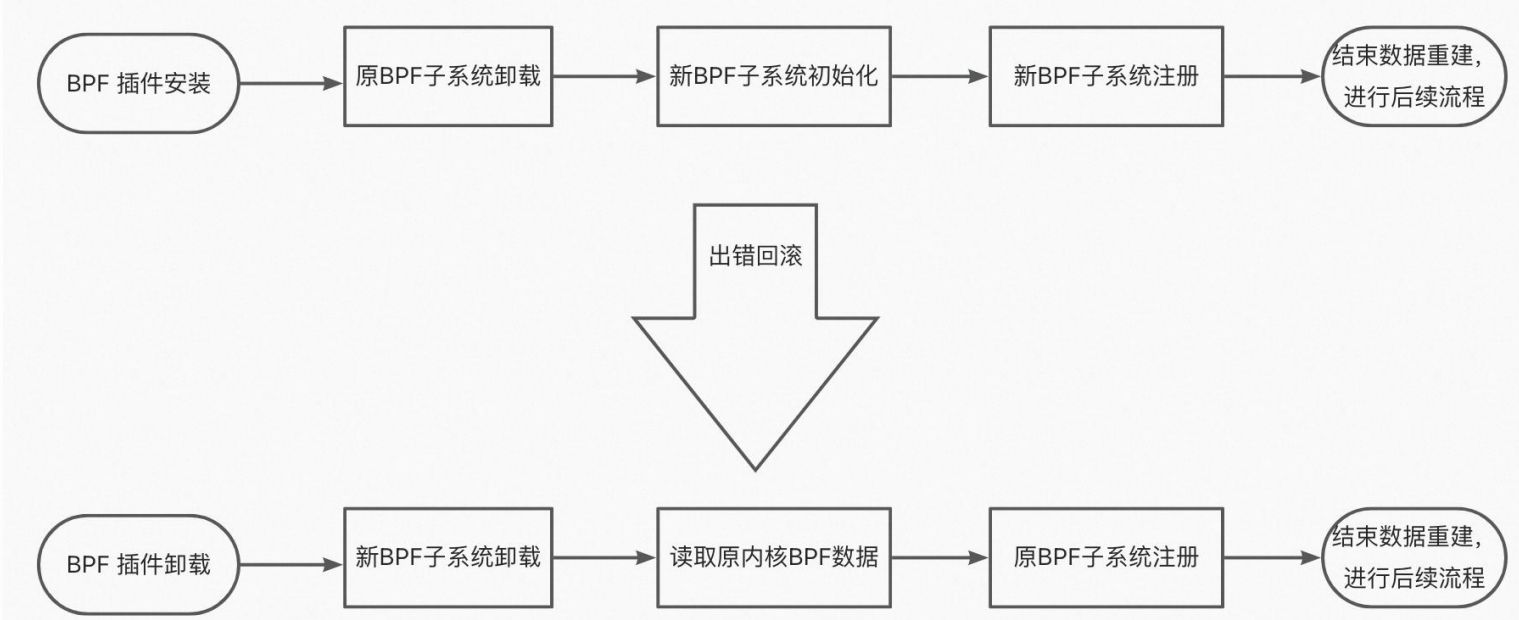


系统设计——数据初始化及接口重新注册

plugbpf模块初始化时，将原内核的BPF子系统注册的文件接口以及向其他子系统注册的数据全部卸载并注册自己模块的数据

卸载时反之

全程通过出错回滚机制保证升级的原子性



应用场景与策略

anolis 4.19

- 在生产环境，根据不同场景需求，通过plugbpf回合新特性
- 经过长时间验证稳定性后的通用特性考虑合入anolis 4.19主线

anolis 5.10

- 除回合新特性外，新增不适合并入主线的自研特性
    - 自研特性需同时维护内核及用户态libbpf库，且要避免与upstream冲突，适合模块化在对应场景按需加载

BPF特性数据

bounded loop (包含Linux 4.19-5.3 verifier重构、之后的bugfix及它们的前置依赖)
79 commits, 23 files, 4000+ insertions, 700+ deletions

tp_btf (包含前置in-kernel BTF、sysfs导出及后续bugfix)
37 commits, 23 files, 1600+ insertions, 160+ deletions

BPF trampoline (包含fentry/fexit/fmod_ret及后续bugfix)
30 commits, 13 files, 1800+ insertions, 150+ deletions

BPF ring buffer (包含前置memlock重构、array mmap及后续bugfix)
21 commits, 20files, 1000+ insertions, 200+ deletions

效果演示

基于anolis 4.19.91-27.1内核

bounded loop：

```
for (i = 0; i < 10; i++) {
    int key = i;
    bpf_map_update_elem(&my_map, &key, &pid, 0);
}
```

```
libbpf:
back-edge from insn 16 to 3

libbpf: -- END LOG --
libbpf: failed to load program 'my_func'
libbpf: failed to load object 'test.kern.o'
load failed, ret 4202544 errno 929427520
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# rpm -i bpf-xxx.rpm
Start plugbpf.service
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# ./test_bounded_loop
libbpf: elf: skipping unrecognized data section(7) .rodata.str1.1
load success!
attach success!
```

效果演示

基于anolis 4.19.91-27.1内核

global data：

```
static volatile u64 a=4;
SEC("raw_tp/sched_wakeup")
int BPF_PROG(my_func, struct task_struct *p)
{
    __sync_fetch_and_add(&a, 1);

    return 0;
}
```

```
libbpf: kernel doesn't support global data
libbpf: failed to load object 'test.kern.o'
load failed, ret 4202544 errno -1410301888
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# rpm -i bpf-xxx.rpm
Start plugbpf.service
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# ./test_global_data
libbpf: elf: skipping unrecognized data section(8) .rodata.str1.1
load success!
attach success!
```

效果演示

基于anolis 4.19.91-27.1内核

BPF ring buffer (from kernel selftests)：

```
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# ./test_progs-no_alu32 -n 75-76
libbpf: kernel doesn't support global data
libbpf: failed to load object 'test_ringbuf'
libbpf: failed to load BPF skeleton 'test_ringbuf': -95
test_ringbuf:FAIL:skel_open_load skeleton open&load failed
#75 ringbuf:FAIL
libbpf: kernel doesn't support global data
libbpf: failed to load object 'test_ringbuf_multi'
libbpf: failed to load BPF skeleton 'test_ringbuf_multi': -95
test_ringbuf_multi:FAIL:skel_open_load skeleton open&load failed
#76 ringbuf_multi:FAIL
Summary: 0/0 PASSED, 0 SKIPPED, 2 FAILED
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# rpm -i /mnt/bpftest/bpf-xxx.rpm
Start plugbpf.service
[root@iZ2ze3g92fv4kf161rq9haZ bpftest]# ./test_progs-no_alu32 -n 75-76
#75 ringbuf:OK
#76 ringbuf_multi:OK
Summary: 2/0 PASSED, 0 SKIPPED, 0 FAILED
```

效果演示

基于anolis 4.19.91-27.1内核

BPF ring buffer (from bpftool feature probe)：

```
100 eBPF map_type sk_storage is NOT available
101 eBPF map_type devmap_hash is NOT available
102 eBPF map_type struct_ops is NOT available
103 eBPF map_type ringbuf is available
104 eBPF map_type inode_storage is NOT available
105 eBPF map_type task_storage is NOT available
106
107 Scanning eBPF helper functions...
108 eBPF helpers supported for program type socket_filter:
109         - bpf_map_lookup_elem
+ 110 +--  6 lines: - bpf_map_update_elem----------------------------------------
116         - bpf_tail_call
117         - bpf_skb_load_bytes
118         - bpf_get_numa_node_id
119         - bpf_get_socket_cookie
120         - bpf_get_socket_uid
121         - bpf_skb_load_bytes_relative
122         - bpf_ringbuf_output
123         - bpf_ringbuf_reserve
124         - bpf_ringbuf_submit
125         - bpf_ringbuf_discard
126         - bpf_ringbuf_query
127 eBPF helpers supported for program type kprobe:
128         - bpf_map_lookup_elem
129         - bpf_map_update_elem
130         - bpf_map_delete_elem
131         - bpf_probe_read
132         - bpf_ktime_get_ns
+ 133 +-- 13 lines: - bpf_trace_printk-----------------------------------------
146         - bpf_get_numa_node_id
147         - bpf_probe_read_str
148         - bpf_perf_event_read_value
149         - bpf_override_return
150         - bpf_get_stack
151         - bpf_get_current_cgroup_id
152         - bpf_ringbuf_output
153         - bpf_ringbuf_reserve
154         - bpf_ringbuf_submit
155         - bpf_ringbuf_discard
156         - bpf_ringbuf_query
157 eBPF helpers supported for program type sched_cls:
158         - bpf_map_lookup_elem
```

```
100 eBPF map_type sk_storage is NOT available
101 eBPF map_type devmap_hash is NOT available
102 eBPF map_type struct_ops is NOT available
103 eBPF map_type ringbuf is NOT available
104 eBPF map_type inode_storage is NOT available
105 eBPF map_type task_storage is NOT available
106
107 Scanning eBPF helper functions...
108 eBPF helpers supported for program type socket_filter:
109         - bpf_map_lookup_elem
+ 110 +--  6 lines: - bpf_map_update_elem----------------------------------------
116         - bpf_tail_call
117         - bpf_skb_load_bytes
118         - bpf_get_numa_node_id
119         - bpf_get_socket_cookie
120         - bpf_get_socket_uid
121         - bpf_skb_load_bytes_relative
122 eBPF helpers supported for program type kprobe:
123         - bpf_map_lookup_elem
124         - bpf_map_update_elem
125         - bpf_map_delete_elem
126         - bpf_probe_read
127         - bpf_ktime_get_ns
+ 128 +-- 13 lines: - bpf_trace_printk-----------------------------------------
141         - bpf_get_numa_node_id
142         - bpf_probe_read_str
143         - bpf_perf_event_read_value
144         - bpf_override_return
145         - bpf_get_stack
146         - bpf_get_current_cgroup_id
147 eBPF helpers supported for program type sched_cls:
148         - bpf_map_lookup_elem
```

总结 & 未来规划

开源工作准备中，子系统级的模块化和热升级细节可参考plugsched代码和论文

- plugsched开源代码仓
  - https://github.com/aliyun/plugsched
  - https://gitee.com/anolis/plugsched
- plugsched论文
  - ASPLOS 2023: Efficient Scheduler Live Update for Linux Kernel with Modularization

未来规划

- 继续调研上游特性的回合支持
- 基于anolis 5.10 支持自研bpf特性
- 支持arm64架构及更多内核版本
- 调研plugbpf继承数据状态的热升级

THANKS

GOTC

龙蜥钉钉交流群

龙蜥 OpenAnolis 社区交流 3 群

7人

扫一扫群二维码，立刻加入该群。

龙蜥微信公众号

全球开

THE GLOBAL OPENSOU